

Features > Flake discovery

Flake discovery

FlakeHub provides several ways of discovering flakes that have been published to the platform.

Search

The FlakeHub UI provides a search bar that enables you to search for flakes using arbitrary search queries, such as "rust" or "nix neovim." You can activate the search bar using $\text{⌘} + \text{K}$ or $\text{Ctrl} + \text{K}$.

You can also search using the [FlakeHub CLI](#)'s `search` command. Here's an example:

```
fh search wayland
```

Listing

All flakes

You can see a listing of all available flakes at flakehub.com/flakes or using `fh`:

```
fh list flakes
```

All organizations

You can see a listing of all public organizations at flakehub.com/orgs or using `fh`:

```
fh list orgs
```

Releases for a flake

You can see a listing of all releases of a flake at a URL with this structure:

```
https://flakehub.com/flake/:org/:project/releases
```

static required

You can also use the `fh`'s [list releases](#) command:

```
fh list releases ublue-os/fleek
```

Matching versions

You can see a listing of all flake versions matching a version constraint at a URL with this structure:

```
https://flakehub.com/label/:org/:project/:version-requirement
```

static required

You can also use the `fh`'s [list versions](#) command:

```
fh list versions DeterminateSystems/flake-checker "0.1.*"
```

By label

You can see a listing of all flakes bearing a specific label at a URL with this structure:

```
https://flakehub.com/label/:label
```

static required

You can also use the `fh`'s [list label](#) command:

```
fh list label nixos
```

Last updated on November 13, 2023

2024 © Determinate Systems, Inc. All rights reserved.

FlakeHub CLI

fh : the FlakeHub CLI

[FlakeHub](#) has a dedicated CLI called `fh` that you can use to interact with FlakeHub. To run the CLI:

```
nix run "https://flakehub.com/f/DeterminateSystems/fh/*.tar.gz"
```

Installation

The `nix run` command above builds `fh` locally on your computer. We will be providing pre-built binaries soon.

NixOS

To make the `fh` CLI readily available on a [NixOS](#) system:

```
{
  description = "My NixOS config.";

  inputs.fh.url = "https://flakehub.com/f/DeterminateSystems/fh/*.tar.gz";
  inputs.nixpkgs.url = "https://flakehub.com/f/NixOS/nixpkgs/0.2305.*.tar.gz";

  outputs = { nixpkgs, fh, ... } @ inputs: {
    nixosConfigurations.nixos = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        {
          environment.systemPackages = [ fh.packages.x86_64-linux.default ];
        }

        # ... the rest of your modules here ...
      ];
    };
  };
}
```

```
};  
}
```

Interface

`fh` enables you to do a variety of things:

| Command | What it does | Example |
|-------------------------------|---|--|
| add | Adds a flake input to a <code>flake.nix</code> file | <code>fh add ublue-os/fleek</code> |
| completion | Provides auto-completion scripts for a variety of common shells | <code>fh completion bash</code> |
| init | Creates a new <code>flake.nix</code> file for you using a combination of the contents of your project and your user input | <code>fh init</code> |
| list flakes | Lists all public flakes on FlakeHub | <code>fh list flakes</code> |
| list orgs | Lists all public organizations on FlakeHub | <code>fh list orgs</code> |
| list releases | Lists all public releases for a flake | <code>fh list releases DeterminateSystems/nuenv</code> |
| list versions | Lists all public releases for a flake that match the provided version requirement | <code>fh list versions NixOS/nixpkgs "0.2305.*"</code> |
| search | Lists all flakes that match the provided search query | <code>fh search "home manager"</code> |

Add

`fh add` adds the most current release of the specified flake to your `flake.nix` and updates the `outputs` function to accept it. This would add the current release of [Nixpkgs](#) to your flake:

```
fh add nixos/nixpkgs
```

The resulting `flake.nix` would look something like this:

```
{
  description = "My new flake.";

  inputs.nixpkgs.url = "https://flakehub.com/f/NixOS/nixpkgs/0.2305.490449.tar.gz";

  outputs = { nixpkgs, ... } @ inputs: {
    # Fill in your outputs here
  };
}
```

Shell completion

You can generate shell completion scripts using the `fh completion` command:

```
fh completion <shell>
```

Here's an example:

```
fh completion bash
```

These shells are supported:

- [Bash](#)
- [Elvish](#)
- [Fish](#)

- [Powershell](#)
- [zsh](#)

Initialize flakes from scratch

`fh init` generates a new `flake.nix` file for you using a combination of:

1. Your responses to interactive questions
2. The contents of the repository in which you run the command.

To create a `flake.nix`, navigate to the directory where you want to create it and run `fh init` (or specify a different directory using the `--root` option). Respond to the prompts it provides you and at the end `fh` will write a `flake.nix` to disk.

`fh init` has built-in support for these languages:

- [Elm](#)
- [Go](#)
- [Java](#)
- [JavaScript](#)
- [PHP](#)
- [Python](#)
- [Ruby](#)
- [Rust](#)
- [Zig](#)



`fh init` operates on a best-guess basis and is opinionated in its suggestions. It's intended less as a comprehensive flake creation solution and more as a helpful kickstarter.

Search

You can search publicly listed flakes using the `fh search` command and passing in a search query. Here's an example:

```
fh search rust
```

| Flake | FlakeHub URL |
|---------------------------|---|
| astro/deadnix | https://flakehub.com/flake/astro/deadnix |
| carlthome/ml-runtimes | https://flakehub.com/flake/carlthome/ml-runtimes |
| ipetkov/crane | https://flakehub.com/flake/ipetkov/crane |
| kamadorueda/alejandra | https://flakehub.com/flake/kamadorueda/alejandra |
| nix-community/fenix | https://flakehub.com/flake/nix-community/fenix |
| nix-community/lanzaboote | https://flakehub.com/flake/nix-community/lanzaboote |
| nix-community/nix-init | https://flakehub.com/flake/nix-community/nix-init |
| nix-community/nixpkgs-fmt | https://flakehub.com/flake/nix-community/nixpkgs-fmt |
| nix-community/patsh | https://flakehub.com/flake/nix-community/patsh |
| ryancn/nyoom | https://flakehub.com/flake/ryancn/nyoom |

`fh search` supports arbitrary search strings. An example:

```
fh search "rust nixos"
```

List

`fh` enables you to list a variety of FlakeHub resources.

List flakes

```
fh list flakes
```

```
+-----
```

| Flake | FlakeHub URL |
|-------------------------|---|
| ajaxbits/audiobookshelf | https://flakehub.com/flake/ajaxbits/audiobookshelf |
| ajaxbits/tone | https://flakehub.com/flake/ajaxbits/tone |
| astro/deadnix | https://flakehub.com/flake/astro/deadnix |
| ... | ... |

List organizations

```
fh list orgs
```

| Organization | FlakeHub URL |
|--------------|---|
| ajaxbits | https://flakehub.com/org/ajaxbits |
| astro | https://flakehub.com/org/astro |
| ... | ... |

List releases

`fh list releases` provides a list of a flake's [releases](#).

```
fh list releases nixos/nixpkgs
```

| Version |
|---|
| 0.1.428801+rev-2788904d26dda6cfa1921c5abb7a2466ffe3cb8c |
| 0.1.429057+rev-42337aad353c5efff4382d7bf99deda491459845 |
| 0.1.429304+rev-27ccd29078f974ddbdd7edc8e38c8c8ae003c877 |
| 0.1.429553+rev-5dc7114b7b256d217fe7752f1614be2514e61bb8 |
| 0.1.429868+rev-a115bb9bd56831941be3776c8a94005867f316a7 |
| ... |

List versions

You can list [versions](#) of a flake by passing the flake name and a version requirement to `fh list versions`:

```
fh list versions <flake> <version_req>
```

Here's an example:

```
fh list versions DeterminateSystems/flake-checker "0.1.*"
```

```
+-----  
| Simplified version  FlakeHub URL  
+-----  
| 0.1.0              https://flakehub.com/flake/DeterminateSystems/flake-checker/0.1.0  
| 0.1.1              https://flakehub.com/flake/DeterminateSystems/flake-checker/0.1.1  
| 0.1.2              https://flakehub.com/flake/DeterminateSystems/flake-checker/0.1.2  
| ...                ...  
+-----
```

List by label

You can list flakes by label using the `fh list label` command:

```
fh list label <label>
```

Here's an example:

```
fh list label python
```

```
+-----+
```

| Flake | FlakeHub URL |
|--------------------------|---|
| nix-community/poetry2nix | https://flakehub.com/flake/nix-community/poetry2nix |

Last updated on November 13, 2023

2024 © Determinate Systems, Inc. All rights reserved.

Concepts > Flake schemas

Flake schemas

[Flake schemas](#) are a way to tell Nix about the structure of your [flake's outputs](#). If you have outputs that are not part of the default schema definitions (see the [flake-schemas README](#) for the default flake output types), you can make your own schema, which will show up on your flake's FlakeHub page.

To add flake schemas to your flake, first add a flake input for the [flake-schemas](#) flake using [fh](#):

```
fh add DeterminateSystems/flake-schemas
```

Or manually:

flake.nix

```
{  
  inputs = {  
    flake-schemas = "https://flakehub.com/f/DeterminateSystems/flake-schemas/*.tar.gz";  
  
    # other inputs  
  };  
}
```

Then add a [schemas](#) output:

flake.nix

```
{  
  outputs = { flake-schemas, ... }: {  
    inherit (flake-schemas) schemas;  
  
    # other outputs  
  };  
}
```

Custom flake schemas

Here is an example flake that defines a schema for its `superSpecialPackage` output:

```
{  
  inputs.nixpkgs.url = "https://flakehub.com/f/NixOS/nixpkgs/*.tar.gz";  
  
  outputs = { ... } @ inputs:  
  let  
    mkChildren = children: { inherit children; };  
  in  
  {  
    superSpecialPackage.x86_64-linux.default = inputs.nixpkgs.legacyPackages.x86_64.  
  
    schemas.superSpecialPackage = {  
      version = 1;  
      doc = "The `superSpecialPackage` output defines a [super special package](ht:  
      inventory = output: mkChildren (  
        builtins.mapAttrs  
        (systemType: packagesForSystem:  
         {  
           forSystems = [ systemType ];  
           children = builtins.mapAttrs  
             (packageName: package:  
              {  
                forSystems = [ systemType ];  
                what = "super special package";  
                shortDescription = package.meta.description or "";  
                derivation = package;  
                evalChecks.isDerivation =  
                  package.type or null == "derivation"  
              })  
            })  
          );  
        );  
      );  
    );  
  );  
}
```

```
    && package ? drvPath;
  })
packagesForSystem;
})
output);
};

}
}
```

Last updated on December 8, 2023

agenix-shell

`agenix-shell` is the [agenix](#) counterpart for `devShell`. It provides options used to define a `shellHook` that, when added to your `devShell`, automatically decrypts secrets and export them.

Here's a template you can start from.

Installation

To use these options, add to your flake inputs:

```
agenix-shell.url = "github:aciceri/agenix-shell";
```

and inside the `mkFlake`:



Run `nix flake lock` and you're set.

Options

`agenix-shell.identityPaths`

`agenix-shell.secrets`

`agenix-shell.secrets.<name>.file`

`agenix-shell.secrets.<name>.mode`

`agenix-shell.secrets.<name>.path`

`agenix-shell.secretsPath`

[perSystem.agenix-shell.package](#)

[perSystem.agenix-shell.installationScript](#)

agenix-shell.identityPaths

Path to SSH keys to be used as identities in age decryption.

Type: list of string

Default:

```
[  
    "$HOME/.ssh/id_ed25519"  
    "$HOME/.ssh/id_rsa"  
]
```

Declared by:

[agenix-shell/flakeModules/agenix-shell.nix](#)



agenix-shell.secrets

Attrset of secrets.

Type: attribute set of (submodule)

Example:

```
{  
    foo.file = "secrets/foo.age";  
    bar = {  
        file = "secrets/bar.age";  
        mode = "0440";  
    };  
}
```

Declared by:

[agenix-shell/flakeModules/agenix-shell.nix](#)

agenix-shell.secrets.<name>.file

Age file the secret is loaded from.

Type: path

Declared by:

[agenix-shell/flakeModules/agenix-shell.nix](#)

agenix-shell.secrets.<name>.mode

Permissions mode of the decrypted secret in a format understood by chmod.

Type: string

Default: "0400"

Declared by:

[~~agenix-shell/flakeModules/agenix-shell.nix~~](#)



agenix-shell.secrets.<name>.path

Path where the decrypted secret is installed.

Type: string

Default: "\${config.agenix-shell.secretsPath}/<name>"

Declared by:

[agenix-shell/flakeModules/agenix-shell.nix](#)

agenix-shell.secretsPath

Where the secrets are created.

Type: string

Default: "/run/user/\$(id -u)/agenix-shell/\$(git rev-parse --show-toplevel | xargs basename)"

Declared by:

[agenix-shell/flakeModules/agenix-shell.nix](#)

perSystem.agenix-shell.package

The age package to use.

Type: package

Default: pkgs.rage

Declared by:

[agenix-shell/flakeModules/agenix-shell.nix](#)



perSystem.agenix-shell.installationScript

Script that exports secrets as variables, it's meant to be used as hook in `devShells`.

Type: package

Default: An automatically generated package

Declared by:

[agenix-shell/flakeModules/agenix-shell.nix](#)

Best Practices for Module Writing

Like in NixOS, writing a configuration is quite different from writing a reusable module.

In a configuration, you may take shortcuts which have little impact, but shortcuts in a reusable module lead to surprises for your module's users.

Do not make assumptions about inputs

The inputs are controlled by the user, so your module should make no assumption about which inputs are present. This way, the user is free to, for example, bundle up their inputs into a distribution, such as a company platform module.

Don't traverse inputs



By scanning through all the `inputs`, you cause two kinds of problems

- You trigger the fetching of all direct dependencies, even though some may not need to be fetched.
- You are making an assumption about the role in which an input is used.

By recursing into inputs, you make the problem literally **exponentially** worse:

- Your module logic becomes susceptible to changes deep inside your dependencies' dependencies. Whereas you might have gotten away with an assumption about the role of direct dependencies, making the same assumptions about dependencies and dependencies' dependencies is unlikely to work out well.
- You trigger the fetching of potentially all transitive dependencies. Instead of a performance inconvenience, we now have a ecosystem-wide scaling problem.

Also note that even if you don't explicitly recurse into the transitive inputs, this behavior still arises if your inputs don't adhere to the rule.

Furthermore it has been observed that lock files can grow indefinitely when mutually dependent flakes don't use `follows` to remove the older version of themselves from the

inputs graph.

Use perSystem

When integrating an existing library, it might be easy to add its options in the top level namespace only, as it might already expose a whole-flake interface. However, as most build and test work is done in `perSystem`, users expect to be able to use it in that context. See also [perSystem first \(custom flake attribute\)](#).

Bundle with existing flake

Most modules are about some piece of software that it integrates. Ideally the flake module is bundled into the same flake. This simplifies the wiring that users have to do, especially when they want to use a patched version. It's also a bit more efficient as far as fetching is concerned.



Do not use overlay general option names

Most modules will put all their options inside a "namespace" named after their module instead. This way, option path collisions are unlikely to occur.

For example: `perSystem.treefmt.programs`, not `perSystem.programs`.

Cheat Sheet

Flake-parts offers a couple of ways to access the same thing. This gives you freedom to pick the most convenient syntax for a use case.

This page is for you to get a feel for what's what.

Get a locally defined package

Getting the locally defined `hello` package on/for an `x86_64-linux` host:

On the command line

```
nix build .#hello
```

In perSystem

```
config.packages.hello
```

```
self'.packages.hello
```

The `self'` parameter is derived from the flake `self`, which may benefit from evaluation caching in the future.

The `config` parameter is conceptually simpler and lets you access all options inside `perSystem`, including unexposed ones if you're into defining such options.

In the top level

Note

Anything you can do at the top level, you can do in `perSystem` as well, although you may have to `@` match those module arguments.

For example, add change the top level function header to e.g. `toplevel@{ config, ... }: /*...*/` so you can access `toplevel.config` despite plain `config` being shadowed by `perSystem = { config, ... }: /*...*/`.

Examples:

```
(getSystem "x86_64-linux").packages.hello
```

```
withSystem "x86_64-linux" ({ config, ... }:
  config.packages.hello
)
```

```
self.x86_64-linux.packages.hello
```

```
allSystems.x86_64-linux.packages.hello
```

~~all~~systems may not be future proof if Nix starts to allow building for all systems. An  opened up system is incompatible with enumerated systems as required by an attribute set.

Define a Module in a Separate File

To avoid writing huge files, you'll want to separate some logic into modules.

When you do so, you'll notice that you've cut access to the lexical scope. You can't access any of the variables in `flake.nix` anymore.

This problem has two possible solutions.

Factor it out

In short, write an inline module in `flake.nix` (or in a flake module), that `imports` the separate module file and also forms a bridge between the `flake.nix` scope and the option values. This way the majority of the module can be in a separate file.

In the separate file, replace all variables that would come from the lexical scope by new options and reference those through `config`. In `flake.nix` fill in the missing defaults.

Example:

`nixos-module.nix`

```
{ lib, config, ... }: {
  options = {
    services.foo = {
      package = mkOption {
        defaultText = lib.literalMD "`packages.default` from the foo flake";
      };
    };
  };
  config = ...;
}
```

Flake module:

```
{ withSystem, ... }: {
  flake.nixosModules.default = { pkgs, ... }: {
    imports = [ ./nixos-module.nix ];
    services.foo.package = withSystem pkgs.stdenv.hostPlatform.system ({ config,
```

```

    config.packages.default
);
};

}

```

For your module users' overriding needs, it's best to make the options as specific as possible; e.g. not a `foo.flake` option, but `foo.package`. You'll find that most modules only need one or two such options.

importApply

The `importApply` function can pass extra variables to a module to import.

Instead of loading a file containing a module, it loads file containing *a function to a module*, and applies it.

`nixos-module.nix`

```

{ localFlake, withSystem }:
{ lib, config, ... }: {
  <                                     >
  options = {
    services.foo = {
      package = mkOption {
        default = withSystem ({ config, ... }: config.packages.default);
        defaultText = lib.literalMD "`packages.default` from the foo flake";
      };
    };
  };
  config = ... use localFlake ...;
}

```

Flake module:

```

{ flake-parts-lib, self, withSystem, ... }:
let
  inherit (flake-parts-lib) importApply;
in
{
  flake.nixosModules.default = importApply ./nixos-module.nix { localFlake = sel
}

```

See Also

- [Dogfooding a Reusable Flake Module](#), which helps avoid an infinite recursion.



Defining a custom flake output attribute

When should I declare a custom flake output?

With flakes, most of the time, you can use the attribute names suggested by the Nix CLI, NixOS, etc, which already have options in flake-parts, or existing modules you can import.

However, if you're doing something that doesn't fit those labels, you may consider adding a custom attribute to the flake outputs. Your custom attribute may not be easy to use with the existing tools, but that may be expected, if you're writing a new tool or something else that's novel.

How do I do it?



If your custom output attribute is a one-off because you need to do something special in a single project, all you have to do is define a value in the [flake option](#).

However, if you want it to be reusable and integrate well, you should declare an option for it, and you could perhaps provide a bit of support logic if that makes sense to do.

perSystem first

Integrating with `perSystem` is highly recommended, because that's where users expect things like packages to be defined. You can bring things that are defined in `perSystem` to the flake outputs in the same way [packages.nix](#) does it.

If your application doesn't follow the same pattern, but you want users to define things in `perSystem`, you may read the top level `config.allSystems` (internal) option. You can read it in the definition for a new option in the `flake` submodule so that its value is added to the flake outputs.

config.flake

[flake](#) is an RFC42-style module, which means that it both has options and it allows arbitrary attributes to be defined in the config, without having to declare an option first. Declaring an option is recommended though, for the purposes of documentation, type checking, and allowing multiple config definitions to be merged into a single output value, if applicable.

Get Help

This is an advanced use case. Feel free to ask questions in [#hercules-ci:matrix.org](#).



devenv

[devenv](#) provides a devShell with many options, and container packages.

See also the [setup guide at devenv.sh](#).

Installation

To use these options, add to your flake inputs:

```
devenv.url = "github:cachix/devenv";
```

and inside the `mkFlake`:

```
imports = [  
    < inputs.devenv.flakeModule  
];  
    >
```

Run `nix flake lock` and you're set.

Options

`perSystem.devenv.shells`

`perSystem.devenv.shells.<name>.packages`

`perSystem.devenv.shells.<name>.certificates`

`perSystem.devenv.shells.<name>.container.isBuilding`

`perSystem.devenv.shells.<name>.containers`

`perSystem.devenv.shells.<name>.containers.<name>.copyToRoot`

`perSystem.devenv.shells.<name>.containers.<name>.defaultCopyArgs`

```
perSystem.devenv.shells.<name>.containers.<name>.entrypoint
perSystem.devenv.shells.<name>.containers.<name>.isBuilding
perSystem.devenv.shells.<name>.containers.<name>.name
perSystem.devenv.shells.<name>.containers.<name>.registry
perSystem.devenv.shells.<name>.containers.<name>.startupCommand
perSystem.devenv.shells.<name>.containers.<name>.version
perSystem.devenv.shells.<name>.devcontainer.enable
perSystem.devenv.shells.<name>.devcontainer.settings
perSystem.devenv.shells.<name>.devcontainer.settings.customizations.vscode.extensions
perSystem.devenv.shells.<name>.devcontainer.settings.image
perSystem.devenv.shells.<name>.devcontainer.settings.overrideCommand
perSystem.devenv.shells.<name>.devcontainer.settings.updateContentCommand >
perSystem.devenv.shells.<name>.devenv.flakesIntegration
perSystem.devenv.shells.<name>.devenv.latestVersion
perSystem.devenv.shells.<name>.devenv.warnOnNewVersion
perSystem.devenv.shells.<name>.difftastic.enable
perSystem.devenv.shells.<name>.enterShell
perSystem.devenv.shells.<name>.env
perSystem.devenv.shells.<name>.hosts
perSystem.devenv.shells.<name>.hostsProfileName
perSystem.devenv.shells.<name>.infoSections
perSystem.devenv.shells.<name>.languages.ansible.enable
perSystem.devenv.shells.<name>.languages.ansible.package
```

```
perSystem.devenv.shells.<name>.languages.c.enable
perSystem.devenv.shells.<name>.languages.clojure.enable
perSystem.devenv.shells.<name>.languages.cplusplus.enable
perSystem.devenv.shells.<name>.languages.crystal.enable
perSystem.devenv.shells.<name>.languages.cue.enable
perSystem.devenv.shells.<name>.languages.cue.package
perSystem.devenv.shells.<name>.languages.dart.enable
perSystem.devenv.shells.<name>.languages.dart.package
perSystem.devenv.shells.<name>.languages.deno.enable
perSystem.devenv.shells.<name>.languages.dotnet.enable
perSystem.devenv.shells.<name>.languages.dotnet.package
perSystem.devenv.shells.<name>.languages.elixir.enable
< perSystem.devenv.shells.<name>.languages.elixir.package >
perSystem.devenv.shells.<name>.languages.elm.enable
perSystem.devenv.shells.<name>.languages.erlang.enable
perSystem.devenv.shells.<name>.languages.erlang.package
perSystem.devenv.shells.<name>.languages.gawk.enable
perSystem.devenv.shells.<name>.languages.go.enable
perSystem.devenv.shells.<name>.languages.go.package
perSystem.devenv.shells.<name>.languages.haskell.enable
perSystem.devenv.shells.<name>.languages.haskell.package
perSystem.devenv.shells.<name>.languages.haskell.languageServer
perSystem.devenv.shells.<name>.languages.java.enable
perSystem.devenv.shells.<name>.languages.java.gradle.enable
```

```
perSystem.devenv.shells.<name>.languages.java.gradle.package  
perSystem.devenv.shells.<name>.languages.java.jdk.package  
perSystem.devenv.shells.<name>.languages.java.maven.enable  
perSystem.devenv.shells.<name>.languages.java.maven.package  
perSystem.devenv.shells.<name>.languages.javascript.enable  
perSystem.devenv.shells.<name>.languages.javascript.package  
perSystem.devenv.shells.<name>.languages.julia.enable  
perSystem.devenv.shells.<name>.languages.julia.package  
perSystem.devenv.shells.<name>.languages.kotlin.enable  
perSystem.devenv.shells.<name>.languages.lua.enable  
perSystem.devenv.shells.<name>.languages.lua.package  
perSystem.devenv.shells.<name>.languages.nim.enable  
 perSystem.devenv.shells.<name>.languages.nim.package  
  
perSystem.devenv.shells.<name>.languages.nix.enable  
perSystem.devenv.shells.<name>.languages.ocaml.enable  
perSystem.devenv.shells.<name>.languages.ocaml.packages  
perSystem.devenv.shells.<name>.languages.perl.enable  
perSystem.devenv.shells.<name>.languages.php.enable  
perSystem.devenv.shells.<name>.languages.php.package  
perSystem.devenv.shells.<name>.languages.php.packages  
perSystem.devenv.shells.<name>.languages.php.packages.composer  
perSystem.devenv.shells.<name>.languages.php.extensions  
perSystem.devenv.shells.<name>.languages.php.fpm.extraConfig  
perSystem.devenv.shells.<name>.languages.php.fpm.phpOptions
```

perSystem.devenv.shells.<name>.languages.php.fpm.pools
perSystem.devenv.shells.<name>.languages.php.fpm.pools.<name>.extraConfig
perSystem.devenv.shells.<name>.languages.php.fpm.pools.<name>.listen
perSystem.devenv.shells.<name>.languages.php.fpm.pools.<name>.phpEnv
perSystem.devenv.shells.<name>.languages.php.fpm.pools.<name>.phpOptions
perSystem.devenv.shells.<name>.languages.php.fpm.pools.<name>.phpPackage
perSystem.devenv.shells.<name>.languages.php.fpm.pools.<name>.settings
perSystem.devenv.shells.<name>.languages.php.fpm.pools.<name>.socket
perSystem.devenv.shells.<name>.languages.php.fpm.settings
perSystem.devenv.shells.<name>.languages.php.ini
perSystem.devenv.shells.<name>.languages.php.version
perSystem.devenv.shells.<name>.languages.purescript.enable
 perSystem.devenv.shells.<name>.languages.purescript.package 
perSystem.devenv.shells.<name>.languages.python.enable
perSystem.devenv.shells.<name>.languages.python.package
perSystem.devenv.shells.<name>.languages.python.poetry.enable
perSystem.devenv.shells.<name>.languages.python.poetry.package
perSystem.devenv.shells.<name>.languages.python.venv.enable
perSystem.devenv.shells.<name>.languages.r.enable
perSystem.devenv.shells.<name>.languages.r.package
perSystem.devenv.shells.<name>.languages.racket.enable
perSystem.devenv.shells.<name>.languages.racket.package
perSystem.devenv.shells.<name>.languages.raku.enable
perSystem.devenv.shells.<name>.languages.robotframework.enable

perSystem.devenv.shells.<name>.languages.robotframework.python
perSystem.devenv.shells.<name>.languages.ruby.enable
perSystem.devenv.shells.<name>.languages.ruby.package
perSystem.devenv.shells.<name>.languages.ruby.version
perSystem.devenv.shells.<name>.languages.ruby.versionFile
perSystem.devenv.shells.<name>.languages.rust.enable
perSystem.devenv.shells.<name>.languages.rust.packages
perSystem.devenv.shells.<name>.languages.rust.packages.cargo
perSystem.devenv.shells.<name>.languages.rust.packages.clippy
perSystem.devenv.shells.<name>.languages.rust.packages.rust-analyzer
perSystem.devenv.shells.<name>.languages.rust.packages.rust-src
perSystem.devenv.shells.<name>.languages.rust.packages.rustc
 perSystem.devenv.shells.<name>.languages.rust.packages.rustfmt 
perSystem.devenv.shells.<name>.languages.rust.version
perSystem.devenv.shells.<name>.languages.scala.enable
perSystem.devenv.shells.<name>.languages.scala.package
perSystem.devenv.shells.<name>.languages.swift.enable
perSystem.devenv.shells.<name>.languages.swift.package
perSystem.devenv.shells.<name>.languages.terraform.enable
perSystem.devenv.shells.<name>.languages.terraform.package
perSystem.devenv.shells.<name>.languages.texlive.enable
perSystem.devenv.shells.<name>.languages.texlive.packages
perSystem.devenv.shells.<name>.languages.texlive.base
perSystem.devenv.shells.<name>.languages.typescript.enable

```
perSystem.devenv.shells.<name>.languages.unison.enable
perSystem.devenv.shells.<name>.languages.unison.package
perSystem.devenv.shells.<name>.languages.v.enable
perSystem.devenv.shells.<name>.languages.v.package
perSystem.devenv.shells.<name>.languages.zig.enable
perSystem.devenv.shells.<name>.languages.zig.package
perSystem.devenv.shells.<name>.name
perSystem.devenv.shells.<name>.pre-commit
perSystem.devenv.shells.<name>.process.after
perSystem.devenv.shells.<name>.process.before
perSystem.devenv.shells.<name>.process.implementation
perSystem.devenv.shells.<name>.process.process-compose
< perSystem.devenv.shells.<name>.processes >
perSystem.devenv.shells.<name>.processes.<name>.exec
perSystem.devenv.shells.<name>.processes.<name>.process-compose
perSystem.devenv.shells.<name>.scripts
perSystem.devenv.shells.<name>.scripts.<name>.exec
perSystem.devenv.shells.<name>.services.adminer.enable
perSystem.devenv.shells.<name>.services.adminer.package
perSystem.devenv.shells.<name>.services.adminer.listen
perSystem.devenv.shells.<name>.services.blackfire.enable
perSystem.devenv.shells.<name>.services.blackfire.package
perSystem.devenv.shells.<name>.services.blackfire.client-id
perSystem.devenv.shells.<name>.services.blackfire.client-token
```

perSystem.devenv.shells.<name>.services.blackfire.server-id
perSystem.devenv.shells.<name>.services.blackfire.server-token
perSystem.devenv.shells.<name>.services.blackfire.socket
perSystem.devenv.shells.<name>.services.caddy.enable
perSystem.devenv.shells.<name>.services.caddy.package
perSystem.devenv.shells.<name>.services.caddy.adapter
perSystem.devenv.shells.<name>.services.caddy.ca
perSystem.devenv.shells.<name>.services.caddy.config
perSystem.devenv.shells.<name>.services.caddy.dataDir
perSystem.devenv.shells.<name>.services.caddy.email
perSystem.devenv.shells.<name>.services.caddy.resume
perSystem.devenv.shells.<name>.services.caddy.virtualHosts

perSystem.devenv.shells.<name>.services.caddy.virtualHosts.<name>.serverAliases
perSystem.devenv.shells.<name>.services.cassandra.enable
perSystem.devenv.shells.<name>.services.cassandra.package
perSystem.devenv.shells.<name>.services.cassandra.allowClients
perSystem.devenv.shells.<name>.services.cassandra.clusterName
perSystem.devenv.shells.<name>.services.cassandra.extraConfig
perSystem.devenv.shells.<name>.services.cassandra.jvmOpts
perSystem.devenv.shells.<name>.services.cassandra.listenAddress
perSystem.devenv.shells.<name>.services.cassandra.seedAddresses
perSystem.devenv.shells.<name>.services.couchdb.enable
perSystem.devenv.shells.<name>.services.couchdb.package

perSystem.devenv.shells.<name>.services.couchdb.settings
perSystem.devenv.shells.<name>.services.couchdb.settings.ckhttpd.bindAddress
perSystem.devenv.shells.<name>.services.couchdb.settings.ckhttpd.logFile
perSystem.devenv.shells.<name>.services.couchdb.settings.ckhttpd.port
perSystem.devenv.shells.<name>.services.couchdb.settings.couchdb.database_dir
perSystem.devenv.shells.<name>.services.couchdb.settings.couchdb.single_node
perSystem.devenv.shells.<name>.services.couchdb.settings.couchdb.uriFile
perSystem.devenv.shells.<name>.services.couchdb.settings.couchdb.viewIndexDir
perSystem.devenv.shells.<name>.services.elasticsearch.enable
perSystem.devenv.shells.<name>.services.elasticsearch.package
perSystem.devenv.shells.<name>.services.elasticsearch.cluster_name
perSystem.devenv.shells.<name>.services.elasticsearch.extraCmdLineOptions

perSystem.devenv.shells.<name>.services.elasticsearch.extraConf
perSystem.devenv.shells.<name>.services.elasticsearch.extraJavaOptions
perSystem.devenv.shells.<name>.services.elasticsearch.listenAddress
perSystem.devenv.shells.<name>.services.elasticsearch.logging
perSystem.devenv.shells.<name>.services.elasticsearch.plugins
perSystem.devenv.shells.<name>.services.elasticsearch.port
perSystem.devenv.shells.<name>.services.elasticsearch.single_node
perSystem.devenv.shells.<name>.services.elasticsearch.tcp_port
perSystem.devenv.shells.<name>.services.mailhog.enable
perSystem.devenv.shells.<name>.services.mailhog.package
perSystem.devenv.shells.<name>.services.mailhog.additionalArgs
perSystem.devenv.shells.<name>.services.mailhog.apiListenAddress

perSystem.devenv.shells.<name>.services.mailhog.smtpListenAddress

perSystem.devenv.shells.<name>.services.mailhog.uiListenAddress

perSystem.devenv.shells.<name>.services.memcached.enable

perSystem.devenv.shells.<name>.services.memcached.package

perSystem.devenv.shells.<name>.services.memcached.bind

perSystem.devenv.shells.<name>.services.memcached.port

perSystem.devenv.shells.<name>.services.memcached.startArgs

perSystem.devenv.shells.<name>.services.minio.enable

perSystem.devenv.shells.<name>.services.minio.package

perSystem.devenv.shells.<name>.services.minio.accessKey

perSystem.devenv.shells.<name>.services.minio.browser

perSystem.devenv.shells.<name>.services.minio.buckets

< perSystem.devenv.shells.<name>.services.minio.consoleAddress >

perSystem.devenv.shells.<name>.services.minio.listenAddress

perSystem.devenv.shells.<name>.services.minio.region

perSystem.devenv.shells.<name>.services.minio.secretKey

perSystem.devenv.shells.<name>.services.mongodb.enable

perSystem.devenv.shells.<name>.services.mongodb.package

perSystem.devenv.shells.<name>.services.mongodb.additionalArgs

perSystem.devenv.shells.<name>.services.mysql.enable

perSystem.devenv.shells.<name>.services.mysql.package

perSystem.devenv.shells.<name>.services.mysql.ensureUsers

perSystem.devenv.shells.<name>.services.mysql.ensureUsers.*.ensurePermissions

perSystem.devenv.shells.<name>.services.mysql.ensureUsers.*.name

```
perSystem.devenv.shells.<name>.services.mysql.ensureUsers.*.password
perSystem.devenv.shells.<name>.services.mysql.importTimeZones
perSystem.devenv.shells.<name>.services.mysql.initialDatabases
perSystem.devenv.shells.<name>.services.mysql.initialDatabases.*.name
perSystem.devenv.shells.<name>.services.mysql.initialDatabases.*.schema
perSystem.devenv.shells.<name>.services.mysql.settings
perSystem.devenv.shells.<name>.services.postgres.enable
perSystem.devenv.shells.<name>.services.postgres.package
perSystem.devenv.shells.<name>.services.postgres.createDatabase
perSystem.devenv.shells.<name>.services.postgres.initdbArgs
perSystem.devenv.shells.<name>.services.postgres.initialDatabases
perSystem.devenv.shells.<name>.services.postgres.initialDatabases.*.name
< perSystem.devenv.shells.<name>.services.postgres.initialDatabases.*.schema >
perSystem.devenv.shells.<name>.services.postgres.initialScript
perSystem.devenv.shells.<name>.services.postgres.listen_addresses
perSystem.devenv.shells.<name>.services.postgres.port
perSystem.devenv.shells.<name>.services.postgres.settings
perSystem.devenv.shells.<name>.services.rabbitmq.enable
perSystem.devenv.shells.<name>.services.rabbitmq.package
perSystem.devenv.shells.<name>.services.rabbitmq.configItems
perSystem.devenv.shells.<name>.services.rabbitmq.cookie
perSystem.devenv.shells.<name>.services.rabbitmq.listenAddress
perSystem.devenv.shells.<name>.services.rabbitmq.managementPlugin.enable
perSystem.devenv.shells.<name>.services.rabbitmq.managementPlugin.port
```

```
perSystem.devenv.shells.<name>.services.rabbitmq.pluginDirs  
perSystem.devenv.shells.<name>.services.rabbitmq.plugins  
perSystem.devenv.shells.<name>.services.rabbitmq.port  
perSystem.devenv.shells.<name>.services.redis.enable  
perSystem.devenv.shells.<name>.services.redis.package  
perSystem.devenv.shells.<name>.services.redis.bind  
perSystem.devenv.shells.<name>.services.redis.extraConfig  
perSystem.devenv.shells.<name>.services.redis.port  
perSystem.devenv.shells.<name>.services.wiremock.enable  
perSystem.devenv.shells.<name>.services.wiremock.package  
perSystem.devenv.shells.<name>.services.wiremock.disableBanner  
perSystem.devenv.shells.<name>.services.wiremock.mappings  
 perSystem.devenv.shells.<name>.services.wiremock.port   
perSystem.devenv.shells.<name>.services.wiremock.verbose  
perSystem.devenv.shells.<name>.starship.enable  
perSystem.devenv.shells.<name>.starship.package  
perSystem.devenv.shells.<name>.starship.config.enable  
perSystem.devenv.shells.<name>.starship.config.path
```

perSystem.devenv.shells

The devenv.sh settings, per shell.

Each definition `devenv.shells.<name>` results in a value for `devShells.<name>`.

Define `devenv.shells.default` for the default `nix develop` invocation - without an argument.

Type: lazy attribute set of (submodule)

Default: { }

Example:

```
{  
    # create devShells.default  
    default = {  
        # devenv settings, e.g.  
        languages.elm.enable = true;  
    };  
}
```

Declared by:

[devenv/flake-module.nix](#)

perSystem.devenv.shells.<name>.packages

A list of packages to expose inside the developer environment. Search available packages using `devenv search NAME`.

Type: list of package

Default: []

Declared by:

[devenv/src/modules/top-level.nix](#)

perSystem.devenv.shells.<name>.certificates

List of domains to generate certificates for.

Type: list of string

Default: []

Example:

```
[  
  "example.com"  
  "*.example.com"  
]
```

Declared by:

[devenv/src/modules/integrations/mkcert.nix](#)

perSystem.devenv.shells. <name>.container.isBuilding

Set to true when the environment is building a container.

Type: boolean

Default: false

Declared by:

[!\[\]\(d5b865035bfbe790b5ce04127faffad2_img.jpg\) devenv/src/modules/containers.nix](#) [!\[\]\(65c3fe619f2a9ede9a51dd9d35a895e0_img.jpg\)](#)

perSystem.devenv.shells.<name>.containers

Container specifications that can be built, copied and ran using `devenv container`.

Type: attribute set of (submodule)

Default: { }

Declared by:

[devenv/src/modules/containers.nix](#)

perSystem.devenv.shells.<name>.containers. <name>.copyToRoot

Add a path to the container. Defaults to the whole git repo.

Type: null or path

Default: "self"

Declared by:

[devenv/src/modules/containers.nix](#)

perSystem.devenv.shells.<name>.containers.<name>.defaultCopyArgs

Default arguments to pass to `skopeo copy`. You can override them by passing arguments to the script.

Type: list of string

Default: []

Declared by:



[devenv/src/modules/containers.nix](#)

perSystem.devenv.shells.<name>.containers.<name>.entrypoint

Entry point of the container.

Type: list of anything

Default: [entrypoint]

Declared by:

[devenv/src/modules/containers.nix](#)

perSystem.devenv.shells.<name>.containers.

<name>.isBuilding

Set to true when the environment is building this container.

Type: boolean

Default: false

Declared by:

[devenv/src/modules/containers.nix](#)

perSystem.devenv.shells.<name>.containers.<name>.name

Name of the container.

Type: null or string

Default: "top-level name or containers.mycontainer.name"

Declared by:

[devenv/src/modules/containers.nix](#)

perSystem.devenv.shells.<name>.containers.<name>.registry

Registry to push the container to.

Type: null or string

Default: "docker://"

Declared by:

[devenv/src/modules/containers.nix](#)

perSystem.devenv.shells.<name>.containers.<name>.startupCommand

Command to run in the container.

Type: null or string or package

Default: null

Declared by:

`devenv/src/modules/containers.nix`

perSystem.devenv.shells.<name>.containers.<name>.version

Version/tag of the container.

Type: null or string

Default: "latest"

Declared by:

`devenv/src/modules/containers.nix`

perSystem.devenv.shells.<name>.devcontainer.enable

Whether to enable generation .devcontainer.json for devenv integration.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/integrations/devcontainer.nix](#)

perSystem.devenv.shells. <name>.devcontainer.settings

Devcontainer settings.

Type: JSON value

Default: {}

Declared by:

[devenv/src/modules/integrations/devcontainer.nix](#)

perSystem.devenv.shells. <name>.devcontainer.settings.customizations.v scode.extensions

List of preinstalled VSCode extensions.

Type: list of string

Default:

```
[  
    "mkhl.direnv"  
]
```

Declared by:

[devenv/src/modules/integrations/devcontainer.nix](#)

perSystem.devenv.shells. <name>.devcontainer.settings.image

The name of an image in a container registry.

Type: string

Default: "ghcr.io/cachix/devenv:latest"

Declared by:

[devenv/src/modules/integrations/devcontainer.nix](#)

perSystem.devenv.shells.

<name>.devcontainer.settings.overrideCommand

Override the default command.

Type: anything

Default: false

Declared by:

 [devenv/src/modules/integrations/devcontainer.nix](#) 

perSystem.devenv.shells.

<name>.devcontainer.settings.updateContentCommand

Command to run after container creation.

Type: anything

Default: "devenv ci"

Declared by:

[devenv/src/modules/integrations/devcontainer.nix](#)

perSystem.devenv.shells. <name>.devenv.flakesIntegration

Tells if devenv is being imported by a flake.nix file

Type: boolean

Default: true when devenv is invoked via the flake integration; false otherwise.

Declared by:

[devenv/src/modules/update-check.nix](#)

perSystem.devenv.shells. <name>.devenv.latestVersion

The latest version of devenv.

Type: string



Default: "0.6.2"

Declared by:

[devenv/src/modules/update-check.nix](#)

perSystem.devenv.shells. <name>.devenv.warnOnNewVersion

Whether to warn when a new version of devenv is available.

Type: boolean

Default: true

Declared by:

[devenv/src/modules/update-check.nix](#)

perSystem.devenv.shells.<name>.difftastic.enable

Integrate difftastic into git: <https://difftastic.wilfred.me.uk/>.

Type: boolean

Default: false

Declared by:

<devenv/src/modules/integrations/diftastic.nix>

perSystem.devenv.shells.<name>.enterShell

Bash code to execute when entering the shell.

Type: strings concatenated with "\n"

Default: ""

Declared by:

<devenv/src/modules/top-level.nix>



perSystem.devenv.shells.<name>.env

Environment variables to be exposed inside the developer environment.

Type: lazy attribute set of anything

Default: { }

Declared by:

<devenv/src/modules/top-level.nix>

perSystem.devenv.shells.<name>.hosts

List of hosts entries.

Type: attribute set of string

Default: { }

Example:

```
{  
    "example.com" = "127.0.0.1";  
}
```

Declared by:

<devenv/src/modules/integrations/hostctl.nix>

perSystem.devenv.shells.<name>.hostsProfileName

Profile name to use.



Type: string

Default: "devenv-e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"

Declared by:

<devenv/src/modules/integrations/hostctl.nix>

perSystem.devenv.shells.<name>.infoSections

Information about the environment

Type: attribute set of list of string

Default: { }

Declared by:

<devenv/src/modules/info.nix>

perSystem.devenv.shells. <name>.languages.ansible.enable

Whether to enable tools for Ansible development.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/languages/ansible.nix](#)

perSystem.devenv.shells. <name>.languages.ansible.package

The Ansible package to use.



Type: package

Default: pkgs.ansible

Declared by:

[devenv/src/modules/languages/ansible.nix](#)

perSystem.devenv.shells. <name>.languages.c.enable

Whether to enable tools for C development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/c.nix`

`perSystem.devenv.shells.
<name>.languages.clojure.enable`

Whether to enable tools for Clojure development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/clojure.nix`

`< >`

`perSystem.devenv.shells.
<name>.languages.cplusplus.enable`

Whether to enable tools for C++ development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/cplusplus.nix`

`perSystem.devenv.shells.
<name>.languages.crystal.enable`

Whether to enable tools for Crystal development...

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/crystal.nix`

`perSystem.devenv.shells.` `<name>.languages.cue.enable`

Whether to enable tools for Cue development.

Type: boolean

Default: false

Example: true



Declared by:

`devenv/src/modules/languages/cue.nix`

`perSystem.devenv.shells.` `<name>.languages.cue.package`

The CUE package to use.

Type: package

Default: pkgs.cue

Declared by:

`devenv/src/modules/languages/cue.nix`

perSystem.devenv.shells. <name>.languages.dart.enable

Whether to enable tools for Dart development.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/languages/dart.nix](#)

perSystem.devenv.shells. <name>.languages.dart.package

The Dart package to use.



Type: package

Default: pkgs.dart

Declared by:

[devenv/src/modules/languages/dart.nix](#)

perSystem.devenv.shells. <name>.languages.deno.enable

Whether to enable tools for Deno development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/deno.nix`

`perSystem.devenv.shells.<name>.languages.dotnet.enable`

Whether to enable tools for .NET development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/dotnet.nix`

<

>

`perSystem.devenv.shells.<name>.languages.dotnet.package`

The .NET SDK package to use.

Type: package

Default: pkgs.dotnet-sdk

Declared by:

`devenv/src/modules/languages/dotnet.nix`

`perSystem.devenv.shells.<name>.languages.elixir.enable`

Whether to enable tools for Elixir development.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/languages/elixir.nix](https://github.com/nix-community/nix-flake-devenv/blob/main/devenv/src/modules/languages/elixir.nix)

perSystem.devenv.shells. <name>.languages.elixir.package

Which package of Elixir to use.

Type: package

Default: pkgs.elixir

Declared by:

 [devenv/src/modules/languages/elixir.nix](https://github.com/nix-community/nix-flake-devenv/blob/main/devenv/src/modules/languages/elixir.nix) 

perSystem.devenv.shells. <name>.languages.elm.enable

Whether to enable tools for Elm development.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/languages/elm.nix](https://github.com/nix-community/nix-flake-devenv/blob/main/devenv/src/modules/languages/elm.nix)

perSystem.devenv.shells. <name>.languages.erlang.enable

Whether to enable tools for Erlang development.

Type: boolean

Default: false

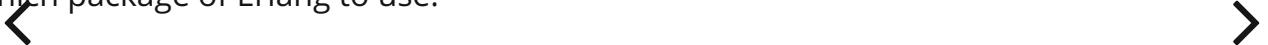
Example: true

Declared by:

[devenv/src/modules/languages/erlang.nix](#)

perSystem.devenv.shells. <name>.languages.erlang.package

Which package of Erlang to use.



Type: package

Default: pkgs.erlang

Declared by:

[devenv/src/modules/languages/erlang.nix](#)

perSystem.devenv.shells. <name>.languages.gawk.enable

Whether to enable tools for GNU Awk development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/gawk.nix`

`perSystem.devenv.shells.<name>.languages.go.enable`

Whether to enable tools for Go development.

Type: boolean

Default: `false`

Example: `true`

Declared by:

`devenv/src/modules/languages/go.nix`

<

>

`perSystem.devenv.shells.<name>.languages.go.package`

The Go package to use.

Type: package

Default: `pkgs.go`

Declared by:

`devenv/src/modules/languages/go.nix`

`perSystem.devenv.shells.<name>.languages.haskell.enable`

Whether to enable tools for Haskell development.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/languages/haskell.nix](https://github.com/nix-community/devenv/blob/main/src/modules/languages/haskell.nix)

perSystem.devenv.shells. <name>.languages.haskell.package

Haskell compiler to use.

Type: package

Default: "pkgs.ghc"

Declared by:

 [</>](https://github.com/nix-community/devenv/blob/main/src/modules/languages/haskell.nix)
devenv/src/modules/languages/haskell.nix

perSystem.devenv.shells. <name>.languages.haskell.languageServer

Haskell language server to use.

Type: null or package

Default: "pkgs.haskell-language-server"

Declared by:

[devenv/src/modules/languages/haskell.nix](https://github.com/nix-community/devenv/blob/main/src/modules/languages/haskell.nix)

perSystem.devenv.shells.

<name>.languages.java.enable

Whether to enable tools for Java development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/java.nix`

perSystem.devenv.shells.

<name>.languages.java.gradle.enable

Whether to enable gradle.

Type: boolean



Default: false

Example: true

Declared by:

`devenv/src/modules/languages/java.nix`

perSystem.devenv.shells.

<name>.languages.java.gradle.package

The Gradle package to use. The Gradle package by default inherits the JDK from `languages.java.jdk.package`.

Type: package

Default: pkgs.gradle.override { jdk = cfg.jdk.package; }

Declared by:

`devenv/src/modules/languages/java.nix`

`perSystem.devenv.shells.<name>.languages.java.jdk.package`

The JDK package to use. This will also become available as `JAVA_HOME`.

Type: package

Default: `pkgs.jdk`

Example: `pkgs.jdk8`

Declared by:

`devenv/src/modules/languages/java.nix`

`perSystem.devenv.shells.<name>.languages.java.maven.enable`

Whether to enable maven.

Type: boolean

Default: `false`

Example: `true`

Declared by:

`devenv/src/modules/languages/java.nix`

`perSystem.devenv.shells.<name>.languages.java.maven.package`

The Maven package to use. The Maven package by default inherits the JDK from `languages.java.jdk.package`.

Type: package

Default: "pkgs.maven.override { jdk = cfg.jdk.package; }"

Declared by:

[devenv/src/modules/languages/java.nix](#)

perSystem.devenv.shells. <name>.languages.javascript.enable

Whether to enable tools for JavaScript development.

Type: boolean

Default: false

Example: true

Declared by:

 [devenv/src/modules/languages/javascript.nix](#) 

perSystem.devenv.shells. <name>.languages.javascript.package

The Node package to use.

Type: package

Default: pkgs.nodejs

Declared by:

[devenv/src/modules/languages/javascript.nix](#)

perSystem.devenv.shells.

<name>.languages.julia.enable

Whether to enable tools for Julia development.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/languages/julia.nix](#)

perSystem.devenv.shells.

<name>.languages.julia.package

The Julia package to use.

Type: package



Default: pkgs.julia-bin

Declared by:

[devenv/src/modules/languages/julia.nix](#)

perSystem.devenv.shells.

<name>.languages.kotlin.enable

Whether to enable tools for Kotlin development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/kotlin.nix`

`perSystem.devenv.shells. <name>.languages.lua.enable`

Whether to enable tools for Lua development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/lua.nix`

`perSystem.devenv.shells. <name>.languages.lua.package`

The Lua package to use.

Type: package

Default: pkgs.lua

Declared by:

`devenv/src/modules/languages/lua.nix`

`perSystem.devenv.shells. <name>.languages.nim.enable`

Whether to enable tools for Nim development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/nim.nix`

perSystem.devenv.shells. <name>.languages.nim.package

The Nim package to use.

Type: package

Default: pkgs.nim

Declared by:

`devenv/src/modules/languages/nim.nix`



perSystem.devenv.shells. <name>.languages.nix.enable

Whether to enable tools for Nix development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/nix.nix`

perSystem.devenv.shells.

<name>.languages.ocaml.enable

Whether to enable tools for OCaml development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/ocaml.nix`

perSystem.devenv.shells.

<name>.languages.ocaml.packages

The package set of OCaml to use

Type: attribute set



Default: pkgs.ocaml-ng.ocamlPackages_4_12

Declared by:

`devenv/src/modules/languages/ocaml.nix`

perSystem.devenv.shells.

<name>.languages.perl.enable

Whether to enable tools for Perl development.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/languages/perl.nix](#)

perSystem.devenv.shells. <name>.languages.php.enable

Whether to enable tools for PHP development.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/languages/php.nix](#)

perSystem.devenv.shells. <name>.languages.php.package

Allows you to override the default used package to adjust the settings or add more extensions. You can find the extensions using `devenv search 'php extensions'`

Type: package

Default: pkgs.php

Example:

```
pkgs.php.buildEnv {  
    extensions = { all, enabled }: with all; enabled ++ [ xdebug ];  
    extraConfig = ''  
        memory_limit=1G  
    '';  
};
```

Declared by:

[devenv/src/modules/languages/php.nix](#)

perSystem.devenv.shells. <name>.languages.php.packages

Attribute set of packages including composer

Type: submodule

Default: pkgs

Declared by:

<devenv/src/modules/languages/php.nix>

perSystem.devenv.shells. <name>.languages.php.packages.composer

composer package

Type: null or package



Default: pkgs.phpPackages.composer

Declared by:

<devenv/src/modules/languages/php.nix>

perSystem.devenv.shells. <name>.languages.php.extensions

PHP extensions to enable.

Type: list of string

Default: []

Declared by:

<devenv/src/modules/languages/php.nix>

perSystem.devenv.shells. <name>.languages.php.fpm.extraConfig

Extra configuration that should be put in the global section of the PHP-FPM configuration file. Do not specify the options `error_log` or `daemonize` here, since they are generated by NixOS.

Type: null or strings concatenated with “\n”

Default: null

Declared by:

[devenv/src/modules/languages/php.nix](#)

perSystem.devenv.shells. <name>.languages.php.fpm.phpOptions

Options appended to the PHP configuration file `php.ini`.

Type: strings concatenated with “\n”

Default: ""

Example:

```
''  
    date.timezone = "CET"  
'''
```

Declared by:

[devenv/src/modules/languages/php.nix](#)

perSystem.devenv.shells. <name>.languages.php.fpm.pools

PHP-FPM pools. If no pools are defined, the PHP-FPM service is disabled.

Type: attribute set of (submodule)

Default: { }

Example:

```
{  
    mypool = {  
        user = "php";  
        group = "php";  
        phpPackage = pkgs.php;  
        settings = {  
            "pm" = "dynamic";  
            "pm.max_children" = 75;  
            "pm.start_servers" = 10;  
            "pm.min_spare_servers" = 5;  
            "pm.max_spare_servers" = 20;  
            "pm.max_requests" = 500;  
        };  
    };  
}
```

Declared by:

[devenv/src/modules/languages/php.nix](#)



perSystem.devenv.shells.
<name>.languages.php.fpm.pools.
<name>.extraConfig

Extra lines that go into the pool configuration. See the documentation on `php-fpm.conf` for details on configuration directives.

Type: null or strings concatenated with “\n”

Default: null

Declared by:

[devenv/src/modules/languages/php.nix](#)

perSystem.devenv.shells.

<name>.languages.php.fpm.pools.<name>.listen

The address on which to accept FastCGI requests.

Type: string

Default: ""

Example: "/path/to/unix/socket"

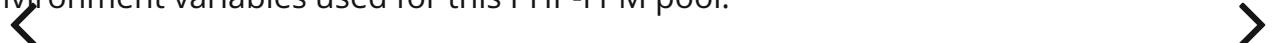
Declared by:

[devenv/src/modules/languages/php.nix](https://flake.parts/src/modules/languages/php.nix)

perSystem.devenv.shells.

<name>.languages.php.fpm.pools.<name>.phpEnv

Environment variables used for this PHP-FPM pool.



Type: attribute set of string

Default: { }

Example:

```
{  
    HOSTNAME = "$HOSTNAME";  
    TMP = "/tmp";  
    TMPDIR = "/tmp";  
    TEMP = "/tmp";  
}
```

Declared by:

[devenv/src/modules/languages/php.nix](https://flake.parts/src/modules/languages/php.nix)

perSystem.devenv.shells.

<name>.languages.php.fpm.pools.

<name>.phpOptions

Options appended to the PHP configuration file `php.ini` used for this PHP-FPM pool.

Type: strings concatenated with “\n”

Declared by:

`devenv/src/modules/languages/php.nix`

`perSystem.devenv.shells.`

`<name>.languages.php.fpm.pools.`

`<name>.phpPackage`

The PHP package to use for running this PHP-FPM pool.

Type: package

Default: `phpfpm.phpPackage`

Declared by:

`devenv/src/modules/languages/php.nix`



`perSystem.devenv.shells.`

`<name>.languages.php.fpm.pools.`

`<name>.settings`

PHP-FPM pool directives. Refer to the “List of pool directives” section of <https://www.php.net/manual/en/install.fpm.configuration.php> the manual for details. Note that settings names must be enclosed in quotes (e.g. `"pm.max_children"` instead of `pm.max_children`).

Type: attribute set of (string or signed integer or boolean)

Default: `{ }`

Example:

```
{  
  "pm" = "dynamic";  
  "pm.max_children" = 75;  
  "pm.start_servers" = 10;  
  "pm.min_spare_servers" = 5;  
  "pm.max_spare_servers" = 20;  
  "pm.max_requests" = 500;  
}
```

Declared by:

[devenv/src/modules/languages/php.nix](#)

perSystem.devenv.shells. <name>.languages.php.fpm.pools.<name>.socket

Path to the Unix socket file on which to accept FastCGI requests.

This option is read-only and managed by NixOS.

Type  string (*read only*) 

Example: "/.devenv/state/php-fpm/<name>.sock"

Declared by:

[devenv/src/modules/languages/php.nix](#)

perSystem.devenv.shells. <name>.languages.php.fpm.settings

PHP-FPM global directives.

Refer to the “List of global php-fpm.conf directives” section of for details.

Note that settings names must be enclosed in quotes (e.g. "pm.max_children" instead of pm.max_children).

You need not specify the options error_log or daemonize here, since they are already set.

Type: attribute set of (string or signed integer or boolean)

Default:

```
{  
    error_log = "/.devenv/state/php-fpm/php-fpm.log";  
}
```

Declared by:

[devenv/src/modules/languages/php.nix](#)

perSystem.devenv.shells. <name>.languages.php.ini

PHP.ini directives. Refer to the “List of php.ini directives” of PHP’s

Type: null or strings concatenated with “\n”

Default: ""

Declared by:

[devenv/src/modules/languages/php.nix](#)

perSystem.devenv.shells. <name>.languages.php.version

The PHP version to use.

Type: string

Default: ""

Declared by:

[devenv/src/modules/languages/php.nix](#)

perSystem.devenv.shells. <name>.languages.purescript.enable

Whether to enable tools for PureScript development.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/languages/purescript.nix](#)

perSystem.devenv.shells. <name>.languages.purescript.package

The PureScript package to use.

Type: package

Default: pkgs.purescript

Declared by:

[devenv/src/modules/languages/purescript.nix](#)

perSystem.devenv.shells. <name>.languages.python.enable

Whether to enable tools for Python development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/python.nix`

`perSystem.devenv.shells.
<name>.languages.python.package`

The Python package to use.

Type: package

Default: `pkgs.python3`

Declared by:

`devenv/src/modules/languages/python.nix`

`perSystem.devenv.shells.
<name>.languages.python.poetry.enable`



Whether to enable poetry.

Type: boolean

Default: `false`

Example: `true`

Declared by:

`devenv/src/modules/languages/python.nix`

`perSystem.devenv.shells.
<name>.languages.python.poetry.package`

The Poetry package to use.

Type: package

Default:

```
pkgs.poetry.override {  
    python3 = config.languages.python.package;  
}
```

Declared by:

[devenv/src/modules/languages/python.nix](#)

perSystem.devenv.shells. <name>.languages.python.venv.enable

Whether to enable Python virtual environment.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/languages/python.nix](#)

perSystem.devenv.shells. <name>.languages.r.enable

Whether to enable tools for R development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/r.nix`

`perSystem.devenv.shells.<name>.languages.r.package`

The R package to use.

Type: package

Default: `pkgs.R`

Declared by:

`devenv/src/modules/languages/r.nix`

`perSystem.devenv.shells.<name>.languages.racket.enable`

Whether to enable tools for Racket development.

Type: boolean

Default: `false`

Example: `true`

Declared by:

`devenv/src/modules/languages/racket.nix`

`perSystem.devenv.shells.<name>.languages.racket.package`

The Racket package to use.

Type: package

Default: `pkgs.racket-minimal`

Declared by:

`devenv/src/modules/languages/racket.nix`

`perSystem.devenv.shells.` `<name>.languages.raku.enable`

Whether to enable tools for Raku development.

Type: boolean

Default: `false`

Example: `true`

Declared by:

`devenv/src/modules/languages/raku.nix`



`perSystem.devenv.shells.` `<name>.languages.robotframework.enable`

Whether to enable tools for Robot Framework development.

Type: boolean

Default: `false`

Example: `true`

Declared by:

`devenv/src/modules/languages/robotframework.nix`

`perSystem.devenv.shells.`

<name>.languages.robotframework.python

The Python package to use.

Type: package

Default: pkgs.python3

Declared by:

<devenv/src/modules/languages/robotframework.nix>

perSystem.devenv.shells.

<name>.languages.ruby.enable

Whether to enable tools for Ruby development.

Type: boolean

Default: false

Example: true

Declared by:

<devenv/src/modules/languages/ruby.nix>



perSystem.devenv.shells.

<name>.languages.ruby.package

The Ruby package to use.

Type: package

Default: pkgs.ruby_3_1

Declared by:

<devenv/src/modules/languages/ruby.nix>

perSystem.devenv.shells. <name>.languages.ruby.version

The Ruby version to use. This automatically sets the `languages.ruby.package` using `nixpkgs-ruby`.

Type: null or string

Default: null

Example: "3.2.1"

Declared by:

`devenv/src/modules/languages/ruby.nix`

perSystem.devenv.shells. <name>.languages.ruby.versionFile

  The `.ruby-version` file path to extract the Ruby version from. This automatically sets the `languages.ruby.package` using `nixpkgs-ruby`. When the `.ruby-version` file exists in the same directory as the devenv configuration, you can use:

```
languages.ruby.versionFile = ./ruby-version;
```

Type: null or path

Default: null

Example:

```
./ruby-version
```

Declared by:

`devenv/src/modules/languages/ruby.nix`

perSystem.devenv.shells.

<name>.languages.rust.enable

Whether to enable tools for Rust development.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/languages/rust.nix](#)

perSystem.devenv.shells.

<name>.languages.rust.packages

Attribute set of packages including rustc and Cargo.

Type: submodule



Default: pkgs

Declared by:

[devenv/src/modules/languages/rust.nix](#)

perSystem.devenv.shells.

<name>.languages.rust.packages.cargo

cargo package

Type: package

Default: pkgs.cargo

Declared by:

[devenv/src/modules/languages/rust.nix](#)

perSystem.devenv.shells.**<name>.languages.rust.packages.clippy**

clippy package

Type: package

Default: pkgs.clippy

Declared by:

<devenv/src/modules/languages/rust.nix>

perSystem.devenv.shells.**<name>.languages.rust.packages.rust-analyzer**

rust-analyzer package

Type: package

Default: pkgs.rust-analyzer

Declared by:

<devenv/src/modules/languages/rust.nix>

perSystem.devenv.shells.**<name>.languages.rust.packages.rust-src**

rust-src package

Type: package or string

Default: pkgs.rustPlatform.rustLibSrc

Declared by:

<devenv/src/modules/languages/rust.nix>

perSystem.devenv.shells. <name>.languages.rust.packages.rustc

rustc package

Type: package

Default: pkgs.rustc

Declared by:

<devenv/src/modules/languages/rust.nix>

perSystem.devenv.shells. <name>.languages.rust.packages.rustfmt

rustfmt package

Type: package

Default: pkgs.rustfmt

Declared by:

<devenv/src/modules/languages/rust.nix>



perSystem.devenv.shells. <name>.languages.rust.version

Set to stable, beta, or latest.

Type: null or string

Default: null

Declared by:

<devenv/src/modules/languages/rust.nix>

perSystem.devenv.shells. <name>.languages.scala.enable

Whether to enable tools for Scala development.

Type: boolean

Default: false

Example: true

Declared by:

<devenv/src/modules/languages/scala.nix>

perSystem.devenv.shells. <name>.languages.scala.package

The Scala package to use.

Type: package

Default: "pkgs.scala_3"

Declared by:

<devenv/src/modules/languages/scala.nix>

perSystem.devenv.shells. <name>.languages.swift.enable

Whether to enable tools for Swift development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/swift.nix`

`perSystem.devenv.shells.<name>.languages.swift.package`

The Swift package to use.

Type: package

Default: "pkgs.swift"

Declared by:

`devenv/src/modules/languages/swift.nix`

`perSystem.devenv.shells.<name>.languages.terraform.enable`



Whether to enable tools for Terraform development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/terraform.nix`

`perSystem.devenv.shells.<name>.languages.terraform.package`

The Terraform package to use.

Type: package

Default: pkgs.terraform

Declared by:

[devenv/src/modules/languages/terraform.nix](#)

perSystem.devenv.shells. <name>.languages.texlive.enable

Whether to enable TeX Live.

Type: boolean

Default: false

Example: true

Declared by:

 [devenv/src/modules/languages/texlive.nix](#) 

perSystem.devenv.shells. <name>.languages.texlive.packages

Packages available to TeX Live

Type: non-empty (list of Concatenated string)

Default:

```
[  
  "collection-basic"  
]
```

Declared by:

[devenv/src/modules/languages/texlive.nix](#)

perSystem.devenv.shells. <name>.languages.texlive.base

TeX Live package set to use

Type: unspecified value

Default: pkgs.texlive

Declared by:

`devenv/src/modules/languages/texlive.nix`

perSystem.devenv.shells. <name>.languages.typescript.enable

Whether to enable tools for TypeScript development.

Type: boolean



Default: false

Example: true

Declared by:

`devenv/src/modules/languages/typescript.nix`

perSystem.devenv.shells. <name>.languages.unison.enable

Whether to enable tools for Unison development.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/languages/unison.nix`

`perSystem.devenv.shells. <name>.languages.unison.package`

Which package of Unison to use

Type: package

Default: `pkgs.unison-ucm`

Declared by:

`devenv/src/modules/languages/unison.nix`

`perSystem.devenv.shells. <name>.languages.v.enable`

Whether to enable tools for V development.

Type: boolean

Default: `false`

Example: `true`

Declared by:

`devenv/src/modules/languages/v.nix`

`perSystem.devenv.shells. <name>.languages.v.package`

The V package to use.

Type: package

Default: pkgs.vlang

Declared by:

[devenv/src/modules/languages/v.nix](#)

perSystem.devenv.shells. <name>.languages.zig.enable

Whether to enable tools for Zig development.

Type: boolean

Default: false

Example: true

Declared by:

 [devenv/src/modules/languages/zig.nix](#) 

perSystem.devenv.shells. <name>.languages.zig.package

Which package of Zig to use.

Type: package

Default: pkgs.zig

Declared by:

[devenv/src/modules/languages/zig.nix](#)

perSystem.devenv.shells.<name>.name

Name of the project.

Type: null or string

Default: null

Declared by:

[devenv/src/modules/top-level.nix](#)

perSystem.devenv.shells.<name>.pre-commit

Integration of <https://github.com/cachix/pre-commit-hooks.nix>

Type: submodule

Default: { }

Declared by:

[devenv/src/modules/integrations/pre-commit.nix](#)



perSystem.devenv.shells.<name>.process.after

Bash code to execute after stopping processes.

Type: strings concatenated with “\n”

Default: ""

Declared by:

[devenv/src/modules/processes.nix](#)

perSystem.devenv.shells.<name>.process.before

Bash code to execute before starting processes.

Type: strings concatenated with “\n”

Default: ""

Declared by:

[devenv/src/modules/processes.nix](#)

perSystem.devenv.shells. <name>.process.implementation

The implementation used when performing `devenv up`.

Type: one of “honcho”, “overmind”, “process-compose”, “hivemind”

Default: "honcho"

Example: "overmind"

Declared by:

 [devenv/src/modules/processes.nix](#) 

perSystem.devenv.shells. <name>.process.process-compose

Top-level process-compose.yaml options when that implementation is used.

Type: attribute set

Default:

```
{  
  port = 9999;  
  tui = true;  
  version = "0.5";  
}
```

Example:

```
{  
    log_level = "fatal";  
    log_location = "/path/to/combined/output/logfile.log";  
    version = "0.5";  
}
```

Declared by:

[devenv/src/modules/processes.nix](#)

perSystem.devenv.shells.<name>.processes

Processes can be started with `devenv up` and run in foreground mode.

Type: attribute set of (submodule)

Default: { }

Declared by:

[devenv/src/modules/processes.nix](#)



perSystem.devenv.shells.<name>.processes.<name>.exec

Bash code to run the process.

Type: string

Declared by:

[devenv/src/modules/processes.nix](#)

perSystem.devenv.shells.<name>.processes.<name>.process-compose

process-compose.yaml specific process attributes.

Example: [https://github.com/F1bonacc1/process-compose/blob/main/process-compose.yaml`](https://github.com/F1bonacc1/process-compose/blob/main/process-compose.yaml)

Only used when using `process.implementation = "process-compose";`

Type: attribute set

Default: `{ }`

Example:

```
{  
    availability = {  
        backoff_seconds = 2;  
        max_restarts = 5;  
        restart = "on_failure";  
    };  
    depends_on = {  
        some-other-process = {  
            condition = "process_completed_successfully";  
        };  
    };  
    environment = [  
        "ENVVAR_FOR_THIS_PROCESS_ONLY=foobar"  
    ];  
}
```



Declared by:

[devenv/src/modules/processes.nix](#)

perSystem.devenv.shells.<name>.scripts

A set of scripts available when the environment is active.

Type: attribute set of (submodule)

Default: `{ }`

Declared by:

[devenv/src/modules/scripts.nix](#)

perSystem.devenv.shells.<name>.scripts.<name>.exec

Bash code to execute when the script is run.

Type: string

Declared by:

<devenv/src/modules/scripts.nix>

perSystem.devenv.shells.<name>.services.adminer.enable

Whether to enable Adminer process.

Type: boolean

Default: false



Example: true

Declared by:

<devenv/src/modules/services/adminer.nix>

perSystem.devenv.shells.<name>.services.adminer.package

Which package of Adminer to use.

Type: package

Default: pkgs.adminer

Declared by:

<devenv/src/modules/services/adminer.nix>

perSystem.devenv.shells. <name>.services.adminer.listen

Listen address for the Adminer.

Type: string

Default: "127.0.0.1:8080"

Declared by:

`devenv/src/modules/services/adminer.nix`

perSystem.devenv.shells. <name>.services.blackfire.enable

Whether to enable Blackfire profiler agent

It automatically installs Blackfire PHP extension. .

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/services/blackfire.nix`

perSystem.devenv.shells. <name>.services.blackfire.package

Which package of blackfire to use

Type: package

Default: pkgs.blackfire

Declared by:

`devenv/src/modules/services/blackfire.nix`

`perSystem.devenv.shells.` `<name>.services.blackfire.client-id`

Sets the client id used to authenticate with Blackfire. You can find your personal client-id at .

Type: string

Default: ""

Declared by:

`devenv/src/modules/services/blackfire.nix`

`<` `perSystem.devenv.shells.` `<name>.services.blackfire.client-token` `>`

Sets the client token used to authenticate with Blackfire. You can find your personal client-token at .

Type: string

Default: ""

Declared by:

`devenv/src/modules/services/blackfire.nix`

`perSystem.devenv.shells.` `<name>.services.blackfire.server-id`

Sets the server id used to authenticate with Blackfire. You can find your personal server-id at .

Type: string

Default: ""

Declared by:

[devenv/src/modules/services/blackfire.nix](#)

perSystem.devenv.shells. <name>.services.blackfire.server-token

Sets the server token used to authenticate with Blackfire. You can find your personal server-token at .

Type: string

Default: ""

Declared by:

[devenv/src/modules/services/blackfire.nix](#)



perSystem.devenv.shells. <name>.services.blackfire.socket

Sets the server socket path

Type: string

Default: "tcp://127.0.0.1:8307"

Declared by:

[devenv/src/modules/services/blackfire.nix](#)

perSystem.devenv.shells. <name>.services.caddy.enable

Whether to enable Caddy web server.

Type: boolean

Default: `false`

Example: `true`

Declared by:

`devenv/src/modules/services/caddy.nix`

`perSystem.devenv.shells.<name>.services.caddy.package`

Caddy package to use.

Type: package

Default: `pkgs.caddy`

Declared by:



`devenv/src/modules/services/caddy.nix`

`perSystem.devenv.shells.<name>.services.caddy.adapter`

Name of the config adapter to use. See for the full list.

Type: string

Default: `"caddyfile"`

Example: `"nginx"`

Declared by:

`devenv/src/modules/services/caddy.nix`

perSystem.devenv.shells. <name>.services.caddy.ca

Certificate authority ACME server. The default (Let's Encrypt production server) should be fine for most people. Set it to null if you don't want to include any authority (or if you want to write a more fine-grained configuration manually).

Type: null or string

Default: "https://acme-v02.api.letsencrypt.org/directory"

Example: "https://acme-staging-v02.api.letsencrypt.org/directory"

Declared by:

[devenv/src/modules/services/caddy.nix](#)

perSystem.devenv.shells. <name>.services.caddy.config >

Verbatim Caddyfile to use. Caddy v2 supports multiple config formats via adapters (see `services.caddy.adapter`).

Type: strings concatenated with "\n"

Default: ""

Example:

```
''  
example.com {  
    encode gzip  
    log  
    root /srv/http  
}  
''
```

Declared by:

[devenv/src/modules/services/caddy.nix](#)

perSystem.devenv.shells. <name>.services.caddy.dataDir

The data directory, for storing certificates. Before 17.09, this would create a .caddy directory. With 17.09 the contents of the .caddy directory are in the specified data directory instead. Caddy v2 replaced CADDYPATH with XDG directories. See .

Type: path

Default: "/.devenv/state/caddy"

Declared by:

[devenv/src/modules/services/caddy.nix](#)

perSystem.devenv.shells. <name>.services.caddy.email

Email address (for Let's Encrypt certificate).  

Type: string

Default: ""

Declared by:

[devenv/src/modules/services/caddy.nix](#)

perSystem.devenv.shells. <name>.services.caddy.resume

Use saved config, if any (and prefer over configuration passed with `caddy.config`).

Type: boolean

Default: false

Declared by:

`devenv/src/modules/services/caddy.nix`

`perSystem.devenv.shells.<name>.services.caddy.virtualHosts`

Declarative vhost config.

Type: attribute set of (submodule)

Default: { }

Example:

```
{  
  "hydra.example.com" = {  
    serverAliases = [ "www.hydra.example.com" ];  
    extraConfig = "";  
    encode gzip  
    log  
    root /srv/http  
  }; >
```

Declared by:

`devenv/src/modules/services/caddy.nix`

`perSystem.devenv.shells.<name>.services.caddy.virtualHosts.<name>.extraConfig`

These lines go into the vhost verbatim.

Type: strings concatenated with “\n”

Default: ""

Declared by:

[devenv/src/modules/services/caddy.nix](#)

perSystem.devenv.shells. <name>.services.caddy.virtualHosts. <name>.serverAliases

Additional names of virtual hosts served by this virtual host configuration.

Type: list of string

Default: []

Example:

```
[  
  "www.example.org"  
  "example.org"  
]
```

Declared by:



[devenv/src/modules/services/caddy.nix](#)

perSystem.devenv.shells. <name>.services.cassandra.enable

Whether to enable Add Cassandra process script...

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/services/cassandra.nix](#)

perSystem.devenv.shells. <name>.services.cassandra.package

Which version of Cassandra to use

Type: package

Default: pkgs.cassandra_4

Example: pkgs.cassandra_4;

Declared by:

`devenv/src/modules/services/cassandra.nix`

perSystem.devenv.shells. <name>.services.cassandra.allowClients

Enables or disables the native transport server (CQL binary protocol)

Type: boolean

Default: true

Declared by:

`devenv/src/modules/services/cassandra.nix`

perSystem.devenv.shells. <name>.services.cassandra.clusterName

The name of the cluster

Type: string

Default: "Test Cluster"

Declared by:

[devenv/src/modules/services/cassandra.nix](#)

perSystem.devenv.shells. <name>.services.cassandra.extraConfig

Extra options to be merged into `cassandra.yaml` as nix attribute set.

Type: attribute set

Default: { }

Example:

```
{  
    commitlog_sync_batch_window_in_ms = 3;  
}
```

Declared by:

[devenv/src/modules/services/cassandra.nix](#)



perSystem.devenv.shells. <name>.services.cassandra.jvmOpts

Options to pass to the JVM through the `JVM_OPTS` environment variable

Type: list of string

Default: []

Declared by:

[devenv/src/modules/services/cassandra.nix](#)

perSystem.devenv.shells. <name>.services.cassandra.listenAddress

Listen address

Type: string

Default: "127.0.0.1"

Example: "127.0.0.1"

Declared by:

[devenv/src/modules/services/cassandra.nix](#)

perSystem.devenv.shells.

<name>.services.cassandra.seedAddresses

The addresses of hosts designated as contact points of the cluster

Type: list of string

Default:

 ["127.0.0.1"] 

Declared by:

[devenv/src/modules/services/cassandra.nix](#)

perSystem.devenv.shells.

<name>.services.couchdb.enable

Whether to enable CouchDB process.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/services/couchdb.nix`

`perSystem.devenv.shells. <name>.services.couchdb.package`

Which version of CouchDB to use

Type: package

Default: `pkgs.couchdb3`

Declared by:

`devenv/src/modules/services/couchdb.nix`

`perSystem.devenv.shells. <name>.services.couchdb.settings`

CouchDB configuration. to know more about all settings, look at: <link xlink:href="https://docs.couchdb.org/en/stable/config/couchdb.html" />

Type: attribute set of attribute set of (INI atom (null, bool, int, float or string))

Default: `{ }`

Example:

```
{  
    couchdb = {  
        database_dir = baseDir;  
        single_node = true;  
        viewIndexDir = baseDir;  
        uriFile = "./devenv/state/couchdb/couchdb.uri";  
    };  
    admins = {  
        "admin_username" = "pass";  
    };  
    chttpd = {  
        bindAddress = "127.0.0.1";  
    };  
}
```

```
    port = 5984;
    logFile = "./devenv/state/couchdb/couchdb.log";
  };
}
```

Declared by:

[devenv/src/modules/services/couchdb.nix](#)

perSystem.devenv.shells.<name>.services.couchdb.settings.ckhttpd.bindAddress

Defines the IP address by which CouchDB will be accessible.

Type: string

Default: "127.0.0.1"

Declared by:

[devenv/src/modules/services/couchdb.nix](#)

perSystem.devenv.shells.<name>.services.couchdb.settings.ckhttpd.logFile

Specifies the location of file for logging output.

Type: path

Default: "./devenv/state/couchdb/couchdb.log"

Declared by:

[devenv/src/modules/services/couchdb.nix](#)

perSystem.devenv.shells. <name>.services.couchdb.settings.ckhttpd.port

Defined the port number to listen.

Type: 16 bit unsigned integer; between 0 and 65535 (both inclusive)

Default: 5984

Declared by:

`devenv/src/modules/services/couchdb.nix`

perSystem.devenv.shells. <name>.services.couchdb.settings.couchdb.data base_dir

Specifies location of CouchDB database files (*.couch named). This location should be
~~writable~~ and readable for the user the CouchDB service runs as (couchdb by default).

Type: path

Default: "/.devenv/state/couchdb"

Declared by:

`devenv/src/modules/services/couchdb.nix`

perSystem.devenv.shells. <name>.services.couchdb.settings.couchdb.sing le_node

When this configuration setting is set to true, automatically create the system databases on startup. Must be set false for a clustered CouchDB installation.

Type: boolean

Default: true

Declared by:

[devenv/src/modules/services/couchdb.nix](#)

perSystem.devenv.shells.<name>.services.couchdb.settings.couchdb.uriFile

This file contains the full URI that can be used to access this instance of CouchDB. It is used to help discover the port CouchDB is running on (if it was set to 0 (e.g. automatically assigned any free one). This file should be writable and readable for the user that runs the CouchDB service (couchdb by default).

Type: path

Default: "/.devenv/state/couchdb/couchdb.uri"

Declared by:

[devenv/src/modules/services/couchdb.nix](#)



perSystem.devenv.shells.<name>.services.couchdb.settings.couchdb.viewIndexDir

Specifies location of CouchDB view index files. This location should be writable and readable for the user that runs the CouchDB service (couchdb by default).

Type: path

Default: "/.devenv/state/couchdb"

Declared by:

[devenv/src/modules/services/couchdb.nix](#)

perSystem.devenv.shells. <name>.services.elasticsearch.enable

Whether to enable elasticsearch.

Type: boolean

Default: false

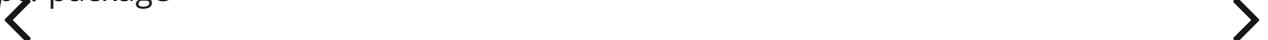
Declared by:

<devenv/src/modules/services/elasticsearch.nix>

perSystem.devenv.shells. <name>.services.elasticsearch.package

Elasticsearch package to use.

Type: package



Default: pkgs.elasticsearch7

Declared by:

<devenv/src/modules/services/elasticsearch.nix>

perSystem.devenv.shells. <name>.services.elasticsearch.cluster_name

Elasticsearch name that identifies your cluster for auto-discovery.

Type: string

Default: "elasticsearch"

Declared by:

<devenv/src/modules/services/elasticsearch.nix>

perSystem.devenv.shells. <name>.services.elasticsearch.extraCommandLineOptions

Extra command line options for the elasticsearch launcher.

Type: list of string

Default: []

Declared by:

[devenv/src/modules/services/elasticsearch.nix](#)

perSystem.devenv.shells. <name>.services.elasticsearch.extraConfig

Extra configuration for elasticsearch.

 
Type: string

Default: ""

Example:

```
''  
  node.name: "elasticsearch"  
  node.master: true  
  node.data: false  
''
```

Declared by:

[devenv/src/modules/services/elasticsearch.nix](#)

perSystem.devenv.shells. <name>.services.elasticsearch.extraJavaOptions

Extra command line options for Java.

Type: list of string

Default: []

Example:

```
[  
  "-Djava.net.preferIPv4Stack=true"  
]
```

Declared by:

[devenv/src/modules/services/elasticsearch.nix](#)

perSystem.devenv.shells.

<name>.services.elasticsearch.listenAddress

Elasticsearch listen address.



Type: string

Default: "127.0.0.1"

Declared by:

[devenv/src/modules/services/elasticsearch.nix](#)

perSystem.devenv.shells.

<name>.services.elasticsearch.logging

Elasticsearch logging configuration.

Type: string

Default:

```
''  
logger.action.name = org.elasticsearch.action
```

```
logger.action.level = info
appender.console.type = Console
appender.console.name = console
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = [%d{ISO8601}] [%-5p] [%-25c{1.}] %marker%m%n
rootLogger.level = info
rootLogger.appenderRef.console.ref = console
''
```

Declared by:

[devenv/src/modules/services/elasticsearch.nix](https://github.com/flake-parts/devenv/blob/main/src/modules/services/elasticsearch.nix)

perSystem.devenv.shells. <name>.services.elasticsearch.plugins

Extra elasticsearch plugins

Type: list of package

Default: []

Example: [pkgs.elasticsearchPlugins.discovery-ec2]

Declared by:

[devenv/src/modules/services/elasticsearch.nix](https://github.com/flake-parts/devenv/blob/main/src/modules/services/elasticsearch.nix)

perSystem.devenv.shells. <name>.services.elasticsearch.port

Elasticsearch port to listen for HTTP traffic.

Type: signed integer

Default: 9200

Declared by:

[devenv/src/modules/services/elasticsearch.nix](https://github.com/flake-parts/devenv/blob/main/src/modules/services/elasticsearch.nix)

perSystem.devenv.shells. <name>.services.elasticsearch.single_node

Start a single-node cluster

Type: boolean

Default: true

Declared by:

<devenv/src/modules/services/elasticsearch.nix>

perSystem.devenv.shells. <name>.services.elasticsearch.tcp_port

Elasticsearch port for the node to node communication.

Type: signed integer

Default: 9300

Declared by:

<devenv/src/modules/services/elasticsearch.nix>

perSystem.devenv.shells. <name>.services.mailhog.enable

Whether to enable mailhog process.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/services/mailhog.nix](#)

perSystem.devenv.shells. <name>.services.mailhog.package

Which package of mailhog to use

Type: package

Default: pkgs.mailhog

Declared by:

[devenv/src/modules/services/mailhog.nix](#)

perSystem.devenv.shells. <name>.services.mailhog.additionalArgs >

Additional arguments passed to `mailhog`.

Type: list of strings concatenated with “\n”

Default: []

Example:

```
[  
  "-invite-jim"  
]
```

Declared by:

[devenv/src/modules/services/mailhog.nix](#)

perSystem.devenv.shells. <name>.services.mailhog.apiListenAddress

Listen address for API.

Type: string

Default: "127.0.0.1:8025"

Declared by:

`devenv/src/modules/services/mailhog.nix`

`perSystem.devenv.shells.`

`<name>.services.mailhog.smtpListenAddress`

Listen address for SMTP.

Type: string

Default: "127.0.0.1:1025"

Declared by:

`devenv/src/modules/services/mailhog.nix`



`perSystem.devenv.shells.`

`<name>.services.mailhog.uiListenAddress`

Listen address for UI.

Type: string

Default: "127.0.0.1:8025"

Declared by:

`devenv/src/modules/services/mailhog.nix`

`perSystem.devenv.shells.`

<name>.services.memcached.enable

Whether to enable memcached process.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/services/memcached.nix](#)

perSystem.devenv.shells.

<name>.services.memcached.package

Which package of memcached to use

Type: package



Default: pkgs.memcached

Declared by:

[devenv/src/modules/services/memcached.nix](#)

perSystem.devenv.shells.

<name>.services.memcached.bind

The IP interface to bind to. null means “all interfaces”.

Type: null or string

Default: "127.0.0.1"

Example: "127.0.0.1"

Declared by:

`devenv/src/modules/services/memcached.nix`

`perSystem.devenv.shells.<name>.services.memcached.port`

The TCP port to accept connections. If port 0 is specified Redis will not listen on a TCP socket.

Type: 16 bit unsigned integer; between 0 and 65535 (both inclusive)

Default: 11211

Declared by:

`devenv/src/modules/services/memcached.nix`

`perSystem.devenv.shells.<name>.services.memcached.startArgs >`

Additional arguments passed to `memcached` during startup.

Type: list of strings concatenated with “\n”

Default: []

Example:

```
[  
  "--memory-limit=100M"  
]
```

Declared by:

`devenv/src/modules/services/memcached.nix`

`perSystem.devenv.shells.`

<name>.services.minio.enable

Whether to enable MinIO Object Storage.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/services/minio.nix`

perSystem.devenv.shells.

<name>.services.minio.package

MinIO package to use.

Type: package



Default: pkgs.minio

Declared by:

`devenv/src/modules/services/minio.nix`

perSystem.devenv.shells.

<name>.services.minio.accessKey

Access key of 5 to 20 characters in length that clients use to access the server. This overrides the access key that is generated by MinIO on first startup and stored inside the `configDir` directory.

Type: string

Default: ""

Declared by:

`devenv/src/modules/services/minio.nix`

`perSystem.devenv.shells.` `<name>.services.minio.browser`

Enable or disable access to web UI.

Type: boolean

Default: true

Declared by:

`devenv/src/modules/services/minio.nix`

`perSystem.devenv.shells.` `<name>.services.minio.buckets`

List of buckets to ensure exist on startup.

Type: list of string

Default: []

Declared by:

`devenv/src/modules/services/minio.nix`

`perSystem.devenv.shells.` `<name>.services.minio.consoleAddress`

IP address and port of the web UI (console).

Type: string

Default: "127.0.0.1:9001"

Declared by:

`devenv/src/modules/services/minio.nix`

`perSystem.devenv.shells.` `<name>.services.minio.listenAddress`

IP address and port of the server.

Type: string

Default: "127.0.0.1:9000"

Declared by:

`devenv/src/modules/services/minio.nix`

`perSystem.devenv.shells.` `<name>.services.minio.region`

The physical location of the server. By default it is set to us-east-1, which is same as AWS S3's and MinIO's default region.

Type: string

Default: "us-east-1"

Declared by:

`devenv/src/modules/services/minio.nix`

`perSystem.devenv.shells.` `<name>.services.minio.secretKey`

Specify the Secret key of 8 to 40 characters in length that clients use to access the server. This overrides the secret key that is generated by MinIO on first startup and stored inside

the `configDir` directory.

Type: string

Default: ""

Declared by:

`devenv/src/modules/services/minio.nix`

perSystem.devenv.shells. <name>.services.mongodb.enable

Whether to enable MongoDB process and expose utilities.

Type: boolean

Default: false

Example: true

Declared by:



`devenv/src/modules/services/mongodb.nix`

perSystem.devenv.shells. <name>.services.mongodb.package

Which MongoDB package to use.

Type: package

Default: pkgs.mongodb

Declared by:

`devenv/src/modules/services/mongodb.nix`

perSystem.devenv.shells.

<name>.services.mongodb.additionalArgs

Additional arguments passed to `mongod`.

Type: list of strings concatenated with “\n”

Default:

```
[  
  "--noauth"  
]
```

Example:

```
[  
  "--port"  
  "27017"  
  "--noauth"  
]
```

Declared by:

 [devenv/src/modules/services/mongodb.nix](https://github.com/nix-community/devenv/blob/main/src/modules/services/mongodb.nix) 

perSystem.devenv.shells.

<name>.services.mysql.enable

Whether to enable MySQL process and expose utilities.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/services/mysql.nix](https://github.com/nix-community/devenv/blob/main/src/modules/services/mysql.nix)

perSystem.devenv.shells. <name>.services.mysql.package

Which package of MySQL to use

Type: package

Default: pkgs.mysql80

Declared by:

`devenv/src/modules/services/mysql.nix`

perSystem.devenv.shells. <name>.services.mysql.ensureUsers

Ensures that the specified users exist and have at least the ensured permissions. The MySQL users will be identified using Unix socket authentication. This authenticates the Unix user with the same name only, and that without the need for a password. This option will never delete existing users or remove permissions, especially not when the value of this option is changed. This means that users created and permissions assigned once through this option or otherwise have to be removed manually.

Type: list of (submodule)

Default: []

Example:

```
[  
  {  
    name = "devenv";  
    ensurePermissions = {  
      "devenv.*" = "ALL PRIVILEGES";  
    };  
  }  
]
```

Declared by:

`devenv/src/modules/services/mysql.nix`

perSystem.devenv.shells. <name>.services.mysql.ensureUsers.*.ensurePermissions

Permissions to ensure for the user, specified as attribute set. The attribute names specify the database and tables to grant the permissions for, separated by a dot. You may use wildcards here. The attribute values specify the permissions to grant. You may specify one or multiple comma-separated SQL privileges here. For more information on how to specify the target and on which privileges exist, see the GRANT syntax. The attributes are used as GRANT \${attrName} ON \${attrValue} .

Type: attribute set of string

Default: { }

Example:

```
{  
    "database.*" = "ALL PRIVILEGES";  
    "*.*" = "SELECT, LOCK TABLES";  
}
```



Declared by:

<devenv/src/modules/services/mysql.nix>

perSystem.devenv.shells. <name>.services.mysql.ensureUsers.*.name

Name of the user to ensure.

Type: string

Declared by:

<devenv/src/modules/services/mysql.nix>

perSystem.devenv.shells.

<name>.services.mysql.ensureUsers.*.password

Password of the user to ensure.

Type: null or string

Default: null

Declared by:

[devenv/src/modules/services/mysql.nix](#)

perSystem.devenv.shells.

<name>.services.mysql.importTimeZones

Whether to import tzdata on the first startup of the mysql server

Type: null or boolean

Default: null

Declared by:

[devenv/src/modules/services/mysql.nix](#)



perSystem.devenv.shells.

<name>.services.mysql.initialDatabases

List of database names and their initial schemas that should be used to create databases on the first startup of MySQL. The schema attribute is optional: If not specified, an empty database is created.

Type: list of (submodule)

Default: []

Example:

[

```
{ name = "foodatabase"; schema = ./foodatabase.sql; }
{ name = "bardatabase"; }
]
```

Declared by:

[devenv/src/modules/services/mysql.nix](#)

perSystem.devenv.shells. <name>.services.mysql.initialDatabases.*.name

The name of the database to create.

Type: string

Declared by:

[devenv/src/modules/services/mysql.nix](#)

<

>

perSystem.devenv.shells. <name>.services.mysql.initialDatabases.*.schema

The initial schema of the database; if null (the default), an empty database is created.

Type: null or path

Default: null

Declared by:

[devenv/src/modules/services/mysql.nix](#)

perSystem.devenv.shells. <name>.services.mysql.settings

MySQL configuration.

Type: attribute set of attribute set of (INI atom (null, bool, int, float or string) or a list of them for duplicate keys)

Default: { }

Example:

```
{  
    mysqlld = {  
        key_buffer_size = "6G";  
        table_cache = 1600;  
        log-error = "/var/log/mysql_err.log";  
        plugin-load-add = [ "server_audit" "ed25519=auth_ed25519" ];  
    };  
    mysqldump = {  
        quick = true;  
        max_allowed_packet = "16M";  
    };  
}
```

Declared by:

[devenv/src/modules/services/mysql.nix](#)



perSystem.devenv.shells. <name>.services.postgres.enable

Whether to enable Add PostgreSQL process. .

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/services/postgres.nix](#)

perSystem.devenv.shells.

<name>.services.postgres.package

Which version of PostgreSQL to use

Type: package

Default: pkgs.postgresql

Example:

```
# see https://github.com/NixOS/nixpkgs/blob/master/pkgs/servers/sql/postgresql/pkgs.postgresql_13.withPackages (p: [ p.pg_cron p.timescaledb p.pg_partman ]);
```

Declared by:

[devenv/src/modules/services/postgres.nix](#)

perSystem.devenv.shells.

<name>.services.postgres.createDatabase



Create a database named like current user on startup. Only applies when initialDatabases is an empty list.

Type: boolean

Default: true

Declared by:

[devenv/src/modules/services/postgres.nix](#)

perSystem.devenv.shells.

<name>.services.postgres.initdbArgs

Additional arguments passed to `initdb` during data dir initialisation.

Type: list of strings concatenated with “\n”

Default:

```
[  
  "--locale=C"  
  "--encoding=UTF8"  
]
```

Example:

```
[  
  "--data-checksums"  
  "--allow-group-access"  
]
```

Declared by:

[devenv/src/modules/services/postgres.nix](#)

perSystem.devenv.shells. <name>.services.postgres.initialDatabases

List of database names and their initial schemas that should be used to create databases on the first startup of Postgres. The schema attribute is optional: If not specified, an empty database is created.

Type: list of (submodule)

Default: []

Example:

```
[  
  {  
    name = "foodatabase";  
    schema = ./foodatabase.sql;  
  }  
  { name = "bardatabase"; }  
]
```

Declared by:

[devenv/src/modules/services/postgres.nix](#)

perSystem.devenv.shells. <name>.services.postgres.initialDatabases.*.name

The name of the database to create.

Type: string

Declared by:

`devenv/src/modules/services/postgres.nix`

perSystem.devenv.shells. <name>.services.postgres.initialDatabases.*.schema

The initial schema of the database; if null (the default), an empty database is created.

Type: null or path



Default: null

Declared by:

`devenv/src/modules/services/postgres.nix`

perSystem.devenv.shells. <name>.services.postgres.initialScript

Initial SQL commands to run during database initialization. This can be multiple SQL expressions separated by a semi-colon.

Type: null or string

Default: null

Example:

```
CREATE USER postgres SUPERUSER;  
CREATE USER bar;
```

Declared by:

[devenv/src/modules/services/postgres.nix](#)

perSystem.devenv.shells. <name>.services.postgres.listen_addresses

Listen address

Type: string

Default: ""

Example: "127.0.0.1"

Declared by:

 [devenv/src/modules/services/postgres.nix](#) 

perSystem.devenv.shells. <name>.services.postgres.port

The TCP port to accept connections.

Type: 16 bit unsigned integer; between 0 and 65535 (both inclusive)

Default: 5432

Declared by:

[devenv/src/modules/services/postgres.nix](#)

perSystem.devenv.shells. <name>.services.postgres.settings

PostgreSQL configuration. Refer to for an overview of `postgresql.conf`.

String values will automatically be enclosed in single quotes. Single quotes will be escaped with two single quotes as described by the upstream documentation linked above.

Type: attribute set of (boolean or floating point number or signed integer or string)

Default: { }

Example:

```
{  
    log_connections = true;  
    log_statement = "all";  
    logging_collector = true  
    log_disconnections = true  
    log_destination = lib.mkForce "syslog";  
}
```

Declared by:

[devenv/src/modules/services/postgres.nix](#)



`perSystem.devenv.shells.<name>.services.rabbitmq.enable`

Whether to enable the RabbitMQ server, an Advanced Message Queuing Protocol (AMQP) broker.

Type: boolean

Default: false

Declared by:

[devenv/src/modules/services/rabbitmq.nix](#)

`perSystem.devenv.shells.<name>.services.rabbitmq.package`

Which rabbitmq package to use.

Type: package

Default: pkgs.rabbitmq-server

Declared by:

[devenv/src/modules/services/rabbitmq.nix](#)

perSystem.devenv.shells.

<name>.services.rabbitmq.configItems

Configuration options in RabbitMQ's new config file format, which is a simple key-value format that can not express nested data structures. This is known as the `rabbitmq.conf` file, although outside NixOS that filename may have Erlang syntax, particularly prior to RabbitMQ 3.7.0. If you do need to express nested data structures, you can use `config` option. Configuration from `config` will be merged into these options by RabbitMQ at runtime to form the final configuration. See For the distinct formats, see



Type: attribute set of string



Default: { }

Example:

```
{  
  "auth_backends.1.authn" = "rabbit_auth_backend_ldap";  
  "auth_backends.1.authz" = "rabbit_auth_backend_internal";  
}
```

Declared by:

[devenv/src/modules/services/rabbitmq.nix](#)

perSystem.devenv.shells.

<name>.services.rabbitmq.cookie

Erlang cookie is a string of arbitrary length which must be the same for several nodes to be

allowed to communicate. Leave empty to generate automatically.

Type: string

Default: ""

Declared by:

[devenv/src/modules/services/rabbitmq.nix](#)

perSystem.devenv.shells.

<name>.services.rabbitmq.listenAddress

IP address on which RabbitMQ will listen for AMQP connections. Set to the empty string to listen on all interfaces. Note that RabbitMQ creates a user named `guest` with password `guest` by default, so you should delete this user if you intend to allow external access. Together with ‘port’ setting it’s mostly an alias for `configItems.listeners.tcp.1` and it’s left for backwards compatibility with previous version of this module.

Type: string



Default: "127.0.0.1"

Example: ""

Declared by:

[devenv/src/modules/services/rabbitmq.nix](#)

perSystem.devenv.shells.

<name>.services.rabbitmq.managementPlugin.enable

Whether to enable the management plugin.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/services/rabbitmq.nix`

`perSystem.devenv.shells.`

`<name>.services.rabbitmq.managementPlugin.port`

On which port to run the management plugin

Type: 16 bit unsigned integer; between 0 and 65535 (both inclusive)

Default: 15672

Declared by:

`devenv/src/modules/services/rabbitmq.nix`

<

>

`perSystem.devenv.shells.`

`<name>.services.rabbitmq.pluginDirs`

The list of directories containing external plugins

Type: list of path

Default: []

Declared by:

`devenv/src/modules/services/rabbitmq.nix`

`perSystem.devenv.shells.`

`<name>.services.rabbitmq.plugins`

The names of plugins to enable

Type: list of string

Default: []

Declared by:

`devenv/src/modules/services/rabbitmq.nix`

`perSystem.devenv.shells. <name>.services.rabbitmq.port`

Port on which RabbitMQ will listen for AMQP connections.

Type: 16 bit unsigned integer; between 0 and 65535 (both inclusive)

Default: 5672

Declared by:

`devenv/src/modules/services/rabbitmq.nix`



`perSystem.devenv.shells. <name>.services.redis.enable`

Whether to enable Redis process and expose utilities.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/services/redis.nix`

perSystem.devenv.shells. <name>.services.redis.package

Which package of Redis to use

Type: package

Default: pkgs.redis

Declared by:

`devenv/src/modules/services/redis.nix`

perSystem.devenv.shells. <name>.services.redis.bind

The IP interface to bind to. `null` means “all interfaces”.

Type: null or string



Default: "127.0.0.1"

Example: "127.0.0.1"

Declared by:

`devenv/src/modules/services/redis.nix`

perSystem.devenv.shells. <name>.services.redis.extraConfig

Additional text to be appended to `redis.conf`.

Type: strings concatenated with “\n”

Default: ""

Declared by:

`devenv/src/modules/services/redis.nix`

`perSystem.devenv.shells. <name>.services.redis.port`

The TCP port to accept connections. If port 0 is specified Redis, will not listen on a TCP socket.

Type: 16 bit unsigned integer; between 0 and 65535 (both inclusive)

Default: 6379

Declared by:

`devenv/src/modules/services/redis.nix`

`perSystem.devenv.shells. <name>.services.wiremock.enable >`

Whether to enable WireMock.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/services/wiremock.nix`

`perSystem.devenv.shells. <name>.services.wiremock.package`

Which package of WireMock to use.

Type: package

Default: `pkgs.wiremock`

Declared by:

`devenv/src/modules/services/wiremock.nix`

`perSystem.devenv.shells.`

`<name>.services.wiremock.disableBanner`

Whether to disable print banner logo.

Type: boolean

Default: `false`

Declared by:

`devenv/src/modules/services/wiremock.nix`

<

>

`perSystem.devenv.shells.`

`<name>.services.wiremock.mappings`

The mappings to mock. See the JSON examples on for more information.

Type: JSON value

Default: `[]`

Example:

```
[  
 {  
   request = {  
     method = "GET";  
     url = "/body";  
   };  
   response = {  
     body = "Literal text to put in the body";  
     headers = {  
       Content-Type = "text/plain";  
     };  
   };  
 }
```

```
        status = 200;
    };
}
{
    request = {
        method = "GET";
        url = "/json";
    };
    response = {
        jsonBody = {
            someField = "someValue";
        };
        status = 200;
    };
}
]
```

Declared by:

[devenv/src/modules/services/wiremock.nix](https://github.com/nix-community/devenv/blob/main/src/modules/services/wiremock.nix)

perSystem.devenv.shells. <name>.services.wiremock.port >

The port number for the HTTP server to listen on.

Type: signed integer

Default: 8080

Declared by:

[devenv/src/modules/services/wiremock.nix](https://github.com/nix-community/devenv/blob/main/src/modules/services/wiremock.nix)

perSystem.devenv.shells. <name>.services.wiremock.verbose

Whether to log verbosely to stdout.

Type: boolean

Default: false

Declared by:

`devenv/src/modules/services/wiremock.nix`

`perSystem.devenv.shells.<name>.starship.enable`

Whether to enable the Starship.rs command prompt.

Type: boolean

Default: false

Example: true

Declared by:

`devenv/src/modules/integrations/starship.nix`

<

>

`perSystem.devenv.shells.<name>.starship.package`

The Starship package to use.

Type: package

Default: pkgs.starship

Declared by:

`devenv/src/modules/integrations/starship.nix`

`perSystem.devenv.shells.<name>.starship.config.enable`

Whether to enable Starship config override.

Type: boolean

Default: false

Example: true

Declared by:

[devenv/src/modules/integrations/starship.nix](#)

perSystem.devenv.shells.<name>.starship.config.path

The Starship configuration file to use.

Type: path

Default: \${config.env.DEVENV_ROOT}/starship.toml

Declared by:

 [devenv/src/modules/integrations/starship.nix](#) 



devshell

Simple per-project developer environments.

Example:

```
perSystem = { config, pkgs, ... }: {
    devshells.default = {
        env = [
            {
                name = "HTTP_PORT";
                value = 8080;
            }
        ];
        commands = [
            {
                help = "print hello";
                name = "hello";
                command = "echo hello";
            }
        ];
        packages = [
            pkgs.cowsay
        ];
    };
};
```

< >

See also the [devshell project page](#)

Installation

To use these options, add to your flake inputs:

```
devshell.url = "github:numtide/devshell";
```

and inside the `mkFlake`:

```
imports = [
    inputs.devshell.flakeModule
];
```

Run `nix flake lock` and you're set.

Options

`perSystem.devshells`

`perSystem.devshells.<name>.commands`

`perSystem.devshells.<name>.commands.*.package`

`perSystem.devshells.<name>.commands.*.category`

`perSystem.devshells.<name>.commands.*.command`

`perSystem.devshells.<name>.commands.*.help`

`perSystem.devshells.<name>.commands.*.name`

`perSystem.devshells.<name>.devshell.packages`

`perSystem.devshells.<name>.devshell.packagesFrom`

`perSystem.devshells.<name>.devshell.interactive.<name>.deps`

`perSystem.devshells.<name>.devshell.interactive.<name>.text`

`perSystem.devshells.<name>.devshell.load_profiles`

`perSystem.devshells.<name>.devshell.meta`

`perSystem.devshells.<name>.devshell.motd`

`perSystem.devshells.<name>.devshell.name`

`perSystem.devshells.<name>.devshell.prj_root_fallback`

`perSystem.devshells.<name>.devshell.prj_root_fallback.eval`

`perSystem.devshells.<name>.devshell.prj_root_fallback.name`

`perSystem.devshells.<name>.devshell.prj_root_fallback.prefix`

`perSystem.devshells.<name>.devshell.prj_root_fallback.unset`

```
perSystem.devshells.<name>.devshell.prj_rootFallback.value
perSystem.devshells.<name>.devshell.startup.<name>.deps
perSystem.devshells.<name>.devshell.startup.<name>.text
perSystem.devshells.<name>.env
perSystem.devshells.<name>.env.*.eval
perSystem.devshells.<name>.env.*.name
perSystem.devshells.<name>.env.*.prefix
perSystem.devshells.<name>.env.*.unset
perSystem.devshells.<name>.env.*.value
perSystem.devshells.<name>.serviceGroups
perSystem.devshells.<name>.serviceGroups.<name>.description
perSystem.devshells.<name>.serviceGroups.<name>.name
< perSystem.devshells.<name>.serviceGroups.<name>.services >
perSystem.devshells.<name>.serviceGroups.<name>.services.<name>.command
perSystem.devshells.<name>.serviceGroups.<name>.services.<name>.name
```

perSystem.devshells

Configure devshells with flake-parts.

Not to be confused with `devShells`, with a capital S. Yes, this is unfortunate.

Each devshell will also configure an equivalent `devShells`.

Used to define devshells. not to be confused with `devShells`

Type: lazy attribute set of (submodule)

Default: {}

Declared by:

[devshell/flake-module.nix](#)

perSystem.devshells.<name>.commands

Add commands to the environment.

Type: list of (submodule)

Default: []

Example:

```
[  
 {  
   help = "print hello";  
   name = "hello";  
   command = "echo hello";  
 }
```



```
{  
 package = "nixpkgs-fmt";  
 category = "formatter";  
 }  
 ]
```

Declared by:

[devshell/modules/commands.nix](#)

perSystem.devshells.<name>.commands.*.package

Used to bring in a specific package. This package will be added to the environment.

Type: null or (package or string convertible to it)

Default: null

Declared by:

[devshell/modules/commands.nix](#)

perSystem.devshells. <name>.commands.*.category

Set a free text category under which this command is grouped and shown in the help menu.

Type: string

Default: "[general commands]"

Declared by:

[devshell/modules/commands.nix](#)

perSystem.devshells. <name>.commands.*.command

If defined, it will add a script with the name of the command, and the content of this value.

By default it generates a bash script, unless a different shebang is provided.

Type: null or string

Default: null

Example:

```
''  
#!/usr/bin/env python  
print("Hello")  
'''
```

Declared by:

[devshell/modules/commands.nix](#)

perSystem.devshells.<name>.commands.*.help

Describes what the command does in one line of text.

Type: null or string

Default: null

Declared by:

[devshell/modules/commands.nix](#)

perSystem.devshells.<name>.commands.*.name

Name of this command. Defaults to attribute name in commands.

Type: null or string

Default: null

Declared by:



[devshell/modules/commands.nix](#)

perSystem.devshells.<name>.devshell.packages

The set of packages to appear in the project environment.

Those packages come from <https://nixos.org/NixOS/nixpkgs> and can be searched by going to <https://search.nixos.org/packages>

Type: list of (package or string convertible to it)

Default: []

Declared by:

[devshell/modules/devshell.nix](#)

perSystem.devshells. <name>.devshell.packagesFrom

Add all the build dependencies from the listed packages to the environment.

Type: list of (package or string convertible to it)

Default: []

Declared by:

[devshell/modules/devshell.nix](#)

perSystem.devshells. <name>.devshell.interactive.<name>.deps

A list of other steps that this one depends on.

Type: list of string



Default: []

Declared by:

[devshell/modules/devshell.nix](#)

perSystem.devshells. <name>.devshell.interactive.<name>.text

Script to run.

Type: string

Declared by:

[devshell/modules/devshell.nix](#)

perSystem.devshells.<name>.devshell.load_profiles

Whether to enable load etc/profiles.d/*.sh in the shell.

Type: boolean

Default: false

Example: true

Declared by:

[devshell/modules/devshell.nix](#)

perSystem.devshells.<name>.devshell.meta

Metadata, such as 'meta.description'. Can be useful as metadata for downstream tooling.

Type ↘: attribute set of anything ↗

Default: {}

Declared by:

[devshell/modules/devshell.nix](#)

perSystem.devshells.<name>.devshell.motd

Message Of The Day.

This is the welcome message that is being printed when the user opens the shell.

You may use any valid ansi color from the 8-bit ansi color table. For example, to use a green color you would use something like {106}. You may also use {bold}, {italic}, {underline}. Use {reset} to turn off all attributes.

Type: string

Default:

```
''  
  {202}⚡ Welcome to devshell{reset}  
  $(type -p menu &>/dev/null && menu)  
''
```

Declared by:

[devshell/modules/devshell.nix](#)

perSystem.devshells.<name>.devshell.name

Name of the shell environment. It usually maps to the project name.

Type: string

Default: "devshell"

Declared by:

[devshell/modules/devshell.nix](#)



perSystem.devshells. <name>.devshell.prj_root_fallback

If IN_NIX_SHELL is nonempty, or DIRENV_IN_ENVRC is set to '1', then PRJ_ROOT is set to the value of PWD.

This option specifies the path to use as the value of PRJ_ROOT in case IN_NIX_SHELL is empty or unset and DIRENV_IN_ENVRC is any value other than '1'.

Set this to null to force PRJ_ROOT to be defined at runtime (except if IN_NIX_SHELL or DIRENV_IN_ENVRC are defined as described above).

Otherwise, you can set this to a string representing the desired default path, or to a submodule of the same type valid in the 'env' options list (except that the 'name' field is ignored).

Type: null or ((submodule) or non-empty string convertible to it)

Default:

```
{  
    eval = "$PWD";  
}
```

Example:

```
{  
    # Use the top-level directory of the working tree  
    eval = "$(git rev-parse --show-toplevel)";  
};
```

Declared by:

[devshell/modules/devshell.nix](#)

perSystem.devshells. <name>.devshell.prj_root_fallback.eval

Like value but not evaluated by Bash. This allows to inject other variable names or even commands using the `$()` notation.

Type: null or string

Default: null

Example: "\$OTHER_VAR"

Declared by:

[devshell/modules/env.nix](#)

perSystem.devshells. <name>.devshell.prj_root_fallback.name

Name of the environment variable

Type: string

Declared by:

`devshell/modules/env.nix`

`perSystem.devshells.`

`<name>.devshell.prj_rootFallback.prefix`

Prepend to PATH-like environment variables.

For example name = "PATH"; prefix = "bin"; will expand the path of ./bin and prepend it to the PATH, separated by ':'.

Type: null or string

Default: null

Example: "bin"

Declared by:

`devshell/modules/env.nix`

<

>

`perSystem.devshells.`

`<name>.devshell.prj_rootFallback.unset`

Whether to enable unsets the variable.

Type: boolean

Default: false

Example: true

Declared by:

`devshell/modules/env.nix`

`perSystem.devshells.`

`<name>.devshell.prj_rootFallback.value`

Shell-escaped value to set

Type: null or string or signed integer or boolean or package

Default: null

Declared by:

[devshell/modules/env.nix](#)

perSystem.devshells.<name>.devshell.startup.<name>.deps

A list of other steps that this one depends on.

Type: list of string

Default: []

Declared by:

 [devshell/modules/devshell.nix](#) 

perSystem.devshells.<name>.devshell.startup.<name>.text

Script to run.

Type: string

Declared by:

[devshell/modules/devshell.nix](#)

perSystem.devshells.<name>.env

Add environment variables to the shell.

Type: list of (submodule)

Default: []

Example:

```
[  
 {  
   name = "HTTP_PORT";  
   value = 8080;  
 }  
 {  
   name = "PATH";  
   prefix = "bin";  
 }  
 {  
   name = "XDG_CACHE_DIR";  
   eval = "$PRJ_ROOT/.cache";  
 }  
 {  
   name = "CARGO_HOME";  
   unset = true;  
 }  
 ]
```

Declared by:



[devshell/modules/env.nix](#)

perSystem.devshells.<name>.env.*.eval

Like value but not evaluated by Bash. This allows to inject other variable names or even commands using the `$()` notation.

Type: null or string

Default: null

Example: "\$OTHER_VAR"

Declared by:

[devshell/modules/env.nix](#)

perSystem.devshells.<name>.env.*.name

Name of the environment variable

Type: string

Declared by:

`devshell/modules/env.nix`

perSystem.devshells.<name>.env.*.prefix

Prepend to PATH-like environment variables.

For example name = "PATH"; prefix = "bin"; will expand the path of ./bin and prepend it to the PATH, separated by ':'.

Type: null or string

Default: null



Example: "bin"

Declared by:

`devshell/modules/env.nix`

perSystem.devshells.<name>.env.*.unset

Whether to enable unsets the variable.

Type: boolean

Default: false

Example: true

Declared by:

`devshell/modules/env.nix`

perSystem.devshells.<name>.env.*.value

Shell-escaped value to set

Type: null or string or signed integer or boolean or package

Default: null

Declared by:

[devshell/modules/env.nix](#)

perSystem.devshells.<name>.serviceGroups

Add services to the environment. Services can be used to group long-running processes.

Type: attribute set of (submodule)

Default: { }

Declared by:



[devshell/modules/services.nix](#)

perSystem.devshells.<name>.serviceGroups.<name>.description

Short description of the service group, shown in generated commands

Type: null or string

Default: null

Declared by:

[devshell/modules/services.nix](#)

perSystem.devshells.<name>.serviceGroups.

<name>.name

Name of the service group. Defaults to attribute name in groups.

Type: null or string

Default: null

Declared by:

[devshell/modules/services.nix](#)

perSystem.devshells.<name>.serviceGroups.<name>.services

Attrset of services that should be run in this group.

Type: attribute set of (submodule)

Default: { }



Declared by:

[devshell/modules/services.nix](#)

perSystem.devshells.<name>.serviceGroups.<name>.services.<name>.command

Command to execute.

Type: string

Declared by:

[devshell/modules/services.nix](#)

perSystem.devshells.<name>.serviceGroups.

<name>.services.<name>.name

Name of this service. Defaults to attribute name in group services.

Type: null or string

Default: null

Declared by:

[devshell/modules/services.nix](#)

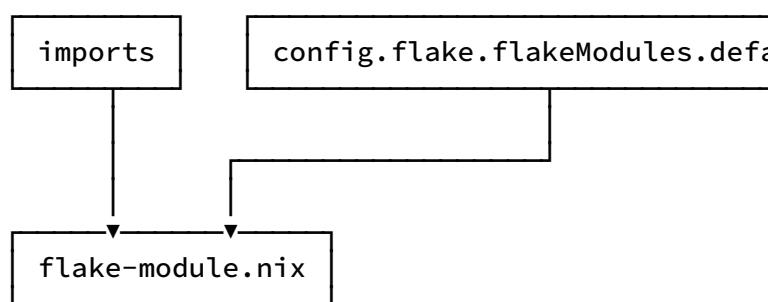


[Introduction](#)[Getting Started](#)[Cheat Sheet](#)[Tutorials](#)[Best Practices for Module Writing](#)[Multi-platform: Working with system](#)[Guides](#)[Explore and debug option values](#)[Define a Module in a Separate File](#)[Define Custom Flake Attribute](#)[Generate Documentation](#)[Dogfood a Reusable Flake Module](#)[Explanation](#)[Overlays](#)[Reference Documentation](#)[Module Arguments](#)[Options](#)[flake-parts \(built in\)](#)[flakeModules](#)[easyOverlay \(warning\)](#)[agenix-shell](#)[devenv](#)[devshell](#)[dream2nix](#)[dream2nix legacy](#)[emanote](#)[ez-configs](#)[flake.parts-website](#)[haskell-flake](#)[hercules-ci-effects](#)[mission-control](#)[nix-cargo-integration](#)

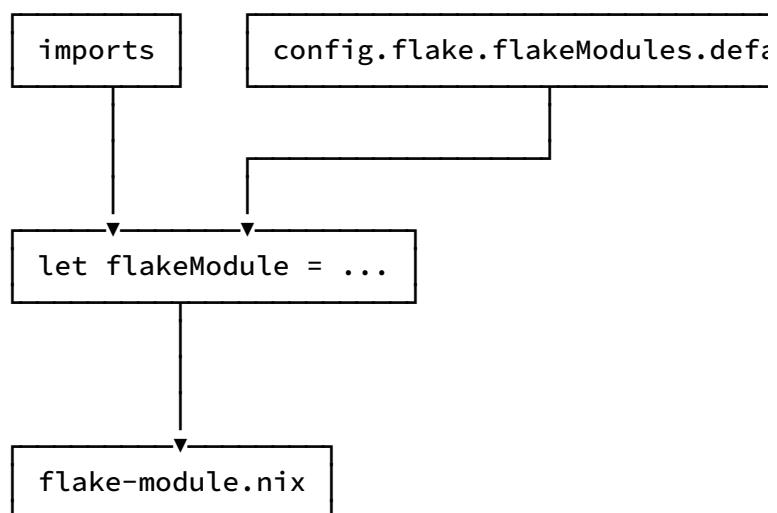
Dogfood a Reusable Mod

to dogfood: test one's own product by using it for your own flakes.

You can distribute reusable module logic through flakes by including `flakeModules`. However, importing from `self` in a `config` import could affect the `self` attribute set. To use your module directly, reference it directly.



If your module does not need anything from the local `config`, implement the references in the diagram above. But if you want to use a package from your local flake, then you need to apply [Define a Module in a Separate File](#). Instead of the arrows joining `imports` and `config`, bind them to a `let` binding.



Example with importApply

[Introduction](#)

[Getting Started](#)

[Cheat Sheet](#)

[Tutorials](#)

[Best Practices for Module Writing](#)

[Multi-platform: Working with system](#)

[Guides](#)

[Explore and debug option values](#)

[Define a Module in a Separate File](#)

[Define Custom Flake Attribute](#)

[Generate Documentation](#)

[Dogfood a Reusable Flake Module](#)

[Explanation](#)

[Overlays](#)

[Reference Documentation](#)

[Module Arguments](#)

[Options](#)

flake-parts (built in)

 flakeModules

 easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

haskell-flake

hercules-ci-effects

mission-control

nix-cargo-integration

Here's an example of how this looks using the `importApply` function to export a flake module that references its own `pkgs.hello` package (which would be available in the module arguments). This is done, in principle, by reexporting a locally defined package in the `outputs` section.

`flake.nix`:

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos";
  };

  outputs = inputs@{ flake-parts, ... }:
    flake-parts.lib.mkFlake { inherit inputs;
      let
        inherit (flake-parts-lib) importApply;
        flakeModules.default = importApply ./flake;
      in
      {
        imports = [
          flakeModules.default
          # inputs.foo.flakeModules.default
        ];
        systems = [ "x86_64-linux" "aarch64-darwin" ];
        perSystem = { pkgs, ... }: {
          packages.default = pkgs.hello;
        };
        flake = {
          inherit flakeModules;
        };
      });
    }
}
```

`flake-module.nix`:

```
# The importApply argument. Use this to refer to the local flake
# as opposed to the flake where this is imported.
localFlake: importApply ./flake;

# Regular module arguments; self, inputs, etc
# where this module was imported.
{ lib, config, self, inputs, ... }:
{
  perSystem = { system, ... }: {
    # A copy of hello that was defined by this
    # flake.
    packages.greeter = localFlake.withSystem {
      config.packages.default
    };
  };
}
```

```
)  
};  
}  
}
```

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

[Dogfood a Reusable Flake Module](#)

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

 flakeModules

 easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

haskell-flake

hercules-ci-effects

mission-control

nix-cargo-integration

The "Factor it out" technique is equally applicable; re|
module.



flac

[Introduction](#)[Getting Started](#)[Cheat Sheet](#)[Tutorials](#)[Best Practices for Module Writing](#)[Multi-platform: Working with system](#)[Guides](#)[Explore and debug option values](#)[Define a Module in a Separate File](#)[Define Custom Flake Attribute](#)[Generate Documentation](#)[Dogfood a Reusable Flake Module](#)[Explanation](#)[Overlays](#)[Reference Documentation](#)[Module Arguments](#)[Options](#)[flake-parts \(built in\)](#)[flakeModules](#)[easyOverlay \(warning\)](#)[agenix-shell](#)[devenv](#)[devshell](#)[dream2nix](#)[dream2nix legacy](#)[emanote](#)[ez-configs](#)[flake.parts-website](#)[haskell-flake](#)[hercules-ci-effects](#)[mission-control](#)[nix-cargo-integration](#)

dream2nix

This page is a placeholder while dream2nix v2 is in the previous API.

Options

dream2nix legacy

`dream2nix` scans your flake files and turns them into packages.

NOTE: a new version of dream2nix, v1, is in the works, and we're figuring out how best to use it.

Installation

To use these options, add to your flake inputs:

```
dream2nix_legacy.url = "github:nix-community/dream2nix/c9c8689f09aa95212e75f3108"
```

and inside the `mkFlake`:

```
< imports = [  
    inputs.dream2nix_legacy.flakeModuleBeta  
]; >
```

Run `nix flake lock` and you're set.

Options

`dream2nix.config`

`dream2nix.config.packagesDir`

`dream2nix.config.modules`

`dream2nix.config.overridesDirs`

`dream2nix.config.projectRoot`

`dream2nix.config.repoName`

```
dream2nix.lib

perSystem.dream2nix.inputs

perSystem.dream2nix.inputs.<name>.packageOverrides

perSystem.dream2nix.inputs.<name>.inject

perSystem.dream2nix.inputs.<name>.pname

perSystem.dream2nix.inputs.<name>.projects

perSystem.dream2nix.inputs.<name>.projects.<name>.builder

perSystem.dream2nix.inputs.<name>.projects.<name>.name

perSystem.dream2nix.inputs.<name>.projects.<name>.relPath

perSystem.dream2nix.inputs.<name>.projects.<name>.subsystem

perSystem.dream2nix.inputs.<name>.projects.<name>.subsystemInfo

perSystem.dream2nix.inputs.<name>.projects.<name>.translator
< perSystem.dream2nix.inputs.<name>.projects.<name>.version >

perSystem.dream2nix.inputs.<name>.settings

perSystem.dream2nix.inputs.<name>.source

perSystem.dream2nix.inputs.<name>.sourceOverrides

perSystem.dream2nix.instance

perSystem.dream2nix.outputs
```

dream2nix.config

The dream2nix config.

Type: submodule

Default:

```
{  
    projectRoot = self;  
}
```

Declared by:

[dream2nix_legacy/src/modules/flake-parts/interface.nix](https://github.com/dream2nix/dream2nix-legacy/blob/main/src/modules/flake-parts/interface.nix)

dream2nix.config.packagesDir

Relative path to the project root to put generated dream-lock files in.

Type: Relative path in a string.

Default: "./dream2nix-packages"

Declared by:

[dream2nix_legacy/src/modules/config/interface.nix](https://github.com/dream2nix/dream2nix-legacy/blob/main/src/modules/config/interface.nix)



dream2nix.config.modules

Extra modules to import in while evaluating the dream2nix framework. This allows you to add new discoverers, translators, builders etc. and lets you override existing ones.

Type: list of path

Default: []

Declared by:

[dream2nix_legacy/src/modules/config/interface.nix](https://github.com/dream2nix/dream2nix-legacy/blob/main/src/modules/config/interface.nix)

dream2nix.config.overridesDirs

Override directories to pull overrides from.

Type: list of path

Default: []

Declared by:

[dream2nix_legacy/src/modules/config/interface.nix](https://github.com/dream2nix/dream2nix/blob/main/src/modules/config/interface.nix)

dream2nix.config.projectRoot

Absolute path to the root of this project.

Type: null or path

Default: null

Declared by:

[dream2nix_legacy/src/modules/config/interface.nix](https://github.com/dream2nix/dream2nix/blob/main/src/modules/config/interface.nix)

dream2nix.config.repoName



Name of the repository this project is in.

Type: null or string

Default: null

Declared by:

[dream2nix_legacy/src/modules/config/interface.nix](https://github.com/dream2nix/dream2nix/blob/main/src/modules/config/interface.nix)

dream2nix.lib

The system-less dream2nix library. This should be the `lib` attribute of the dream2nix flake.

Type: raw value (*read only*)

Declared by:

`dream2nix_legacy/src/modules/flake-parts/interface.nix`

perSystem.dream2nix.inputs

A list of inputs to generate outputs from. Each one takes the same arguments `makeOutputs` takes.

Type: attribute set of (submodule)

Default: { }

Declared by:

`dream2nix_legacy/src/modules/flake-parts/interface.nix`

perSystem.dream2nix.inputs.

<name>.packageOverrides



Overrides to customize build logic for dependencies or top-level packages

Type: lazy attribute set of (attribute set)

Default: { }

Declared by:

`dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix`

perSystem.dream2nix.inputs.<name>.inject

Inject missing dependencies into the dependency tree

Type: lazy attribute set of lazy attribute set of list of list of string

Default: { }

Example:

```
{  
  foo."6.4.1" = [  
    ["bar" "13.2.0"]  
    ["baz" "1.0.0"]  
  ];  
  "@tiptap/extension-code"."2.0.0-beta.26" = [  
    "@tiptap/core" "2.0.0-beta.174"]  
  ];  
};
```

Declared by:

[dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix](https://github.com/dream2nix/dream2nix/blob/main/src/modules/flake-parts/makeOutputsArgs.nix)

perSystem.dream2nix.inputs.<name>.pname

The name of the package to be built with dream2nix

Type: null or string

Default: null

Declared by:

[dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix](https://github.com/dream2nix/dream2nix/blob/main/src/modules/flake-parts/makeOutputsArgs.nix)

perSystem.dream2nix.inputs.<name>.projects

Projects that dream2nix will build

Type: attribute set of (submodule)

Default: { }

Declared by:

[dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix](https://github.com/dream2nix/dream2nix/blob/main/src/modules/flake-parts/makeOutputsArgs.nix)

perSystem.dream2nix.inputs.<name>.projects.

<name>.builder

Name of builder to use

Type: null or string

Default: null

Example: "strict-builder"

Declared by:

`dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix`

perSystem.dream2nix.inputs.<name>.projects.<name>.name

Name of the project

Type: string

Default: "<name>"

Declared by:

`dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix`

perSystem.dream2nix.inputs.<name>.projects.<name>.relPath

Relative path to project tree from source

Type: string

Default: ""

Declared by:

`dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix`

perSystem.dream2nix.inputs.<name>.projects.<name>.subsystem

Name of subsystem to use. Examples: rust, python, nodejs

Type: string

Example: "nodejs"

Declared by:

[dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix](https://github.com/nix-community/dream2nix/blob/main/src/modules/flake-parts/makeOutputsArgs.nix)

perSystem.dream2nix.inputs.<name>.projects.<name>.subsystemInfo

Translator specific arguments

Type: lazy attribute set of anything



Default: { }

Declared by:

[dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix](https://github.com/nix-community/dream2nix/blob/main/src/modules/flake-parts/makeOutputsArgs.nix)

perSystem.dream2nix.inputs.<name>.projects.<name>.translator

Translator to use

Type: string

Example:

```
[  
  "yarn-lock"  
  "package-json"  
]
```

Declared by:

[dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix](https://github.com/nix-community/dream2nix/blob/main/src/modules/flake-parts/makeOutputsArgs.nix)

perSystem.dream2nix.inputs.<name>.projects.<name>.version

Version of the project

Type: null or string

Default: null

Declared by:

[dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix](https://github.com/nix-community/dream2nix/blob/main/src/modules/flake-parts/makeOutputsArgs.nix)

perSystem.dream2nix.inputs.<name>.settings

Settings to customize dream2nix's behaviour.

This is likely to be removed in the future:

Quote from DavHau @ <https://github.com/nix-community/dream2nix/pull/399/file>

Eventually this option should be removed.

This custom settings merging logic I once implemented is an ugly quick hack, and not needed anymore since we now have the module system for merging options

Type: list of (attribute set)

Default: []

Example:

```
[  
 {  
   aggregate = true;  
 }  
 {  
   filter = <function>;  
   subsystemInfo = {  
     nodejs = 18;
```

```
    npmArgs = "--legacy-peer-deps";
  };
}
```

Declared by:

[dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix](#)

`perSystem.dream2nix.inputs.<name>.source`

Source of the package to build with dream2nix

Type: path or package

Declared by:

[dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix](#)

```
<perSystem.dream2nix.inputs.  
<name>.sourceOverrides
```

Override the sources of dependencies or top-level packages. Refer to the `dream-lock.json` for the package version to override. For more details, refer to <https://nix-community.github.io/dream2nix/intro/override-system.html>

Type: function that evaluates to a(n) lazy attribute set of attribute set of package

Default: <function>

Example:

Declared by:

[dream2nix_legacy/src/modules/flake-parts/makeOutputsArgs.nix](https://github.com/dream2nix/dream2nix-legacy/blob/main/src/modules/flake-parts/makeOutputsArgs.nix)

perSystem.dream2nix.instance

The dream2nix instance.

Type: raw value (*read only*)

Declared by:

[dream2nix_legacy/src/modules/flake-parts/interface.nix](https://github.com/dream2nix/dream2nix-legacy/blob/main/src/modules/flake-parts/interface.nix)

perSystem.dream2nix.outputs

The raw outputs that were generated for each input.

Type: lazy attribute set of lazy attribute set of raw value (*read only*)



Declared by:

[dream2nix_legacy/src/modules/flake-parts/interface.nix](https://github.com/dream2nix/dream2nix-legacy/blob/main/src/modules/flake-parts/interface.nix)

flake-parts.easyOverlay

WARNING

This module does NOT make *consuming* an overlay easy. This module is intended for *creating* overlays. While it is possible to consume the overlay created by this module using the `final` module argument, this is somewhat unconventional. Instead:

- *Avoid* overlays. Many flakes can do without them.
- Initialize `pkgs` yourself:

```
perSystem = { system, ... }: {
    _module.args.pkgs = import inputs.nixpkgs {
        inherit system;
        overlays = [
            inputs.foo.overlays.default
            (final: prev: {
                # ... things you really need to patch ...
            })
        ];
        config = { };
    };
};
```



Who this is for

This module is for flake authors who need to provide a simple overlay in addition to the common flake attributes. It is not for users who want to consume an overlay.

What it actually does

This module overrides the `pkgs` module argument and provides the `final` module argument so that the `perSystem` module can be evaluated as an overlay. Attributes added by the overlay must be defined in `overlayAttrs`. The resulting overlay is defined in the `overlays.default` output.

The resulting behavior tends to be not 100% idiomatic. A hand-written overlay would usually use `final` more often, but nonetheless it gets the job done for simple use cases; certainly the simple use cases where overlays aren't strictly necessary.

The status of this module

It has an unfortunate name and may be renamed. Alternatively, its functionality may be moved out of flake-parts, into some Nixpkgs module. Certainly until then, feel free to use the module if you understand what it does.

Installation

To use these options, add inside the `mkFlake` :

```
imports = [
    inputs.flake-parts.flakeModules.easyOverlay
];
```

Run `nix flake lock` and you're set.

Options

`perSystem`

`perSystem.overlayAttrs`

perSystem

A function from system to flake-like attributes omitting the `<system>` attribute.

Modules defined here have access to the suboptions and [some convenient module arguments](#).

Type: module

Declared by:

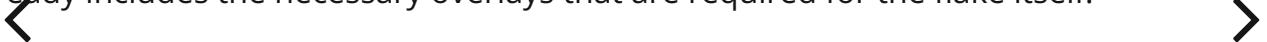
[flake-parts](#)

perSystem.overlayAttrs

Attributes to add to `overlays.default`.

The `overlays.default` overlay will re-evaluate `perSystem` with the “prev” (or “super”) overlay argument value as the `pkgs` module argument. The `easyOverlay` module also adds the `final` module argument, for the result of applying the overlay.

When not in an overlay, `final` defaults to `pkgs` plus the generated overlay. This requires Nixpkgs to be re-evaluated, which is more expensive than setting `pkgs` to a Nixpkgs that already includes the necessary overlays that are required for the flake itself.



See [Overlays](#).

Type: lazy attribute set of raw value

Default: { }

Declared by:

[flake-parts](#)



fla

[Introduction](#)[Getting Started](#)[Cheat Sheet](#)[Tutorials](#)[Best Practices for Module Writing](#)[Multi-platform: Working with system](#)[Guides](#)[Explore and debug option values](#)[Define a Module in a Separate File](#)[Define Custom Flake Attribute](#)[Generate Documentation](#)[Dogfood a Reusable Flake Module](#)[Explanation](#)[Overlays](#)[Reference Documentation](#)[Module Arguments](#)[Options](#)[flake-parts \(built in\)](#)[flakeModules](#)[easyOverlay \(warning\)](#)[agenix-shell](#)[devenv](#)[devshell](#)[dream2nix](#)[dream2nix legacy](#)[emanote](#)[ez-configs](#)[flake.parts-website](#)[haskell-flake](#)[hercules-ci-effects](#)[mission-control](#)[nix-cargo-integration](#)

emanote

[Emanote](#) renders your Markdown files as a nice static site.

Use `nix run` to run the live server, and `nix build` to build it.

See [emanote-template](#) for an example `flake.nix`.

Installation

To use these options, add to your flake inputs:

```
emanote.url = "github:srid/emanote";
```

and inside the `mkFlake`:

```
imports = [
    inputs.emanote.flakeModule
];
```

Run `nix flake lock` and you're set.

Options

[perSystem.emanote](#)

[perSystem.emanote.package](#)

[perSystem.emanote.sites](#)

[perSystem.emanote.sites.<name>.allowBrokenLink](#)

[perSystem.emanote.sites.<name>.basePath](#)

[perSystem.emanote.sites.<name>.baseUrl](#)

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

flakeModules

easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

haskell-flake

hercules-ci-effects

mission-control

nix-cargo-integration

`perSystem.emanote.sites.<name>.layers`

`perSystem.emanote.sites.<name>.layersString`

`perSystem.emanote.sites.<name>.port`

`perSystem.emanote.sites.<name>.prettyUrls`

perSystem.emanote

Emanote sites config

Type: submodule

Declared by:

`emanote/nix/flake-module.nix`

perSystem.emanote.package

The emanote package to use.

By default, the 'emanote' flake input will be used.

Type: package

Default: "inputs'.emanote.packages.default"

Declared by:

`emanote/nix/flake-module.nix`

perSystem.emanote.sites

Emanote sites

Type: attribute set of (submodule)

Declared by:

[emanote/nix/flake-module.nix](#)[Introduction](#)[Getting Started](#)[Cheat Sheet](#)[Tutorials](#)[Best Practices for Module Writing](#)[Multi-platform: Working with system](#)[Guides](#)[Explore and debug option values](#)[Define a Module in a Separate File](#)[Define Custom Flake Attribute](#)[Generate Documentation](#)[Dogfood a Reusable Flake Module](#)[Explanation](#)[Overlays](#)[Reference Documentation](#)[Module Arguments](#)[Options](#)[flake-parts \(built in\)](#)[flakeModules](#)[easyOverlay \(warning\)](#)[agenix-shell](#)[devenv](#)[devshell](#)[dream2nix](#)[dream2nix legacy](#)[emanote](#)[ez-configs](#)[flake.parts-website](#)[haskell-flake](#)[hercules-ci-effects](#)[mission-control](#)[nix-cargo-integration](#)

perSystem.emanote.sites.<name>.allowBrokenLinks

Allow broken links in the static site

Type: boolean

Default: false

Declared by:

[emanote/nix/flake-module.nix](#)

perSystem.emanote.sites.<n>

Top-level directory to copy the static site to

Type: string

Default: "Root path"

Declared by:

[emanote/nix/flake-module.nix](#)

perSystem.emanote.sites.<n>

Base URL for relative links

Type: string

Default: "Root URL"

Declared by:

[emanote/nix/flake-module.nix](#)

Introduction

Getting Started

Cheat Sheet

Tutorials

- Best Practices for Module Writing

- Multi-platform: Working with system

Guides

- Explore and debug option values

- Define a Module in a Separate File

- Define Custom Flake Attribute

- Generate Documentation

- Dogfood a Reusable Flake Module

Explanation

- Overlays

Reference Documentation

- Module Arguments

- Options

 - flake-parts (built in)

 - flakeModules

 - easyOverlay (warning)

 - agenix-shell

 - devenv

 - devshell

 - dream2nix

 - dream2nix legacy

 - emanote

 - ez-configs

 - flake.parts-website

 - haskell-flake

 - hercules-ci-effects

 - mission-control

 - nix-cargo-integration

perSystem.emanote.sites.<n

List of directory paths to run Emanote on

Type: list of path

Declared by:

<emanote/nix/flake-module.nix>

perSystem.emanote.sites.<n

Like `layers` but local (not in Nix store)

Type: list of string

Declared by:

<emanote/nix/flake-module.nix>

perSystem.emanote.sites.<n

Port to listen on

Type: signed integer

Default: "Random port"

Declared by:

<emanote/nix/flake-module.nix>

perSystem.emanote.sites.<n

Generate links without .html

Type: boolean

Default: false

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

 flakeModules

 easyOverlay (warning)

 agenix-shell

 devenv

 devshell

 dream2nix

 dream2nix legacy

 emanote

 ez-configs

 flake.parts-website

 haskell-flake

 hercules-ci-effects

 mission-control

 nix-cargo-integration

Declared by:

[emanote/nix/flake-module.nix](#)

Explore and debug option values

Sometimes the public interface of a flake is not enough. To inspect all option values, you can enable `debug` and explore otherwise private values with the repl.

Start debugging

1. Add `debug = true;` Example:

```
{  
  debug = true;  
  
  systems = /* ... */;  
  perSystem = /* ... */;  
<  }  >
```

2. Load the flake

```
$ nix repl  
nix-repl> :lf .
```

Inspect the `perSystem` configuration for your machine

```
nix-repl> currentSystem.allModuleArgs.pkgs.stdenv.hostPlatform.system  
"x86_64-linux"
```

Inspect the `perSystem` configuration for a different system type

```
nix-repl> debug.allSystems.armv7l-linux.allModuleArgs.pkgs.stdenv.hostPlatform.s  
"armv7l-linux"
```

Inspect a top level option

```
nix-repl> debug.systems  
[ "x86_64-linux" "aarch64-darwin" ]
```

Where is a per system value defined?

```
nix-repl> currentSystem.options.pre-commit.settings.files  
[ "/nix/store/pqp5kwdihyyymfnqq9sk9jsm9xw2lw6s-source/dev-module.nix", via option
```



Where is a top level value defined?

```
nix-repl> debug.options.system.files  
[ "/nix/store/3na6c6mmmyw2yf5chzwwwrp54b8yf96ry-source/flake.nix" ]
```

Where is a top level option declared?

```
nix-repl> debug.options.systems.declarations  
[ "/nix/store/3na6c6mmmyw2yf5chzwwwrp54b8yf96ry-source/modules/perSystem.nix" ]
```

See also

- The [debug option](#) reference.





ez-configs

`ez-configs` lets you define multiple nixos, darwin, and home manager configurations, and reuse common modules using your flake directory structure.

Installation

To use these options, add to your flake inputs:

```
ez-configs.url = "github:ehllie/ez-configs";
```

and inside the `mkFlake`:

```
imports = [  
    inputs.ez-configs.flakeModule  
], < >
```

Run `nix flake lock` and you're set.

Options

`ezConfigs.darwin.configurationsDirectory`

`ezConfigs.darwin.hosts`

`ezConfigs.darwin.hosts.<name>.importDefault`

`ezConfigs.darwin.hosts.<name>.userHomeModules`

`ezConfigs.darwin.modulesDirectory`

`ezConfigs.darwin.specialArgs`

`ezConfigs.globalArgs`

```
ezConfigs.home.configurationsDirectory  
ezConfigs.home.extraSpecialArgs  
ezConfigs.home.modulesDirectory  
ezConfigs.home.users  
ezConfigs.home.users.<name>.importDefault  
ezConfigs.home.users.<name>.nameFunction  
ezConfigs.home.users.<name>.passInOsConfig  
ezConfigs.home.users.<name>.standalone.enable  
ezConfigs.home.users.<name>.standalone.pkgs  
ezConfigs.nixos.configurationsDirectory  
ezConfigs.nixos.hosts  
ezConfigs.nixos.hosts.<name>.importDefault  
 ezConfigs.nixos.hosts.<name>.userHomeModules   
ezConfigs.nixos.modulesDirectory  
ezConfigs.nixos.specialArgs  
ezConfigs.root
```

ezConfigs.darwin.configurationsDirectory

The directory containing darwinConfigurations.

Type: path in the Nix store

Default: "\${ezConfigs.root}/darwin-configurations"

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.darwin.hosts

Settings for creating darwinConfigurations.

It's not necessary to specify this option to create flake outputs. It's only needed if you want to change the defaults for specific darwinConfigurations.

Type: attribute set of (submodule)

Default: { }

Example:

```
{  
  hostA = {  
    userHomeModules = [ "bob" ];  
  };
```



```
hostB = {  
  importDefault = false;  
  arch = "aarch64"  
}; < >
```

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.darwin.hosts.<name>.importDefault

Whether to import the default module for this host.

Type: boolean

Default: true

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.darwin.hosts. <name>.userHomeModules

List of users in \${ezConfigs.hm.usersDirectory}, whose configurations to import as home manager darwinModules. They will be put inside `home-manager.${user}.imports` list for this host.

When this list is not empty, the `home-manager.extraSpecialArgs` option is also set to the one it would receive in `homeManagerConfigurations` output, and the appropriate `homeManager` module is imported.

Type: list of string

Default: []

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.darwin.modulesDirectory >

The directory containing darwinModules.

Type: path in the Nix store

Default: "\${ezConfigs.root}/darwin-modules"

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.darwin.specialArgs

Extra arguments to pass to all darwinConfigurations.

Type: attribute set of anything

Default: `ezConfigs.globalArgs`

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.globalArgs

Extra arguments to pass to all configurations.

Type: attribute set of anything

Default: { }

Example: { inherit inputs; }

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.home.configurationsDirectory

The  directory containing homeConfigurations. 

Type: path in the Nix store

Default: "\${ezConfigs.root}/home-configurations"

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.home.extraSpecialArgs

Extra arguments to pass to all homeConfigurations.

Type: attribute set of anything

Default: ezConfigs.globalArgs

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.home.modulesDirectory

The directory containing homeModules.

Type: path in the Nix store

Default: "\${ezConfigs.root}/home-modules"

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.home.users

Settings for creating homeConfigurations.

It's not necessary to specify this option to create flake outputs. It's only needed if you want to change the defaults for specific homeConfigurations.

Type: attribute set of (submodule)

 
Default: { }

Example:

```
{  
  alice = {  
    standalone = {  
      enable = true;  
      pkgs = import nixpkgs { system = "x86_64-linux"; };  
    };  
  };  
};
```



```
bob = {  
importDefault = false;  
};  
}
```

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.home.users.<name>.importDefault

Whether to import the default module for this user.

Type: boolean

Default: true

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.home.users.<name>.nameFunction

Function to generate the name of the user configuration using the host name.

Type: null or (function that evaluates to a(n) string)

Default: \${username}@\${hostname}

Example: (host: "\${host}-\${name}")



Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.home.users.<name>.passInOsConfig

Whether to pass the osConfig argument to extraSpecialArgs. This will be the nixosConfiguration.config or darwinConfiguration.config, whose pkgs are being used to build this homeConfiguration.

Type: boolean

Default: true

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.home.users.<name>.standalone.enable

Whether to create a standalone user configuration.

By default each user and host pair gets its own homeConfigurations attribute, and the pkgs passed into homeConfiguration function come from that system.

This will prevent the \${user}@\${host} outputs from being created. Instead a standalone user configuration will be created with user name.

Type: boolean

Default: false

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.home.users.<name>.standalone.pkgs

The package set with which to construct the homeManagerConfiguration.

Non standalone user configurations will use the package set of the host system.

Type: Nixpkgs package set

Example: import nixpkgs {system = "x86_64-linux";}

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.nixos.configurationsDirectory

The directory containing nixosConfigurations.

Type: path in the Nix store

Default: "\${ezConfigs.root}/nixos-configurations"

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.nixos.hosts

Settings for creating nixosConfigurations.

It's not necessary to specify this option to create flake outputs. It's only needed if you want to change the defaults for specific nixosConfigurations.

Type: attribute set of (submodule)

Default: { }

Example:

```
{  
  hostA = {  
    userHomeModules = [ "bob" ];  
  }; >  
  hostB = {  
    importDefault = false;  
    arch = "aarch64"  
  };  
}
```

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.nixos.hosts.<name>.importDefault

Whether to import the default module for this host.

Type: boolean

Default: true

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.nixos.hosts.<name>.userHomeModules

List of users in \${ezConfigs.hm.usersDirectory}, whose configurations to import as home manager nixosModules. They will be put inside `home-manager.${user}.imports` list for this host.

When this list is not empty, the `home-manager.extraSpecialArgs` option is also set to the one it would receive in `homeManagerConfigurations` output, and the appropriate `homeManager` module is imported.

Type: list of string

Default: []

Declared by:

[ez-<configs/flake-module.nix](#)



ezConfigs.nixos.modulesDirectory

The directory containing nixosModules.

Type: path in the Nix store

Default: "\${ezConfigs.root}/nixos-modules"

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.nixos.specialArgs

Extra arguments to pass to all nixosConfigurations.

Type: attribute set of anything

Default: `ezConfigs.globalArgs`

Declared by:

[ez-configs/flake-module.nix](#)

ezConfigs.root

The root from which configurations and modules should be searched.

Type: null or path in the Nix store

Default: null

Example: `./.`

Declared by:

[ez-configs/flake-module.nix](#)



flake-parts.flakeModules

Adds the `flakeModules` attribute and `flakeModule` alias.

This module makes deduplication and `disabledModules` work, even if the definitions are inline modules or `importApply`.

Installation

To use these options, add inside the `mkFlake`:

```
imports = [
    inputs.flake-parts.flakeModules.easyOverlay
];
```

Run `nix flake lock` and you're set.



Options

`flake`

`flake.flakeModule`

`flake.flakeModules`

flake

Raw flake output attributes. Any attribute can be set here, but some attributes are represented by options, to provide appropriate configuration merging.

Type: lazy attribute set of raw value

Declared by:

[flake-parts](#)

flake.flakeModule

Alias of `flakeModules.default`.

Type: submodule

Declared by:

[flake-parts](#)

flake.flakeModules

flake-parts modules for use by other flakes.

If the flake defines only one module, it should be `flakeModules.default`.

You ~~can~~ not read this option in defining the flake's own `imports`. Instead, you can put ~~the~~ the module in question into its own file or let binding and reference it both in `imports` and export it with this option.

See [Dogfood a Reusable Module](#) for details and an example.

Type: lazy attribute set of module

Default: `{ }`

Declared by:

[flake-parts](#)

Core Options

These options are provided by default. They reflect what Nix expects, plus a small number of helpful options, notably [perSystem](#).

Options

[debug](#)[flake](#)[flake.packages](#)[flake.apps](#)[flake.apps.<name>.<name>.program](#)[flake.apps.<name>.<name>.type](#)[flake.checks](#)[flake.devShells](#)[flake.flakeModule](#)[flake.flakeModules](#)[flake.formatter](#)[flake.legacyPackages](#)[flake.nixosConfigurations](#)[flake.nixosModules](#)[flake.overlays](#)[perInput](#)

`perSystem`

`perSystem.packages`

`perSystem.apps`

`perSystem.apps.<name>.program`

`perSystem.apps.<name>.type`

`perSystem.checks`

`perSystem.debug`

`perSystem.devShells`

`perSystem.formatter`

`perSystem.legacyPackages`

`perSystem.overlayAttrs`

`systems`

`< transposition >`

`transposition.<name>.adHoc`

debug

Whether to add the attributes `debug`, `allSystems` and `currentSystem` to the flake output.
When `true`, this allows inspection of options via `nix repl`.

```
$ nix repl
nix-repl> :lf .
nix-repl> currentSystem._module.args.pkgs.hello
<derivation /nix/store/7vf0d0j7majv1ch1xymdyql80cn5fp-hello-2.12.1.drv>
```

Each of `debug`, `allSystems.<system>` and `currentSystem` is an attribute set consisting of the `config` attributes, plus the extra attributes `_module`, `config`, `options`, `extendModules`. So note that these are not part of the `config` parameter, but are merged in for debugging convenience.

- `debug` : The top-level options
- `allSystems` : The `perSystem` submodule applied to the configured `systems`.
- `currentSystem` : Shortcut into `allSystems`. Only available in impure mode. Works for arbitrary system values.

See [Explore and debug option values](#) for more examples.

Type: boolean

Default: `false`

Declared by:

[flake-parts/modules/debug.nix](#)

flake

Raw flake output attributes. Any attribute can be set here, but some attributes are represented by options, to provide appropriate configuration merging.

Type: lazy attribute set of raw value

Declared by:

[flake-parts/extras/flakeModules.nix](#), [flake-parts/modules/packages.nix](#), [flake-parts/modules/overlays.nix](#), [flake-parts/modules/nixosModules.nix](#), [flake-parts/modules/nixosConfigurations.nix](#), [flake-parts/modules/legacyPackages.nix](#), [flake-parts/modules/formatter.nix](#), [flake-parts/modules/flake.nix](#), [flake-parts/modules/devShells.nix](#), [flake-parts/modules/checks.nix](#), [flake-parts/modules/apps.nix](#)

flake.packages

See `perSystem.packages` for description and examples.

Type: lazy attribute set of lazy attribute set of package

Default: `{ }`

Declared by:

[flake-parts/modules/packages.nix](#)

flake.apps

See `perSystem.apps` for description and examples.

Type: lazy attribute set of lazy attribute set of (submodule)

Default: { }

Declared by:

[flake-parts/modules/apps.nix](#)

flake.apps.<name>.<name>.program

A  path to an executable or a derivation with `meta.mainProgram`. 

Type: string or package convertible to it

Declared by:

[flake-parts/modules/apps.nix](#)

flake.apps.<name>.<name>.type

A type tag for `apps` consumers.

Type: value “app” (singular enum)

Default: "app"

Declared by:

[flake-parts/modules/apps.nix](#)

flake.checks

See `perSystem.checks` for description and examples.

Type: lazy attribute set of lazy attribute set of package

Default: { }

Declared by:

[flake-parts/modules/checks.nix](#)



flake.devShells

See `perSystem.devShells` for description and examples.

Type: lazy attribute set of lazy attribute set of package

Default: { }

Declared by:

[flake-parts/modules/devShells.nix](#)



flake.flakeModule

Alias of `flakeModules.default`.

Type: submodule

Declared by:

[flake-parts/extras/flakeModules.nix](#)



flake.flakeModules

flake-parts modules for use by other flakes.

If the flake defines only one module, it should be `flakeModules.default`.

You can not read this option in defining the flake's own `imports`. Instead, you can put the module in question into its own file or let binding and reference it both in `imports` and export it with this option.

See [Dogfood a Reusable Module](#) for details and an example.

Type: lazy attribute set of module

Default: { }

Declared by:

[flake-parts/extras/flakeModules.nix](#)



flake.formatter

An attribute set of per system a package used by `nix fmt`.

Type: lazy attribute set of package

Default: { }

Declared by:

[flake-parts/modules/formatter.nix](#)



flake.legacyPackages

See `perSystem.legacyPackages` for description and examples.

Type: lazy attribute set of lazy attribute set of raw value

Default: { }

Declared by:

[flake-parts/modules/legacyPackages.nix](#)

flake.nixosConfigurations

Instantiated NixOS configurations. Used by `nixos-rebuild`.

`nixosConfigurations` is for specific machines. If you want to expose reusable configurations, add them to `nixosModules` in the form of modules (no `lib.nixosSystem`), so that you can reference them in this or another flake's `nixosConfigurations`.

Type: lazy attribute set of raw value

Default: { }

Example:

```
{  
    my-machine = inputs.nixpkgs.lib.nixosSystem {  
        # system is not needed with freshly generated hardware-configuration.nix  
        # system = "x86_64-linux"; # or set nixpkgs.hostPlatform in a module.  
        modules = [  
            ./my-machine/nixos-configuration.nix  
            config.nixosModules.my-module  
        ];  
    };  
}
```



Declared by:

[flake-parts/modules/nixosConfigurations.nix](#)

flake.nixosModules

NixOS modules.

You may use this for reusable pieces of configuration, service modules, etc.

Type: lazy attribute set of unspecified value

Default: { }

Declared by:

[flake-parts/modules/nixosModules.nix](#)

flake.overlays

An attribute set of [overlays](#).

Note that the overlays themselves are not mergeable. While overlays can be composed, the order of composition is significant, but the module system does not guarantee sufficiently deterministic definition ordering, across versions and when changing `imports`.

Type: lazy attribute set of function that evaluates to a(n) function that evaluates to a(n) lazy attribute set of unspecified value

Default: `{ }`

Example:

```
{  
  default = final: prev: {};  
}
```

Declared by:

[flake-parts/modules/overlays.nix](#)



perInput

A function that pre-processes flake inputs.

It is called for users of `persystem` such that `inputs'}.${name} = config.perInput system inputs.${name}`.

This is used for `inputs'` and `self'`.

The attributes returned by the `perInput` function definitions are merged into a single namespace (per input), so each module should return an attribute set with usually only one or two predictable attribute names. Otherwise, the `inputs'` namespace gets polluted.

Type: function that evaluates to a(n) function that evaluates to a(n) lazy attribute set of unspecified value

Declared by:

[flake-parts/modules/persystem.nix](#)

perSystem

A function from system to flake-like attributes omitting the `<system>` attribute.

Modules defined here have access to the suboptions and [some convenient module arguments](#).

Type: module

Declared by:

`flake-parts/extras/easyOverlay.nix`, `flake-parts/modules/withSystem.nix`, `flake-parts/modules/perSystem.nix`, `flake-parts/modules/packages.nix`, `flake-parts/modules/legacyPackages.nix`, `flake-parts/modules/formatter.nix`, `flake-parts/modules/devShells.nix`, `flake-parts/modules/debug.nix`, `flake-parts/modules/checks.nix`, `flake-parts/modules/apps.nix`

perSystem.packages

An attribute set of packages to be built by `nix build`.



`nix build .#<name>` will build `packages.<name>`.

Type: lazy attribute set of package

Default: { }

Declared by:

`flake-parts/modules/packages.nix`

perSystem.apps

Programs runnable with `nix run <name>`.

Type: lazy attribute set of (submodule)

Default: { }

Example:

```
{  
    default.program = "${config.packages.hello}/bin/hello";  
}
```

Declared by:

[flake-parts/modules/apps.nix](#)

perSystem.apps.<name>.program

A path to an executable or a derivation with `meta.mainProgram`.

Type: string or package convertible to it

Declared by:

[flake-parts/modules/apps.nix](#)

perSystem.apps.<name>.type

A type tag for `apps` consumers.

Type: value “app” (singular enum)

Default: "app"

Declared by:

[flake-parts/modules/apps.nix](#)

perSystem.checks

Derivations to be built by `nix flake check`.

Type: lazy attribute set of package

Default: { }

Declared by:

`flake-parts/modules/checks.nix`

perSystem.debug

Values to return in e.g. `allSystems.<system>` when `debug = true`.

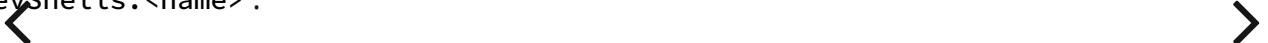
Type: lazy attribute set of raw value

Declared by:

`flake-parts/modules/formatter.nix`

perSystem.devShells

An attribute set of packages to be used as shells. `nix develop .#<name>` will run `devShells.<name>`.



Type: lazy attribute set of package

Default: `{ }`

Example:

```
{
  default = pkgs.mkShell {
    nativeBuildInputs = with pkgs; [ wget bat cargo ];
  };
}
```

Declared by:

`flake-parts/modules/devShells.nix`

perSystem.formatter

A package used by `nix fmt`.

Type: null or package

Default: null

Declared by:

[flake-parts/modules/formatter.nix](#)

perSystem.legacyPackages

An attribute set of unmergeable values. This is also used by `nix build .#<attrpath>`.

Type: lazy attribute set of raw value

Default: { }

Declared by:

[flake-parts/modules/legacyPackages.nix](#)



perSystem.overlayAttrs

Attributes to add to `overlays.default`.

The `overlays.default` overlay will re-evaluate `perSystem` with the “prev” (or “super”) overlay argument value as the `pkgs` module argument. The `easyOverlay` module also adds the `final` module argument, for the result of applying the overlay.

When not in an overlay, `final` defaults to `pkgs` plus the generated overlay. This requires Nixpkgs to be re-evaluated, which is more expensive than setting `pkgs` to a Nixpkgs that already includes the necessary overlays that are required for the flake itself.

See [Overlays](#).

Type: lazy attribute set of raw value

Default: { }

Declared by:

[flake-parts/extras/easyOverlay.nix](#)

systems

All the system types to enumerate in the flake output subattributes.

In other words, all valid values for `system` in e.g. `packages.<system>.foo`.

Type: list of string

Declared by:

[flake-parts/modules/perSystem.nix](#)

transposition

A helper that defines transposed attributes in the flake outputs.

When you define `transposition.foo = { };`, definitions are added to the effect of (pseudo-code):

```
flake.foo.${system} = (perSystem system).foo;  
perInput = system: inputFlake: inputFlake.foo.${system};
```



Transposition is the operation that swaps the indices of a data structure. Here it refers specifically to the transposition between

```
perSystem: .${system}.${attribute}  
outputs:   .${attribute}.${system}
```

It also defines the reverse operation in `perInput`.

Type: lazy attribute set of (submodule)

Declared by:

[flake-parts/modules/transposition.nix](#)

transposition.<name>.adHoc

Whether to provide a stub option declaration for `perSystem.<name>`.

The stub option declaration does not support merging and lacks documentation, so you are recommended to declare the `perSystem.<name>` option yourself and avoid `adHoc`.

Type: boolean

Default: `false`

Declared by:

[flake-parts/modules/transposition.nix](#)





Getting Started

New flake

If your project does not have a flake yet:

```
nix flake init -t github:hercules-ci/flake-parts
```

Existing flake

Otherwise, add the input,

```
flake-parts.url = "github:hercules-ci/flake-parts"; >
```

then slide `mkFlake` between your outputs function head and body,

```
outputs = inputs@{ flake-parts, ... }:
  flake-parts.lib.mkFlake { inherit inputs; } {
    flake = {
      # Put your original flake attributes here.
    };
    systems = [
      # systems for which you want to build the `perSystem` attributes
      "x86_64-linux"
      # ...
    ];
    perSystem = { config, ... }: {
    };
  };
};
```

Now you can start using the flake-parts options.



flc

[Introduction](#)[Getting Started](#)[Cheat Sheet](#)[Tutorials](#)[Best Practices for Module Writing](#)[Multi-platform: Working with system](#)[Guides](#)[Explore and debug option values](#)[Define a Module in a Separate File](#)[Define Custom Flake Attribute](#)[Generate Documentation](#)[Dogfood a Reusable Flake Module](#)[Explanation](#)[Overlays](#)[Reference Documentation](#)[Module Arguments](#)[Options](#)[flake-parts \(built in\)](#)[flakeModules](#)[easyOverlay \(warning\)](#)[agenix-shell](#)[devenv](#)[devshell](#)[dream2nix](#)[dream2nix legacy](#)[emanote](#)[ez-configs](#)[flake.parts-website](#)[haskell-flake](#)[hercules-ci-effects](#)[mission-control](#)[nix-cargo-integration](#)

haskell-flake

[haskell-flake](#) scans your flake files for Haskell projects using the Nixpkgs Haskell infrastructure.

It also provides [checks](#) and [devShells](#)

Multiple projects can be declared to represent each project's frontends.

Installation

To use these options, add to your flake inputs:

```
haskell-flake.url = "github:srid/haskell-flake"
```

and inside the `mkFlake`:

```
imports = [
    inputs.haskell-flake.flakeModule
];
```

Run `nix flake lock` and you're set.

Options

[flake.haskellFlakeProjectModules](#)

[perSystem.haskellProjects](#)

[perSystem.haskellProjects.<name>.packages](#)

[perSystem.haskellProjects.<name>.autoWire](#)

[perSystem.haskellProjects.<name>.basePackages](#)

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

 flakeModules

 easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

haskell-flake

hercules-ci-effects

mission-control

nix-cargo-integration

perSystem.haskellProjects.<name>.defaults.enabled

perSystem.haskellProjects.<name>.defaults.package

perSystem.haskellProjects.<name>.defaults.devShell

perSystem.haskellProjects.<name>.defaults.project

perSystem.haskellProjects.<name>.defaults.settings

perSystem.haskellProjects.<name>.devShell

perSystem.haskellProjects.<name>.devShell.enabled

perSystem.haskellProjects.<name>.devShell.benchmark

perSystem.haskellProjects.<name>.devShell.extra

perSystem.haskellProjects.<name>.devShell.haskellCI

perSystem.haskellProjects.<name>.devShell.haskellCI

perSystem.haskellProjects.<name>.devShell.haskellCI

perSystem.haskellProjects.<name>.devShell.haskellCI

perSystem.haskellProjects.<name>.mkShell

perSystem.haskellProjects.<name>.devShell.tools

perSystem.haskellProjects.<name>.outputs

perSystem.haskellProjects.<name>.outputs.packages

perSystem.haskellProjects.<name>.outputs.packages

perSystem.haskellProjects.<name>.outputs.packages

perSystem.haskellProjects.<name>.outputs.packages

perSystem.haskellProjects.<name>.outputs.packages

perSystem.haskellProjects.<name>.outputs.apps

perSystem.haskellProjects.<name>.outputs.apps

perSystem.haskellProjects.<name>.outputs.apps

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

flakeModules

easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

[haskell-flake](#)

hercules-ci-effects

mission-control

nix-cargo-integration

`perSystem.haskellProjects.<name>.outputs.check`

`perSystem.haskellProjects.<name>.outputs.devShell`

`perSystem.haskellProjects.<name>.outputs.final`

`perSystem.haskellProjects.<name>.projectFlakeName`

`perSystem.haskellProjects.<name>.projectRoot`

`perSystem.haskellProjects.<name>.settings`

flake.haskellFlakeProjectM

A lazy attrset of `haskellProjects.<name>` modules to

Type: lazy attribute set of module

Default: Package and dependency information for this flake, when using this project as a Haskell dependency.

The ‘output’ module of the default project is included in `defaults.projectModules.output`.

Declared by:

`haskell-flake/nix/modules/project-modules.nix`

perSystem.haskellProjects

Haskell projects

Type: attribute set of (submodule)

Declared by:

`haskell-flake/nix/modules/projects.nix`

perSystem.haskellProjects.

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

 flakeModules

 easyOverlay (warning)

 agenix-shell

 devenv

 devshell

 dream2nix

 dream2nix legacy

 emanote

 ez-configs

 flake.parts-website

[haskell-flake](#)

 hercules-ci-effects

 mission-control

 nix-cargo-integration

Additional packages to add to `basePackages`.

Local packages are added automatically (see `config`)

You can also override the source for existing packages

Type: lazy attribute set of module

Default: { }

Declared by:

[haskell-flake/nix/modules/project/packages](#)

perSystem.haskellProjects.

List of flake output types to autowire.

Using an empty list will disable autowiring entirely, or using `config.haskellProjects.<name>.outputs`.

Type: list of (one of "packages", "checks", "apps", "devShells")

Default:

```
[  
  "packages"  
  "checks"  
  "apps"  
  "devShells"  
]
```

Declared by:

[haskell-flake/nix/modules/project](#)

perSystem.haskellProjects. <name>.basePackages

Which Haskell package set / compiler to use.

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

flakeModules

easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

[haskell-flake](#)

hercules-ci-effects

mission-control

nix-cargo-integration

You can effectively select the GHC version here.

To get the appropriate value, run:

```
nix-env -f "<nixpkgs>" -qaP -A haskell.compiler.ghc
```

And then, use that in `pkgs.haskell.packages.ghc<version>`

Type: attribute set of raw value

Default: `pkgs.haskellPackages`

Example: `"pkgs.haskell.packages.ghc924"`

Declared by:

```
haskell-flake/nix/modules/project
```

`perSystem.haskellProjects.<name>.defaults.enable`

Whether to enable haskell-flake's default settings for

Type: boolean

Default: `true`

Declared by:

```
haskell-flake/nix/modules/project/defaults.nix
```

`perSystem.haskellProjects.<name>.defaults.packages`

Local packages scanned from `projectRoot`

Type: lazy attribute set of module

Default: If you have a `cabal.project` file (under `projectRoot`) automatically discovered. Otherwise, the top-level `cabal.project`

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

 flakeModules

 easyOverlay (warning)

 agenix-shell

 devenv

 devshell

 dream2nix

 dream2nix legacy

 emanote

 ez-configs

 flake.parts-website

 haskell-flake

 hercules-ci-effects

 mission-control

 nix-cargo-integration

local package.

haskell-flake currently supports a limited range of syi
requires an explicit list of package directories under t

Declared by:

[haskell-flake/nix/modules/project/defaults.nix](https://flake.parts/nix/modules/project/defaults.nix)

perSystem.haskellProjects.<name>.defaults.devShell.to

Build tools always included in devShell

Type: function that evaluates to a(n) attribute set of (r

Default: <function>

Declared by:

[haskell-flake/nix/modules/project/defaults.nix](https://flake.parts/nix/modules/project/defaults.nix)

perSystem.haskellProjects.<name>.defaults.projectModi

A haskell-flake project module that exports the `package` consuming flake. This enables the use of this flake's `package` using its overrides.

Type: module

Default: a generated module

Declared by:

[haskell-flake/nix/modules/project/defaults.nix](https://flake.parts/nix/modules/project/defaults.nix)

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

flakeModules

easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

[haskell-flake](#)

hercules-ci-effects

mission-control

nix-cargo-integration

perSystem.haskellProjects. <name>.defaults.settings.de

Default settings for all packages in packages option.

Type: module

Default:

''

Speed up builds by disabling haddock and lik



This uses local.toDefinedProject option to determine override. Thus, it applies to both local packages (transitively imported packages that are local to haskell-flake). The goal being to use the same consistently for all packages using haskell-fla

''

Declared by:

[haskell-flake/nix/modules/project/defaults.nix](#)

perSystem.haskellProjects.

Development shell configuration

Type: submodule

Default: { }

Declared by:

[haskell-flake/nix/modules/project/hls-check.nix](#)
[/project/devshell.nix](#)

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

 flakeModules

 easyOverlay (warning)

 agenix-shell

 devenv

 devshell

 dream2nix

 dream2nix legacy

 emanote

 ez-configs

 flake.parts-website

 haskell-flake

 hercules-ci-effects

 mission-control

 nix-cargo-integration

perSystem.haskellProjects.<name>.devShell.enable

Whether to enable a development shell for the project.

Type: boolean

Default: true

Declared by:

[haskell-flake/nix/modules/project/devshell.nix](https://flake.parts/nix/modules/project/devshell.nix)

perSystem.haskellProjects.<name>.devShell.benchmark

Whether to include benchmark dependencies in the build. This option will set the corresponding doBenchmark flag in the build command.

Type: boolean

Default: false

Declared by:

[haskell-flake/nix/modules/project/devshell.nix](https://flake.parts/nix/modules/project/devshell.nix)

perSystem.haskellProjects.<name>.devShell.extraLibraries

Extra Haskell libraries available in the shell's environment. These can be runghc and ghci for instance.

The argument is the Haskell package set.

The return type is an attribute set for overridability and substitutability.

Type: null or (function that evaluates to a(n) attribute

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

flakeModules

easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

[haskell-flake](#)

hercules-ci-effects

mission-control

nix-cargo-integration

Default: null

Example: hp: { inherit (hp) releaser; }

Declared by:

[haskell-flake/nix/modules/project/devshell.nix](#)

perSystem.haskellProjects.<name>.devShell.hlsCheck

A [check](#) to make sure that your IDE will work.

Type: submodule

Default: { }

Declared by:

[haskell-flake/nix/modules/project/hls-check.nix](#)

perSystem.haskellProjects.<name>.devShell.hlsCheck.enabled

Whether to enable a flake check to verify that HLS works correctly.

This is equivalent to `nix develop -i -c haskell-language-server`.

Note that, HLS will try to access the network through [/haskell-language-server/issues/3128](#), therefore sanitizing the URL before evaluating this check.

Type: boolean

Default: false

Declared by:

[haskell-flake/nix/modules/project/hls-check.nix](#)

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

flakeModules

easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

[haskell-flake](#)

hercules-ci-effects

mission-control

nix-cargo-integration

perSystem.haskellProjects.<name>.devShell.hlsCheck.dir

The `hlsCheck` derivation generated for this project.

Type: package (*read only*)

Declared by:

[haskell-flake/nix/modules/project/hls-check.nix](#)

perSystem.haskellProjects.<name>.devShell.hoogle

Whether to include Hoogle in the development shell.
corresponding `withHoogle` flag in the `shellFor` derivation.

Type: boolean

Default: true

Declared by:

[haskell-flake/nix/modules/project/devshell.nix](#)

perSystem.haskellProjects.<name>.devShell.mkShellArgs

Extra arguments to pass to `pkgs.mkShell`.

Type: attribute set of raw value

Default: { }

Example:

```
''  
{  
    shellHook = '\\'`
```

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

flakeModules

easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

[haskell-flake](#)

hercules-ci-effects

mission-control

nix-cargo-integration

```
# Re-generate .cabal files so HLS will v
' ${pkgs.findutils}/bin/find -name packa
  \'';
}
"
```

Declared by:

[haskell-flake/nix/modules/project/devshell.nix](https://flake.parts/nix/modules/project/devshell.nix)

perSystem.haskellProjects.<name>.devShell.tools

Build tools for developing the Haskell project.

These tools are merged with the `haskell-flake` default `defaults.devShell.tools` option. Set the value to `r`

Type: function that evaluates to a(n) attribute set of (r

Default: <function>

Declared by:

[haskell-flake/nix/modules/project/devshell.nix](https://flake.parts/nix/modules/project/devshell.nix)

perSystem.haskellProjects.

The flake outputs generated for this project.

This is an internal option, not meant to be set by the

Type: submodule

Declared by:

[haskell-flake/nix/modules/project/hls-check.nix](https://flake.parts/nix/modules/project/hls-check.nix), [haskell-flake/nix/modules/project/outputs.nix](https://flake.parts/nix/modules/project/outputs.nix), [haskell-flake/nix/modules/project/outputs.nix](https://flake.parts/nix/modules/project/outputs.nix)

Introduction

Getting Started

Cheat Sheet

Tutorials

- Best Practices for Module Writing

- Multi-platform: Working with system

Guides

- Explore and debug option values

- Define a Module in a Separate File

- Define Custom Flake Attribute

- Generate Documentation

- Dogfood a Reusable Flake Module

Explanation

- Overlays

Reference Documentation

- Module Arguments

- Options

 - flake-parts (built in)

 - flakeModules

 - easyOverlay (warning)

 - agenix-shell

 - devenv

 - devshell

 - dream2nix

 - dream2nix legacy

 - emanote

 - ez-configs

 - flake.parts-website

 - [haskell-flake](#)

 - hercules-ci-effects

 - mission-control

 - nix-cargo-integration

perSystem.haskellProjects. <name>.outputs.packages

Package information for all local packages. Contains 1

- package : The Haskell package derivation
- exes : Attrset of executables found in the .cabal

Type: attribute set of (submodule) (*read only*)

Declared by:

[haskell-flake/nix/modules/project/outputs.nix](#)

perSystem.haskellProjects. <name>.outputs.packages.<n>

The local package derivation.

Type: package

Declared by:

[haskell-flake/nix/modules/project/outputs.nix](#)

perSystem.haskellProjects. <name>.outputs.packages.<n>

Attrset of executables from .cabal file.

If the associated Haskell project has a separate bin output, then this exe will refer only to the bin output.

NOTE: Evaluating up to this option will involve IFD.

Type: attribute set of (submodule)

Declared by:

[haskell-flake/nix/modules/project/outputs.nix](https://flake.parts/nix/modules/project/outputs.nix)

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

 flakeModules

 easyOverlay (warning)

 agenix-shell

 devenv

 devshell

 dream2nix

 dream2nix legacy

 emanote

 ez-configs

 flake.parts-website

[haskell-flake](#)

 hercules-ci-effects

 mission-control

 nix-cargo-integration

perSystem.haskellProjects.<name>.outputs.packages.<name>.program

A path to an executable or a derivation with `meta.name`

Type: string or package convertible to it

Declared by:

[haskell-flake/nix/modules/project/outputs.nix](https://flake.parts/nix/modules/project/outputs.nix)

perSystem.haskellProjects.<name>.outputs.packages.<name>.type

A type tag for `apps` consumers.

Type: value "app" (singular enum)

Default: "app"

Declared by:

[haskell-flake/nix/modules/project/outputs.nix](https://flake.parts/nix/modules/project/outputs.nix)

perSystem.haskellProjects.<name>.outputs.apps

Flake apps for each Cabal executable in the project.

Type: attribute set of (submodule) (*read only*)

Declared by:

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

flakeModules

easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

[haskell-flake](#)

hercules-ci-effects

mission-control

nix-cargo-integration

[haskell-flake/nix/modules/project/outputs.nix](#)

perSystem.haskellProjects.<name>.outputs.apps.<name>

A path to an executable or a derivation with `meta.name`

Type: string or package convertible to it

Declared by:

[haskell-flake/nix/modules/project/outputs.nix](#)

perSystem.haskellProjects.<name>.outputs.apps.<name>

A type tag for `apps` consumers.

Type: value “app” (singular enum)

Default: "app"

Declared by:

[haskell-flake/nix/modules/project/outputs.nix](#)

perSystem.haskellProjects.<name>.outputs.checks

The flake checks generated for this project.

Type: lazy attribute set of package (*read only*)

Declared by:

[haskell-flake/nix/modules/project/hls-check.nix](#)

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

 flakeModules

 easyOverlay (warning)

 agenix-shell

 devenv

 devshell

 dream2nix

 dream2nix legacy

 emanote

 ez-configs

 flake.parts-website

 haskell-flake

 hercules-ci-effects

 mission-control

 nix-cargo-integration

perSystem.haskellProjects. <name>.outputs.devShell

The development shell derivation generated for this |

Type: package (*read only*)

Declared by:

[haskell-flake/nix/modules/project/devshell.nix](https://flake.parts/nix/modules/project/devshell.nix)

perSystem.haskellProjects. <name>.outputs.finalPackage

The final Haskell package set including local packages:
basePackages .

Type: attribute set of raw value (*read only*)

Declared by:

[haskell-flake/nix/modules/project/outputs.nix](https://flake.parts/nix/modules/project/outputs.nix)

perSystem.haskellProjects. <name>.projectFlakeName

A descriptive name for the flake in which this project

If unspecified, the Nix store path's basename will be |

Type: null or string

Default: null

Declared by:

[haskell-flake/nix/modules/project](https://flake.parts/nix/modules/project)

Introduction

Getting Started

Cheat Sheet

Tutorials

Best Practices for Module Writing

Multi-platform: Working with system

Guides

Explore and debug option values

Define a Module in a Separate File

Define Custom Flake Attribute

Generate Documentation

Dogfood a Reusable Flake Module

Explanation

Overlays

Reference Documentation

Module Arguments

Options

flake-parts (built in)

flakeModules

easyOverlay (warning)

agenix-shell

devenv

devshell

dream2nix

dream2nix legacy

emanote

ez-configs

flake.parts-website

[haskell-flake](#)

hercules-ci-effects

mission-control

nix-cargo-integration

perSystem.haskellProjects.

Path to the root of the project directory.

Chaning this affects certain functionality, like where t

Type: path

Default: "Top-level directory of the flake"

Declared by:

[haskell-flake/nix/modules/project](#)

perSystem.haskellProjects.

Overrides for packages in basePackages and packageOverrides

Attr values are submodules that take the following attributes:

- name : Package name
- package : The reference to the package in packageOverrides
- self / super : The 'self' and 'super' (aka. 'final') attribute
- pkgs : Nixpkgs instance of the module user (implied)

Default settings are defined in `project.config.default` and can be overridden.

Type: lazy attribute set of module

Default: { }

Declared by:

[haskell-flake/nix/modules/project/settings](#)

hercules-ci-effects

This module provides

- a mergeable `herculesCI` attribute; read by [Hercules CI](#) and the `hci` command,
- the `hci-effects` library as a module argument in `perSystem` / `withSystem`,
- ready to go, configurable continuous deployment jobs

Installation

To use these options, add to your flake inputs:

```
hercules-ci-effects.url = "github:hercules-ci/hercules-ci-effects";
```

and inside the `mkFlake`:



```
imports = [
    inputs.hercules-ci-effects.flakeModule
];
```

Run `nix flake lock` and you're set.

Options

`defaultEffectSystem`

`hercules-ci.flake-update.enable`

`hercules-ci.flake-update.autoMergeMethod`

`hercules-ci.flake-update.baseBranch`

`hercules-ci.flake-update.baseMerge.enable`

`hercules-ci.flake-update.baseMerge.branch`

hercules-ci.flake-update.baseMerge.method
hercules-ci.flake-update.createPullRequest
hercules-ci.flake-update.effect.settings
hercules-ci.flake-update.effect.system
hercules-ci.flake-update.flakes
hercules-ci.flake-update.flakes.<name>.commitSummary
hercules-ci.flake-update.flakes.<name>.inputs
hercules-ci.flake-update.forgeType
hercules-ci.flake-update.nix.package
hercules-ci.flake-update.pullRequestBody
hercules-ci.flake-update.pullRequestTitle
hercules-ci.flake-update.updateBranch
 hercules-ci.flake-update.when 
hercules-ci.github-pages.branch
hercules-ci.github-pages.check.enable
hercules-ci.github-pages.pushJob
hercules-ci.github-pages.settings
hercules-ci.github-releases.checkArtifacts
hercules-ci.github-releases.condition
hercules-ci.github-releases.files
hercules-ci.github-releases.files.*.archiver
hercules-ci.github-releases.files.*.label
hercules-ci.github-releases.files.*.path
hercules-ci.github-releases.files.*.paths

```
hercules-ci.github-releases.filesPerSystem
hercules-ci.github-releases.filesPerSystem.<function body>.*.archiver
hercules-ci.github-releases.filesPerSystem.<function body>.*.label
hercules-ci.github-releases.filesPerSystem.<function body>.*.path
hercules-ci.github-releases.filesPerSystem.<function body>.*.paths
hercules-ci.github-releases.pushJobName
hercules-ci.github-releases.releaseTag
hercules-ci.github-releases.systems
herculesCI
herculesCI.ciSystems
herculesCI.onPush
herculesCI.onPush.<name>.outputs
<herculesCI.onSchedule >
herculesCI.onSchedule.<name>.outputs
herculesCI.onSchedule.<name>.when
herculesCI.onSchedule.<name>.when.dayOfMonth
herculesCI.onSchedule.<name>.when.dayOfWeek
herculesCI.onSchedule.<name>.when.hour
herculesCI.onSchedule.<name>.when.minute
herculesCI.repo
herculesCI.repo.branch
herculesCI.repo.forgeType
herculesCI.repo.name
herculesCI.repo.owner
```

`herculesCI.repo.ref`

`herculesCI.repo.remoteHttpUrl`

`herculesCI.repo.remoteSshUrl`

`herculesCI.repo.rev`

`herculesCI.repo.shortRev`

`herculesCI.repo.tag`

`herculesCI.repo.webUrl`

`perSystem.hercules-ci.github-pages.settings`

`perSystem.herculesCIEffects.pkgs`

defaultEffectSystem

The default system type that some integrations will use to run their effects on.

Type: string

Default: "x86_64-linux"

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-helpers.nix](#)

hercules-ci.flake-update.enable

Whether to create a scheduled flake update job.

For a complete example, see the [hercules-ci-effects documentation](#) on `hercules-ci.flake-update`.

Requires `hercules-ci-agent 0.9.8 or newer`.

Type: boolean

Default: false

Example: true

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](https://github.com/herculesci/hercules-ci-effects/blob/main/effects/flake-update/flake-module.nix)

hercules-ci.flake-update.autoMergeMethod

Whether to enable auto-merge on new pull requests, and how to merge it.

This requires [GitHub branch protection](#) to be configured for the repository.

Type: one of <null>, "merge", "rebase", "squash"

Default: null

Declared by:

[<hercules-ci-effects/effects/flake-update/flake-module.nix>](https://github.com/herculesci/hercules-ci-effects/blob/main/effects/flake-update/flake-module.nix)

hercules-ci.flake-update.baseBranch

Branch name on the remote that the update branch will be

- based on (via `hercules-ci.flake-update.baseMerge.branch`), and
- merged back into if `hercules-ci.flake-update.createPullRequest` is enabled.

"HEAD" refers to the default branch, which is often `main` or `master`.

Type: string

Default: "HEAD"

Example: "develop"

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](https://github.com/herculesci/hercules-ci-effects/blob/main/effects/flake-update/flake-module.nix)

hercules-ci.flake-update.baseMerge.enable

Whether to merge the base branch into the update branch before running the update.

This is useful to ensure that the update branch is up to date with the base branch.

If this option is `false`, you may have to merge or rebase the update branch manually sometimes.

Type: boolean

Default: `false`

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)

hercules-ci.flake-update.baseMerge.branch

Branch name on the remote to merge into the update branch before running the update.



Used when `hercules-ci.flake-update.baseMerge.enable` is true.

Type: string

Default: `hercules-ci.flake-update.baseBranch`

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)

hercules-ci.flake-update.baseMerge.method

How to merge the base branch into the update branch before running the update.

Used when `hercules-ci.flake-update.baseMerge.enable` is true.

Type: one of "merge", "rebase"

Default: "merge"

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)

hercules-ci.flake-update.createPullRequest

Whether to create a pull request for the updated `flake.lock`.

Type: boolean

Default: `true`

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)

hercules-ci.flake-update.effect.settings

A module that extends the flake-update effect arbitrarily.



See also:

- [Effect Modules / Core Options](#)
- [Effect Modules / Git Options](#)

Type: module

Default: `{ }`

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)

hercules-ci.flake-update.effect.system

The `system` on which to run the flake update job.

Type: string

Default: config.defaultEffectSystem

Example: "aarch64-linux"

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)

hercules-ci.flake-update.flakes

Flakes to update.

The attribute names refer to the relative paths where the flakes/subflakes are located in the repository.

The values specify further details about how to update the lock. See the sub-options for details.

NOTE: If you provide a definition for this option, it does *not* extend the default. You must specify all flakes you want to update, including the project root (".") if applicable.

 *Type:* attribute set of (submodule) 

Default:

```
{  
    "." = { };  
}
```

Example:

```
{  
    "." = {  
        commitSummary = "/flake.lock: Update";  
    };  
    "path/to/subflake" = {  
        inputs = [  
            "nixpkgs"  
        ];  
    };  
}
```

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)

hercules-ci.flake-update.flakes. <name>.commitSummary

Summary for commit. "" means to use the default.

Type: string

Default: ""

Example: "chore: update flake inputs"

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)

hercules-ci.flake-update.flakes. <name>.inputs >

Flake inputs to update. The default, [] means to update all inputs.

Type: list of string

Default: []

Example: "[\"nixpkgs\" \"nixpkgs-unstable\"]"

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)

hercules-ci.flake-update.forgeType

The type of Git server committed to.

Type: string

Default: "github"

Example: "gitlab"

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](https://github.com/nix-community/hercules-ci-effects/blob/main/effects/flake-update/flake-module.nix)

hercules-ci.flake-update.nix.package

The Nix package to use for performing the lockfile updates.

The function arguments are the module arguments of `perSystem` for `hercules-ci.flake-update.effect.system`.

Type: function that evaluates to a(n) package

Default: { pkgs, ... }: pkgs.nix

Declared by:

 [hercules-ci-effects/effects/flake-update/flake-module.nix](https://github.com/nix-community/hercules-ci-effects/blob/main/effects/flake-update/flake-module.nix) 

hercules-ci.flake-update.pullRequestBody

The body of the pull request being made

Type: null or string

Default:

```
''  
  Update `flake.lock`. See the commit message(s) for details.
```

You may reset this branch by deleting it and re-running the update job.

```
git push origin :flake-update
```



Example: "updated flake.lock"

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)

hercules-ci.flake-update.pullRequestTitle

The title of the pull request being made

Type: string

Default: ```flake.lock`': Update``

Example: "chore: update flake.lock"

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)



hercules-ci.flake-update.updateBranch

To which branch to push the updated flake lock .

Type: string

Default: "flake-update"

Example: "update"

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](#)

hercules-ci.flake-update.when

See `herculesCI.onSchedule.<name>.when` for details.

Type: raw value

Declared by:

[hercules-ci-effects/effects/flake-update/flake-module.nix](https://github.com/nix-community/hercules-ci-effects/blob/main/effects/flake-update/flake-module.nix)

hercules-ci.github-pages.branch

A GitHub Pages deployment will be triggered when changes are pushed to this branch.

A non-null value enables the effect.

Type: null or string

Default: null

Declared by:

[hercules-ci-effects/flake-modules/github-pages.nix](https://github.com/nix-community/hercules-ci-effects/blob/main/flake-modules/github-pages.nix)



hercules-ci.github-pages.check.enable

Whether to make sure that the effect is buildable. This adds `checks.${config.defaultEffectSystem}.gh-pages` to `onPush.default`.

Type: boolean

Default: true

Declared by:

[hercules-ci-effects/flake-modules/github-pages.nix](https://github.com/nix-community/hercules-ci-effects/blob/main/flake-modules/github-pages.nix)

hercules-ci.github-pages.pushJob

The Hercules CI job in which to perform the deployment.

By default the GitHub pages deployment is triggered by the `onPush.default` job, so that the deployment only proceeds when the default builds are successful.

Type: string

Default: "default"

Declared by:

[hercules-ci-effects/flake-modules/github-pages.nix](#)

hercules-ci.github-pages.settings

Modular settings for the GitHub Pages effect.

For system-dependent settings, define `perSystem.hercules-ci.github-pages.settings` instead.

See `gitWriteBranch` for options.

Type: module



Example:

```
{  
  message = "Update GitHub Pages";  
}
```

Declared by:

[hercules-ci-effects/flake-modules/github-pages.nix](#)

hercules-ci.github-releases.checkArtifacts

Condition under which to check whether artifacts can be built.

Type: function that evaluates to a(n) boolean

Default: _: true

Declared by:

[hercules-ci-effects/flake-modules/github-releases](#)

hercules-ci.github-releases.condition

Condition under which a release is going to be pushed. This is a function accepting [HerculesCI parameters](#) and returning boolean. By default, pushing happens if a tag is present.

Type: function that evaluates to a(n) boolean

Default:

```
{ tag, ... }: tag != null
```

Declared by:

[hercules-ci-effects/flake-modules/github-releases](#)

hercules-ci.github-releases.files

List of asset files or archives.

Each entry must be either an attribute set of type

- `{ label: string, path: string }` for a single file, or
- `{ label: string, paths: [string], archiver: 'zip' }` for an archive.

In case of archive, `paths` may contain directories: their *contents* will be archived recursively.

Type: list of (submodule)

Default: []

Example:

```
[  
 {  
   label = "api.json";  
   path = withSystem "x86_64-linux" ({config, ...}: config.packages.api-json);  
 }]
```

```
    }
    {
        label = "api-docs.zip";
        paths = withSystem "x86_64-linux" ({config, ...}: [ config.packages.api-docs
            archiver = "zip";
        ]
    }
```

Declared by:

[hercules-ci-effects/flake-modules/github-releases](https://github.com/herculesci/hercules-ci-effects/tree/main/flake-modules/github-releases)

hercules-ci.github-releases.files.*.archiver

The archiver to use for the archive.

This must be set when `paths` is set.

Type: value “zip” (singular enum)

Default: (unset)

 *Declared by:* 

[hercules-ci-effects/flake-modules/github-releases](https://github.com/herculesci/hercules-ci-effects/tree/main/flake-modules/github-releases)

hercules-ci.github-releases.files.*.label

Label of the asset file or archive.

This is the name that will be used in the GitHub release.

Type: string

Declared by:

[hercules-ci-effects/flake-modules/github-releases](https://github.com/herculesci/hercules-ci-effects/tree/main/flake-modules/github-releases)

hercules-ci.github-releases.files.*.path

Path to the asset file. Must not be a directory. Mutually exclusive with `paths`.

Type: path

Declared by:

[hercules-ci-effects/flake-modules/github-releases](#)

hercules-ci.github-releases.files.*.paths

Paths to the asset files. Mutually exclusive with `path`.

Directories are allowed, and their contents will be archived recursively.

Type: list of path

Declared by:

[hercules-ci-effects/flake-modules/github-releases](#)



hercules-ci.github-releases.filesPerSystem

List of asset files or archives for each system.

The arguments passed are the same as those passed to `perSystem` modules.

The function is invoked for each of the `systems`. The returned labels must be unique across invocations. This generally means that you have to include the `system` value in the attribute names.

NOTE: If you are implementing generic logic, consider placing the function in a `mkIf`, so that the function remains undefined in cases where it is statically known to produce no files. When `filesPerSystem` has no definitions, a traversal of potentially many `perSystems` modules is avoided.

Type: function that evaluates to a(n) list of (submodule)

Example:

```
{ system, config, ... }: [  
  {
```

```
    label = "foo-static-${system}";
    path = lib.getExe config.packages.foo-static;
}
]
```

Declared by:

[hercules-ci-effects/flake-modules/github-releases](#)

hercules-ci.github-releases.filesPerSystem. <function body>.*.archiver

The archiver to use for the archive.

This must be set when `paths` is set.

Type: value “zip” (singular enum)

Default: (*unset*)

Declared by:



[hercules-ci-effects/flake-modules/github-releases](#)

hercules-ci.github-releases.filesPerSystem. <function body>.*.label

Label of the asset file or archive.

This is the name that will be used in the GitHub release.

Type: string

Declared by:

[hercules-ci-effects/flake-modules/github-releases](#)

hercules-ci.github-releases.filesPerSystem.

<function body>.*.path

Path to the asset file. Must not be a directory. Mutually exclusive with `paths`.

Type: path

Declared by:

[hercules-ci-effects/flake-modules/github-releases](#)

hercules-ci.github-releases.filesPerSystem.

<function body>.*.paths

Paths to the asset files. Mutually exclusive with `path`.

Directories are allowed, and their contents will be archived recursively.

Type: list of path

Declared by:

[hercules-ci-effects/flake-modules/github-releases](#)



hercules-ci.github-releases.pushJobName

Name of the Hercules CI job in which to perform the deployment. By default the GitHub pages deployment is triggered by the `onPush.default` job, so that the deployment only proceeds when the default builds are successful.

Type: string

Default: `default`

Declared by:

[hercules-ci-effects/flake-modules/github-releases](#)

hercules-ci.github-releases.releaseTag

Tag to be assigned to the release.

Type: function that evaluates to a(n) string

Default: `herculesCI: herculesCI.config.repo.tag`

Declared by:

[hercules-ci-effects/flake-modules/github-releases](#)

hercules-ci.github-releases.systems

List of systems for which to call `filesPerSystem`.

Type: null or (list of string)

Default: `null`, which means that `herculesCI.ciSystems` will be used.

Declared by:



[hercules-ci-effects/flake-modules/github-releases](#)

herculesCI

Hercules CI environment and configuration. See the sub-options for details.

This module represents a function. Hercules CI calls this function to provide expressions in the flake with extra information, such as repository and job metadata.

While this attribute feels a lot like a submodule, it can not be queried by definitions outside of `herculesCI`. This is required by the design of flakes: evaluation of the standard flake attribute values is hermetic.

Data that is unique to Hercules CI (as opposed to the flake itself) is provided by in the sub-options of `herculesCI`. This is syntactically different from the [native herculesCI attribute interface](#). For example, instead of `{ primaryRepo, ... }: ... primaryRepo.ref`, you would write `{ config, ... }: ... config.repo.ref`.

See e.g. [ref](#).

Type: module

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-helpers.nix](#), [hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.ciSystems

Flake systems for which to generate attributes in `herculesCI.onPush.default.outputs`.

Type: list of string

Default: `config.systems # from flake parts`

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)



herculesCI.onPush

This declares what to do when a Git ref is updated, such as when you push a commit or after you merge a pull request.

By default `onPush.default` defines a job that builds the known flake output attributes. It can be disabled by setting `onPush.default.enable = false;`.

The name of the job (from `onPush.<name>`) will be used as part of the commit status of the resulting job.

Type: lazy attribute set of (submodule)

Default: { }

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.onPush.<name>.outputs

A collection of builds and effects. These may be nested recursively into attribute sets.

Hercules CI's traversal of nested sets can be cancelled with `lib.dontRecurseIntoAttrs`.

See the parent option for details about when the job runs.

Type: a tree of attribute sets and derivations

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.onSchedule

Since hercules-ci-agent 0.9.8

Behaves similar to `onPush`, but is responsible for jobs that respond to the passing of time rather than to a git push or equivalent.



Type: lazy attribute set of (submodule)

Default: { }

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.onSchedule.<name>.outputs

A collection of builds and effects. These may be nested recursively into attribute sets.

Hercules CI's traversal of nested sets can be cancelled with `lib.dontRecurseIntoAttrs`.

See the parent option for details about when the job runs.

Type: a tree of attribute sets and derivations

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.onSchedule.<name>.when

The time at which to schedule a job.

Each subattribute represents an equality, all of which will hold at the next planned time.
The time zone is UTC.

The `minute` or `hour` attributes can be omitted, in which case Hercules CI will pick an arbitrary time for you.

See the `when.*` options below for details.

Type: submodule

Default: { }

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)



herculesCI.onSchedule.<name>.when.dayOfMonth

An optional list of day of the month during which to create a job.

The default value `null` represents all days.

Type: null or ((list of integer between 0 and 31 (both inclusive)) or integer between 0 and 31 (both inclusive) convertible to it)

Default: null

Declared by:

[hercules-ci-effects/flake-modules/types/when.nix](#)

herculesCI.onSchedule.<name>.when.dayOfWeek

An optional list of week days during which to create a job.

The default value `null` represents all days.

Type: null or ((list of (one of "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")) or (one of "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun") convertible to it)

Default: `null`

Declared by:

[hercules-ci-effects/flake-modules/types/when.nix](https://github.com/herculesci/hercules-ci-effects/blob/main/flake-modules/types/when.nix)

herculesCI.onSchedule.<name>.when.hour

An optional integer representing the hours at which a job should be created.

The default value `null` represents an arbitrary hour.

Type: null or ((list of integer between 0 and 23 (both inclusive)) or integer between 0 and 23 (both inclusive) convertible to it)

Default: `null`

Declared by:

[hercules-ci-effects/flake-modules/types/when.nix](https://github.com/herculesci/hercules-ci-effects/blob/main/flake-modules/types/when.nix)

herculesCI.onSchedule.<name>.when.minute

An optional integer representing the minute mark at which a job should be created.

The default value `null` represents an arbitrary minute.

Type: null or integer between 0 and 59 (both inclusive)

Default: `null`

Declared by:

[hercules-ci-effects/flake-modules/types/when.nix](https://github.com/herculesci/hercules-ci-effects/blob/main/flake-modules/types/when.nix)

herculesCI.repo

The repository and checkout metadata of the current checkout, provided by Hercules CI. These options are read-only.

You may read options by querying the `config` module argument.

Type: submodule (*read only*)

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](https://github.com/herculesci/hercules-ci-effects/blob/main/flake-modules/herculesCI-attribute.nix)

herculesCI.repo.branch

The branch of the checkout. `null` when not on a branch; e.g. when on a tag.

Type: null or string (*read only*)

Example: "main"



Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](https://github.com/herculesci/hercules-ci-effects/blob/main/flake-modules/herculesCI-attribute.nix)

herculesCI.repo.forgeType

What forge implementation hosts the repository.

E.g. "github" or "gitlab"

Since hercules-ci-agent 0.9.8

Type: string (*read only*)

Default:

Example: "github"

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.repo.name

The name of the repository.

Since hercules-ci-agent 0.9.8

Type: string (*read only*)

Default:

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.repo.owner

The owner of the repository.



Since hercules-ci-agent 0.9.8

Type: string (*read only*)

Default:

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.repo.ref

The git “ref” of the checkout.

Type: string (*read only*)

Example: "refs/heads/main"

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.repo.remoteHttpUrl

HTTP url for cloning the repository.

Since hercules-ci-agent 0.9.8

Type: string (*read only*)

Default:

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.repo.remoteSshUrl

SS~~K~~url for cloning the repository. >

Since hercules-ci-agent 0.9.8

Type: string (*read only*)

Default:

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.repo.rev

The git revision, also known as the commit hash.

Type: string (*read only*)

Example: "17ae1f614017447a983c34bb046892b3c571df52"

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

herculesCI.repo.shortRev

An abbreviated `rev`.

Type: string (*read only*)

Example: "17ae1f6"

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)



herculesCI.repo.tag

The tag of the checkout. `null` when not on a tag; e.g. when on a branch.

Type: null or string (*read only*)

Example: "1.0"

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)



herculesCI.repo.webUrl

A URL to open the repository in the browser.

Since hercules-ci-agent 0.9.8

Type: string (*read only*)

Default:

Declared by:

[hercules-ci-effects/flake-modules/herculesCI-attribute.nix](#)

perSystem.hercules-ci.github-pages.settings

Modular settings for the GitHub Pages effect.

See `gitWriteBranch` for options.

Type: module

Example:

```
{  
    contents = config.packages.docs + "/share/doc/mypkg/html";  
}
```

Declared by:

[hercules-ci-effects/flake-modules/github-pages.nix](#)

perSystem.herculesCIEffects.pkgs

Nixpkgs instance to use for `hercules-ci-effects`.

The effects functions, etc, will be provided as the `effects` module argument of `perSystem`.

Type: raw value

Default: `pkgs` (module argument)

Declared by:

[hercules-ci-effects/flake-modules/module-argument.nix](#)

Flake Parts

Core of a distributed framework for writing Nix Flakes.

`flake-parts` provides the options that represent standard flake attributes and establishes a way of working with `system`. Opinionated features are provided by an ecosystem of modules that you can import.

`flake-parts` *itself* has the goal to be a minimal mirror of the Nix flake schema. Used by itself, it is very lightweight.

Documentation: [flake.parts](#)

Why Modules?

Flakes are configuration. The module system lets you refactor configuration into modules that can be shared.

It reduces the proliferation of custom Nix glue code, similar to what the module system has done for NixOS configurations.

Unlike NixOS, but following Flakes' spirit, `flake-parts` is not a monorepo with the implied goal of absorbing all of open source, but rather a single module that other repositories can build upon, while ensuring a baseline level of compatibility: the core attributes that constitute a flake.

Features

- Split your `flake.nix` into focused units, each in their own file.
- Take care of `system`.
- Allow users of your library flake to easily integrate your generated flake outputs into their flake.
- Reuse project logic written by others

This documentation

You can find guides and the options reference in the menu (top left).

A site wide search is available by typing `s`.



mission-control

A flake-parts module for your Nix devshell scripts.

Lets you configure commands that will be run in the repository root.

Provides an informative "message of the day" when launching your shell.

See the [Platonic-Systems/mission-control readme](#).

Installation

To use these options, add to your flake inputs:

```
mission-control.url = "github:Platonic-Systems/mission-control";
```

and inside the `mkFlake`:



```
imports = [  
    inputs.mission-control.flakeModule  
];
```

Run `nix flake lock` and you're set.

Options

`perSystem.mission-control`

`perSystem.mission-control.banner`

`perSystem.mission-control.devShell`

`perSystem.mission-control.scripts`

`perSystem.mission-control.scripts.<name>.category`

```
perSystem.mission-control.scripts.<name>.cdToProjectRoot
```

```
perSystem.mission-control.scripts.<name>.description
```

```
perSystem.mission-control.scripts.<name>.exec
```

```
perSystem.mission-control.wrapper
```

```
perSystem.mission-control.wrapperName
```

perSystem.mission-control

Specification for the scripts in dev shell

Type: submodule

Default: { }

Declared by:

```
mission-control/nix/flake-module.nix
```



perSystem.mission-control.banner

The generated shell banner.

Type: string

Default: generated package

Declared by:

```
mission-control/nix/flake-module.nix
```

perSystem.mission-control.devShell

A devShell containing the banner and wrapper.

Type: package (*read only*)

Declared by:

`mission-control/nix/flake-module.nix`

perSystem.mission-control.scripts

List of scripts to be added to the shell

Type: attribute set of (submodule)

Default: `{ }`

Declared by:

`mission-control/nix/flake-module.nix`

perSystem.mission-control.scripts.

<`name`>.category



The category under which this script will be grouped.

Type: string

Default: "Commands"

Declared by:

`mission-control/nix/flake-module.nix`

perSystem.mission-control.scripts.

<`name`>.cdToProjectRoot

Whether to change the working directory to the project root before running the script.

Type: boolean

Default: `true`

Declared by:

[mission-control/nix/flake-module.nix](#)

perSystem.mission-control.scripts.<name>.description

A description of what this script does.

This will be displayed in the banner and help menu.

Type: null or string

Default: null

Declared by:

[mission-control/nix/flake-module.nix](#)

<

>

perSystem.mission-control.scripts.<name>.exec

The script or package to run

The \$FLAKE_ROOT environment variable will be set to the project root, as determined by the github:srid/flake-root module.

Type: string or package

Declared by:

[mission-control/nix/flake-module.nix](#)

perSystem.mission-control.wrapper

The generated wrapper script.

Type: package

Default: generated package

Declared by:

[mission-control/nix/flake-module.nix](#)

perSystem.mission-control.wrapperName

The name of the wrapper script

Type: string

Default: ","

Declared by:

[mission-control/nix/flake-module.nix](#)



**AUG 31, 2023****BY EELCO DOLSTRA**

Flake schemas: making flake outputs extensible

[Flakes](#) are a generic way to package [Nix](#) artifacts. Flake output attributes are arbitrary Nix values, so they can be [packages](#), [NixOS modules](#), [CI jobs](#), and so on. While there are a number of “well-known” flake output types that are recognized by tools like the [nix CLI](#)—[nix develop](#), for example, operates on the `devShells` output—nothing prevents you from defining your own flake output types.



Unfortunately, such “non-standard” [flake output](#) types have a big

problem: tools like `nix flake show` and `nix flake check` don't know anything about them, so they can't display or check anything about those outputs. The `nixpkgs` flake, for instance, has a `lib` output that Nix knows nothing about:

```
# nix flake show nixpkgs
github:NixOS/nixpkgs/4ecab3273592f27479a583fb(
  ...
  └─lib: unknown
```

This was a problem when we were creating [FlakeHub](#): the FlakeHub web interface should be able to display the contents of a flake, including documentation, but we want to do so in an extensible way.

Today we're proposing a solution to this problem: **flake schemas**. Flake schemas enable flakes to declare functions that enumerate and check the contents of their outputs. Schemas themselves are defined as a flake output named `schemas`. Tools like `nix flake check` and FlakeHub can then use these schemas to display or check the contents of flakes in a generic way.

In this approach, flakes carry their own schema definitions, so you are not dependent on some central registry of schema definitions—you define what your flake outputs are supposed to look like.

Here is an example of what the outputs of a flake, extracted using that flake's schemas, look like in FlakeHub:

| Systems | x86_64-linux | x86_64-darwin | i686-linux | aarch64-darwin | aarch64-linux |
|---|--------------|---------------|------------|----------------|---------------|
| packages | | | | | |
| The packages flake output contains packages that can be added to a shell using <code>nix shell</code> . | | | | | |
| aarch64-darwin | | | | | |



macOS on Apple Silicon, like the M1 or M2 chips.

```
packages.aarch64-darwin.default
packages.aarch64-darwin.nix
```

Show 7 more

Show 3 more systems

overlays

The overlays flake output defines "overlays" that can be plugged into Nixpkgs. Overlays add additional packages or modify or replace existing packages.

Example: Using the overlays.default overlay

All operating systems and architectures.

```
overlays.default
● ● ● flake.nix
{
  inputs = {
    nixpkgs.url = "https://flakehub.com/f/NixOS/nixpkgs/*.tar.gz";
    nix.url = "https://flakehub.com/f/NixOS/nix/2.17.0.tar.gz";
  };
  outputs = { self, nixpkgs, nix }: let
    system = "aarch64-darwin";
    pkgs = import nixpkgs {
      inherit system;
      overlays = [
        nix.overlays.default
      ];
    };
    in
    {
      # `pkgs` is nixpkgs for the system, with nix's overlay applied.
    };
}
```

checks

The checks flake output contains derivations that will be built by `nix flake check`.

devShells

The devShells flake output contains derivations that provide a build environment for `nix develop`.

hydraJobs

The hydraJobs flake output defines derivations to be built by the Hydra continuous integration system.

FlakeHub showing flake outputs extracted via flake schemas

Using flake schemas

While you can define your own schema definition (see below), usually you would use schema definitions provided by others. We provide a repository named [flake-schemas](#) with schemas for the most widely used flake outputs (the ones for which Nix has built-in support).



Declaring what schemas to use is easy: you just define a `schemas` output.

```
{  
    # `flake-schemas` is a flake that provides:  
    # like `packages` and `devShells`.  
    inputs.flake-schemas.url = github:Determinan-  
  
    # Another flake that provides schemas.  
    inputs.other-schemas.url = ...;  
  
    outputs = { self, flake-schemas, other-sche-  
        # Tell Nix what schemas to use.  
        schemas = flake-schemas.schemas // other-  
  
        # These flake outputs will now be checked  
        packages = ...;  
        devShells = ...;  
    };  
}
```

Defining your own schemas



With schemas, we can now teach Nix about the `lib` output mentioned previously. Below is a flake that has a `lib` output. Similar to `lib` in Nixpkgs, it has a nested structure (e.g. it provides a function `lists.singleton`). The flake also has a `schemas.lib` attribute that tells Nix two things:

1. How to list the contents of the `lib` output.
2. To check that every function name follows the camelCase naming convention.

```
{  
  outputs = { self }: {  
  
    schemas.lib = {  
      version = 1;  
      doc = ''  
        The `lib` flake output defines Nix fu  
      '';  
      inventory = output:  
        let  
          recurse = attrs: {  
            children = builtins.mapAttrs (attr  
              if builtins.isFunction attr  
              then  
                {  
                  # Tell `nix flake show` what  
                  what = "library function";  
                  # Make `nix flake check` en  
                  evalChecks.camelCase = buil  
                }  
              else if builtins.isAttrs attr  
              then  
                # Recurse into nested sets of  
                recurse attr  
              
```



```
        else
            throw "unsupported 'lib' type"
        attrs;
    };
    in recurse output;
};

lib.id = x: x;
lib.const = x: y: x;
lib.lists.singleton = x: [x];
#lib.ConcatStrings = ...; # disallowed
};
}
```

With this schema, `nix flake show` can now show information about `lib`:

```
# nix flake show
git+file:///home/eelco/Determinate/flake-scher
└─lib
    ├─const: library function
    ├─id: library function
    └─lists
        └─singleton: library function
```



While `nix flake check` will now complain if we add a function that violates the naming convention we defined:

```
# nix flake check  
warning: Evaluation check 'camelCase' of flake
```

For more information on how to write schema definitions, see [the Nix documentation](#).

Find out more about how Determinate Systems is transforming the developer experience with Nix flakes

Subscribe

What schemas are not

Flake schemas are not a type system for Nix, since that would be a huge project. They merely provide an *interface* that enables users to tell Nix how to enumerate and check flake outputs. For instance, for a [NixOS](#) configuration, this means using the module system to check that the configuration evaluates correctly; for a Nix package, it just means that the output attribute evaluates to a [derivation](#).



Next steps

Flake schemas are new, and they're a valuable expansion of the user experience of [FlakeHub](#) and [Nix flakes](#) in general. We believe that

incorporating schemas into Nix itself will make flakes more broadly valuable and cover use cases that we haven't yet imagined. We're looking for input from the community to see whether the current schema design covers all use cases. We've submitted a [pull request](#) to the [Nix project](#) that adds schema support to [nix flake show](#) and [nix flake check](#). Please take a look!

As future work, schemas will enable us to finally make flakes *configurable* in a discoverable way: flake schemas can return the configuration options supported by a flake output—for the `nixosConfigurations` output, for example, these would be all the NixOS options—and then the [Nix CLI](#) can enable users to override options from the command line.

Conclusion

SHARE  



WRITTEN BY

Eelco Dolstra

Eelco started the Nix project as a PhD student at Utrecht University. He serves as president of the NixOS Foundation and is a member of the Nix team.



Get the latest updates

Subscribe

hello@determinate.systems

+1 (641) NIX-HELP (649-4357)

© 2021-2024 Determinate Systems. All rights reserved.

[Terms of service](#) [Privacy](#) [DMCA](#) [Code of conduct](#) [Security](#)



Flakes

What is usually referred to as “flakes” is:

- A policy for managing dependencies between [Nix expressions](#).
- An [experimental feature](#) in Nix, implementing that policy and supporting functionality.

Technically, a [flake](#) is a file system tree that contains a file named `flake.nix` in its root directory.

Flakes add the following behavior to Nix:

1. A `flake.nix` file offers a uniform [schema](#), where:
 - Other flakes can be referenced as dependencies providing [Nix language code](#) or other files.
 - The values produced by the [Nix expressions](#) in `flake.nix` are structured according to pre-defined use cases.
2. References to other flakes can be specified using a dedicated [URL-like syntax](#). A [flake registry](#) allows using symbolic identifiers for further brevity. References can be automatically locked to their current specific version and later updated programmatically.
3. A [new command line interface](#), implemented as a separate experimental feature, leverages flakes by accepting flake references in order to build, run, or deploy software defined as a flake.

Nix handles flakes differently than regular [Nix files](#) in the following ways:

- The `flake.nix` file is checked for schema validity.
In particular, the metadata fields cannot be arbitrary Nix expressions. This is to prevent complex, possibly non-terminating computations while querying the metadata.
- The entire flake directory is copied to Nix store before evaluation.
This allows for effective evaluation caching, which is relevant for large expressions such as Nixpkgs, but also requires copying the entire flake directory again on each change.

- No external variables, parameters, or impure language values are allowed.

It means full reproducibility of a Nix expression, and, by extension, the resulting build instructions by default, but also prohibits parameterisation of results by consumers.

Flakes

From NixOS Wiki

Nix flakes is an experimental feature (<https://nixos.org/manual/nix/stable/contributing/experimental-features.html>) of the Nix package manager. Flakes was introduced with Nix 2.4 (see release notes (<https://nixos.org/manual/nix/unstable/release-notes/r1-2.4.html>)).

Introduction

Flakes is a feature of managing Nix packages to simplify usability and improve reproducibility of Nix installations. Flakes manages dependencies between Nix expressions, which are the primary protocols for specifying packages. Flakes implements these protocols in a consistent schema with a common set of policies for managing packages.

- 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`.
- An installation may contain any number of flakes, independent of each other or even call each other.
- The contents of `flake.nix` file follow the uniform naming schema for expressing packages and dependencies on Nix.
- Flakes use the standard Nix protocols, including the URL-like syntax (<https://nixos.org/manual/nix/stable/command-ref/new-cli/nix3-flake.html#flake-references>) for specifying repositories and package names.
- 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 easily queried and updated programmatically.
- Nix 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 system configuration (https://nixos.wiki/wiki/Overview_of_the_NixOS_Linux_distribution#Declarative_Configuration) (flakes (https://nixos.wiki/wiki/Flakes#Using_nix_flakes_with_NixOS)):

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

Other Distros, with Home-Manager

Add the following to your home-manager config:

```
nix = {
  package = pkgs.nix;
  settings.experimental-features = [ "nix-command" "flakes" ];
};
```

Other Distros, without Home-Manager

Note: The Nix Determinate 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.

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/> (<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>).

Output schema

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.

This is described in the nix package manager src/nix/flake.cc (<https://github.com/NixOS/nix/blob/master/src/nix/flake.cc>) in CmdFlakeCheck.

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/12c64ca55c1014cdc1b16ed5a80
    sha256 = "0jm6nzb83wa6ai17ly9fpq40wg1viib8klq8lby54agpl213w5"; }
) {
  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);
  in fetchTarball {
    url = "https://github.com/edolstra/flake-compat/archive/${lock.nodes.flake-compat.l
    sha256 = lock.nodes.flake-compat.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:/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 {
    # Note that you cannot put arbitrary configuration here: the configuration must be
    system = "x86_64-linux";
    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:

```
{
  outputs = { self, nixpkgs }: {
    # replace 'joes-desktop' with your hostname here.
    nixosConfigurations.joes-desktop = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      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, ... }@attrs: {
    nixosConfigurations.fnord = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      specialArgs = attrs;
      modules = [ ./configuration.nix ];
    };
  };
}
```

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

```
{ config, lib, nixpkgs, home-manager, ... }: {
  # do something with home-manager here, for instance:
  imports = [ 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 current 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 configuration, 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 to the system pkgs on NixOS

```
nix.registry = {
  nixpkgs.to = {
    type = "path";
    path = pkgs.path;
  };
};
```

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> { } }:
with pkgs;
mkShell {
  buildInputs = [
    nixpkgs-fmt
  ];

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

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

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

```
{
  description = "my project description";

  inputs.flake-utils.url = "github:numtide/flake-utils";

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem
      (system:
        let pkgs = nixpkgs.legacyPackages.${system}; in
        {
          devShells.default = import ./shell.nix { inherit pkgs; };
        }
      );
}
```

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.

Direnv integration

Assuming that the flake defines a `devShell` output attribute and that you are using direnv. Here is how to replace the old `use nix stdlib` function with the faster flake version:

```
use_flake() {
    watch_file flake.nix
    watch_file flake.lock
    eval "$(nix print-dev-env --profile "$(direnv_layout_dir)/flake-profile")"
}
```

Copy this in `~/.config/direnv/lib/use_flake.sh` or in `~/.config/direnv/direnvrc` or directly in your project specific `.envrc`.

Note: You may not need to create `use_flake()` yourself; as of direnv 2.29, (<https://github.com/direnv/direnv/releases/tag/v2.29.0#:~:text=add%20use%20function>) `use flake` is part of direnv's standard library.

With this in place, you can now replace the `use nix` invocation in the `.envrc` file with `use flake`:

```
# .envrc
use flake
```

The nice thing about this approach is that evaluation is cached.

Optimize the reloads

Nix Flakes has a Nix evaluation caching mechanism. Is it possible to expose that somehow to automatically trigger direnv reloads?

With the previous solution, direnv would only reload if the `flake.nix` or `flake.lock` files have changed. This is not completely precise as the `flake.nix` file might import other files in the repository.

Setting the bash prompt like nix-shell

A new experimental feature of flakes (<https://github.com/NixOS/nix/pull/4189>) allow to setup a bash-prompt per flake:

```
{
  description = "...";
  nixConfig.bash-prompt = "\[nix-develop\]$ ";
  ...
}
```

Otherwise it's also possible to set the `nix develop` bash prompt system wide using the `nix.conf` option `bash-prompt` (<https://nixos.org/manual/nix/unstable/command-ref/conf-file.html>). (On nixos I think it is set in

```
nix.extraOptions )
```

Pushing Flakes to Cachix

<https://docs.cachix.org/pushing#flakes> (<https://docs.cachix.org/pushing#flakes>)

To push *all* flake outputs automatically, use `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> (<https://github.com/NixOS/nix/pull/5434>)

Importing packages from multiple channels

A NixOS config flake skeleton could be as follows:

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

  inputs = {
    nixpkgs.url = "nixpkgs/nixos-21.11";
    nixpkgs-unstable.url = "nixpkgs/nixos-unstable";
  };

  outputs = { self, nixpkgs, nixpkgs-unstable }:
  let
    system = "x86_64-linux";
    overlay-unstable = final: prev: {
      unstable = nixpkgs-unstable.legacyPackages.${prev.system};
      # use this variant if unfree packages are needed:
      # unstable = import nixpkgs-unstable {
      #   inherit system;
      #   config.allowUnfree = true;
      # };
    };
  in {
    nixosConfigurations."<hostname>" = nixpkgs.lib.nixosSystem {
      inherit system;
      modules = [
        # Overlays-module makes "pkgs.unstable" available in configuration.nix
        ({ config, pkgs, ... }: { nixpkgs.overlays = [ overlay-unstable ]; })
        ./configuration.nix
      ];
    };
  };
}
```

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

Same can be done with the NURs, as it already has an `overlay` attribute in the `flake.nix` of the project, you can just add

```
nixpkgs.overlays = [ nur.overlay ];
```

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

```
pkgs = import nixpkgs { overlays = [ /*the overlay in question*/ ]; };
```

Getting Instant System Flakes Repl

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

```
# nix repl
>> :lf /etc/nixos
>> nixosConfigurations.myhost.config
{ ... }
```

Or out of your current flake:

```
# nix repl
>> :lf .#
```

You can then access to the inputs, outputs... For instance if you would like to check the default version of the kernel present in nixpkgs:

```
nix-repl> inputs.nixpkgs.legacyPackages.x86_64-linux.linuxPackages.kernel.version
"5.15.74"
```

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.n
```

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/.*)|$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 (/wiki/Unfree_Software).

Development tricks

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 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

- 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-nixos-flakes/>) (Eelco Dolstra, 2020)
- NixOS & Flakes Book (<https://github.com/ryan4yin/nixos-and-flakes-book>)(Ryan4yin, 2023) - 🔨 ❤️ An unofficial NixOS & Flakes book for beginners.
- 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: 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 101: Introduction to nix flakes (https://www.youtube.com/watch?v=QXUlhnhuRX4&list=PLgknCdxP89RcGPTjngfNR9WmBgvD_xW0I) (Jörg Thalheim, 2020)
- RFC 49 (<https://github.com/NixOS/rfcs/pull/49>) (2019) - Original flakes specification
- 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>)

Retrieved from "<https://nixos.wiki/index.php?title=Flakes&oldid=11071>" (<https://nixos.wiki/index.php?title=Flakes&oldid=11071>)

Categories (/wiki/Special:Categories):

Pages with syntax highlighting errors (/index.php?title=Category:Pages_with_syntax_highlighting_errors&action=edit&redlink=1)

| Software (/wiki/Category:Software) | Nix (/wiki/Category:Nix)

| Flakes (/index.php?title=Category:Flakes&action=edit&redlink=1)

Flakes

From NixOS Wiki

Nix flakes is an experimental feature (<https://nixos.org/manual/nix/stable/contributing/experimental-features.html>) of the Nix package manager. Flakes was introduced with Nix 2.4 (see release notes (<https://nixos.org/manual/nix/unstable/release-notes/r1-2.4.html>)).

Introduction

Flakes is a feature of managing Nix packages to simplify usability and improve reproducibility of Nix installations. Flakes manages dependencies between Nix expressions, which are the primary protocols for specifying packages. Flakes implements these protocols in a consistent schema with a common set of policies for managing packages.

- 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`.
- An installation may contain any number of flakes, independent of each other or even call each other.
- The contents of `flake.nix` file follow the uniform naming schema for expressing packages and dependencies on Nix.
- Flakes use the standard Nix protocols, including the URL-like syntax (<https://nixos.org/manual/nix/stable/command-ref/new-cli/nix3-flake.html#flake-references>) for specifying repositories and package names.
- 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 easily queried and updated programmatically.
- Nix 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 system configuration (<https://nixos.wiki/wik...> /Overview_of_the_NixOS_Linux_distribution#Declarative_Configuration) (flakes (<https://nixos.wiki/wik...> /Flakes#Using_nix_flakes_with_NixOS)):

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



Other Distros, with Home-Manager

Add the following to your home-manager config:

```
nix = {  
    package = pkgs.nix;  
    settings.experimental-features = [ "nix-command" "flakes" ];  
};
```

Other Distros, without Home-Manager

Note: The Nix Determinate 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.

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/> (<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>).

Output schema

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.

This is described in the nix package manager src/nix/flake.cc (<https://github.com/NixOS/nix/blob/master/src/nix/flake.cc>) in CmdFlakeCheck.

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/12c64ca55c1014cdc1b16ed5a80
    sha256 = "0jm6nzb83wa6ai17ly9fpq40wg1viib8klq8lby54agpl213w5"; }
) {
  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);
  in fetchTarball {
    url = "https://github.com/edolstra/flake-compat/archive/${lock.nodes.flake-compat.l
    sha256 = lock.nodes.flake-compat.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:/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 {
    # Note that you cannot put arbitrary configuration here: the configuration must be
    system = "x86_64-linux";
    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:



```
{
  outputs = { self, nixpkgs }: {
    # replace 'joes-desktop' with your hostname here.
    nixosConfigurations.joes-desktop = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      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, ... }@attrs: {
    nixosConfigurations.fnord = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      specialArgs = attrs;
      modules = [ ./configuration.nix ];
    };
  };
}
```

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

```
{ config, lib, nixpkgs, home-manager, ... }: {
  # do something with home-manager here, for instance:
  imports = [ 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 current 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 configuration, 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 to the system pkgs on NixOS

```
nix.registry = {
  nixpkgs.to = {
    type = "path";
    path = pkgs.path;
  };
};
```

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> { } }:
with pkgs;
mkShell {
  buildInputs = [
    nixpkgs-fmt
  ];

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

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

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

```
{
  description = "my project description";

  inputs.flake-utils.url = "github:numtide/flake-utils";

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem
      (system:
        let pkgs = nixpkgs.legacyPackages.${system}; in
        {
          devShells.default = import ./shell.nix { inherit pkgs; };
        }
      );
}
```



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.

Direnv integration

Assuming that the flake defines a `devShell` output attribute and that you are using direnv. Here is how to replace the old use `nix stdlib` function with the faster flake version:

```
use_flake() {
    watch_file flake.nix
    watch_file flake.lock
    eval "$(nix print-dev-env --profile "$(direnv_layout_dir)/flake-profile")"
}
```

Copy this in `~/.config/direnv/lib/use_flake.sh` or in `~/.config/direnv/direnvrc` or directly in your project specific `.envrc`.

Note: You may not need to create `use_flake()` yourself; as of direnv 2.29, (<https://github.com/direnv/direnv/releases/tag/v2.29.0#:~:text=add%20use%20flake%20function>) `use flake` is part of direnv's standard library.

With this in place, you can now replace the `use nix` invocation in the `.envrc` file with `use flake`:

```
# .envrc
use flake
```

The nice thing about this approach is that evaluation is cached.

Optimize the reloads

Nix Flakes has a Nix evaluation caching mechanism. Is it possible to expose that somehow to automatically trigger direnv reloads?

With the previous solution, direnv would only reload if the `flake.nix` or `flake.lock` files have changed. This is not completely precise as the `flake.nix` file might import other files in the repository.

Setting the bash prompt like nix-shell

A new experimental feature of flakes (<https://github.com/NixOS/nix/pull/4189>) allow to setup a bash-prompt per flake:

```
{
  description = "...";
  nixConfig.bash-prompt = "\[nix-develop\]$ ";
  ...
}
```

Otherwise it's also possible to set the `nix develop` bash prompt system wide using the `nix.conf` option `bash-prompt` (<https://nixos.org/manual/nix/unstable/command-ref/conf-file.html>). (On nixos I think it is set in



```
nix.extraOptions )
```

Pushing Flakes to Cachix

<https://docs.cachix.org/pushing#flakes> (<https://docs.cachix.org/pushing#flakes>)

To push *all* flake outputs automatically, use `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> (<https://github.com/NixOS/nix/pull/5434>)

Importing packages from multiple channels

A NixOS config flake skeleton could be as follows:



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

  inputs = {
    nixpkgs.url = "nixpkgs/nixos-21.11";
    nixpkgs-unstable.url = "nixpkgs/nixos-unstable";
  };

  outputs = { self, nixpkgs, nixpkgs-unstable }:
  let
    system = "x86_64-linux";
    overlay-unstable = final: prev: {
      unstable = nixpkgs-unstable.legacyPackages.${prev.system};
      # use this variant if unfree packages are needed:
      # unstable = import nixpkgs-unstable {
      #   inherit system;
      #   config.allowUnfree = true;
      # };
    };
    in {
      nixosConfigurations."<hostname>" = nixpkgs.lib.nixosSystem {
        inherit system;
        modules = [
          # Overlays-module makes "pkgs.unstable" available in configuration.nix
          ({ config, pkgs, ... }: { nixpkgs.overlays = [ overlay-unstable ]; })
          ./configuration.nix
        ];
      };
    };
  };
}
```

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

Same can be done with the NURs, as it already has an `overlay` attribute in the `flake.nix` of the project, you can just add

```
nixpkgs.overlays = [ nur.overlay ];
```



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

```
pkgs = import nixpkgs { overlays = [ /*the overlay in question*/ ]; };
```

Getting Instant System Flakes Repl

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

```
# nix repl
>> :lf /etc/nixos
>> nixosConfigurations.myhost.config
{ ... }
```

Or out of your current flake:

```
# nix repl
>> :lf .#
```

You can then access to the inputs, outputs... For instance if you would like to check the default version of the kernel present in nixpkgs:

```
nix-repl> inputs.nixpkgs.legacyPackages.x86_64-linux.linuxPackages.kernel.version
"5.15.74"
```

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/ *|
```

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 (/wiki/Unfree_Software).

Development tricks

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 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

- 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-nixos-flakes/>) (Eelco Dolstra, 2020)
- NixOS & Flakes Book (<https://github.com/ryan4yin/nixos-and-flakes-book>)(Ryan4yin, 2023) - 🔨 ❤️ An unofficial NixOS & Flakes book for beginners.
- 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: 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 101: Introduction to nix flakes (https://www.youtube.com/watch?v=QXUlhnhuRX4&list=PLgknCdxP89RcGPTjngfNR9WmBgvD_xW0I) (Jörg Thalheim, 2020)
- RFC 49 (<https://github.com/NixOS/rfcs/pull/49>) (2019) - Original flakes specification
- 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>)

Retrieved from "<https://nixos.wiki/index.php?title=Flakes&oldid=11071>" (<https://nixos.wiki/index.php?title=Flakes&oldid=11071>)

Categories (/wiki/Special:Categories):

Pages with syntax highlighting errors (/index.php?title=Category:Pages_with_syntax_highlighting_errors&action=edit&redlink=1)

| Software (/wiki/Category:Software) | Nix (/wiki/Category:Nix)

| Flakes (/index.php?title=Category:Flakes&action=edit&redlink=1)



community.flake.parts >

Managing OS and home configurations using **nixos-flake** >



Flake Templates

We provide four templates, depending on your needs:

Available templates

Both platforms

NixOS, nix-darwin, [home-manager](#) configuration combined, with common modules.

```
nix flake init -t github:srid/nixos-flake
```

NixOS only

NixOS configuration only, with [home-manager](#)

```
nix flake init -t github:srid/nixos-flake#linux
```

macOS only

nix-darwin configuration only, with [home-manager](#)

```
nix flake init -t github:srid/nixos-flake#macos
```



Home only

[home-manager](#) configuration only (useful if you use other Linux distros or do not have admin access to the machine)

```
nix flake init -t github:srid/nixos-flake#home
```

After initializing the template

1. open the generated `flake.nix` and change the user (from “john”) as well as hostname (from “example1”) to match that of your environment (Run `echo $USER` and `hostname -s` to determine the new values). ¹
2. Then run `nix run .#activate` (`nix run .#activate-home` if you are using the 4th template) to activate the configuration.

Footnotes

1.

If you are on an Intel Mac, change `mkARMMacosSystem` to `mkIntelMacosSystem`.



Links to this page

Guide

Flake Templates

Getting Started

Convert your `flake.nix` to using `nixos-flake` using the NixOS only template as reference.

Use the HOME only template

Use the macOS only template*

Alternatively, use the “Both platforms” template if you are sharing your configuration with the other platform as well.



Examples

<https://github.com/hkmangla/nixos> (using `#linux` template)

<https://github.com/juspay/nix-dev-home> (using `#home` template)

<https://github.com/srid/nixos-config> (using `#both` template)



[Xe](#)[Blog](#)[Contact](#)[Resume](#)[Talks](#)[VODs](#)[Signalboost](#)

How to look up a Nix package's Nix store path from flake inputs

Published on 08/06/2022, 710 words, 3 minutes to read



The fall of the Archons, colored pencil drawing, fireball spell, bright sky, digital art, lake of fire - Midjourney

Sometimes God is dead and you need to figure out what the version of a package in your Nix flake's inputs is. With flakes, you can figure this out using `nix eval` on a flake reference, but what the hecc is a flake reference?

Every URL has a "fragment", or the thing after the `#` that mainly tells the browser where to scroll to. Nix flakes uses this to allow you to sub-index on flakes by either name or URL. This allows you to address packages like tmux as:

```
nixpkgs#legacyPackages.x86_64-linux.tmux
```

Or my site's code as:



```
github:Xe/site#packages.x86_64-linux.default
```

Or even some arbitrary git repo:

```
git+https://tulpa.dev/cadey/printefacts.git?ref=main#defaultPackage.x86_64-linux
```



<Mara> The flake reference is what drills down into the flake so that Nix can tell the difference between part of the path to the repository and the path inside the flake. It usually works on the `outputs` of a flake that you can find out with `nix flake show`:

```
$ nix flake show github:Xe/site
github:Xe/site/5a50caccef85c9e3b9695a2749359b27d0530f86a
├── devShell
│   ├── aarch64-linux: development environment 'nix-shell'
│   └── x86_64-linux: development environment 'nix-shell'
├── nixosModules
│   ├── aarch64-linux: NixOS module
│   └── x86_64-linux: NixOS module
└── packages
    ├── aarch64-linux
    │   ├── bin: package 'xesite-2.4.0'
    │   ├── config: package 'xesite-config-2.4.0'
    │   ├── default: package 'xesite-2.4.0'
    │   ├── posts: package 'xesite-posts-2.4.0'
    │   └── static: package 'xesite-static-2.4.0'
    └── x86_64-linux
        ├── bin: package 'xesite-2.4.0'
        ├── config: package 'xesite-config-2.4.0'
        ├── default: package 'xesite-2.4.0'
        ├── posts: package 'xesite-posts-2.4.0'
        └── static: package 'xesite-static-2.4.0'
```



<Mara> Tracing through things, you can see how you can go from the package



name `default` to get `github.com:Xe/site#packages.x86_64-linux.default` as the full legal name of the package!

You can use the `nix eval --raw` command to figure out what the nix store path of those packages either is or would be:

```
$ nix eval --raw nixpkgs#legacyPackages.x86_64-linux.tmux  
/nix/store/3mn362yak70vq137wnryi7whgvq2cmn2-tmux-3.3a
```

```
$ nix eval --raw github:Xe/site#packages.x86_64-linux.default  
/nix/store/v3lwgh6vk03yw3aq3j398g613ff3gja0-xesite-2.4.0
```

```
$ nix eval --raw git+https://tulpa.dev/cadey/printefacts.git'?ref=main#defaultPackage.x86_64  
/nix/store/7df21bqkrx2csazc2s8ji8chaabbpqhs-printefacts-0.3.1
```

However, this operates on the latest version by default. It won't operate on the current version in your Nix flake. If you need to find it out from the current flake, pass the `--inputs-from` flag with the current directory `.`:

```
$ nix eval --inputs-from . --raw nixpkgs#legacyPackages.x86_64-linux.tmux  
/nix/store/5sz57bwrlwhv1l8h4c4sq1bl1fhmcwfw-tmux-3.3a
```

This will use the `nixpkgs` defined in your current directory's `flake.lock` file. You can then share that store path with people to share in your tale of woe as to why a service is crashing when you try and update it.



<**Mara**> You can also pass in an arbitrary version for a flake URL if you specify it in the URL parameters with `ref`. You can do it like this:

`github:Xe/site?ref=5a50cacef85c9e3b9695a2749359b27d0530f86a` or by specifying the commit hash as an additional path parameter as in `github:Xe/site/5a50cacef85c9e3b9695a2749359b27d0530f86a`. Here's an example for this xesite commit:



```
$ nix flake show github:Xe/site/5a50cacef85c9e3b9695a2749359b27d0530f86a
```

```
github:Xe/site/5a50cacef85c9e3b9695a2749359b27d0530f86a
├── devShell
│   ├── aarch64-linux: development environment 'nix-shell'
│   └── x86_64-linux: development environment 'nix-shell'
├── nixosModules
│   ├── aarch64-linux: NixOS module
│   └── x86_64-linux: NixOS module
└── packages
    ├── aarch64-linux
    │   ├── bin: package 'xesite-2.4.0'
    │   ├── config: package 'xesite-config-2.4.0'
    │   ├── default: package 'xesite-2.4.0'
    │   ├── posts: package 'xesite-posts-2.4.0'
    │   └── static: package 'xesite-static-2.4.0'
    └── x86_64-linux
        ├── bin: package 'xesite-2.4.0'
        ├── config: package 'xesite-config-2.4.0'
        ├── default: package 'xesite-2.4.0'
        ├── posts: package 'xesite-posts-2.4.0'
        └── static: package 'xesite-static-2.4.0'
```



<**Cadey**> I have reached a point where googling for these questions gets me results on my own blog. Of course they didn't help me in the moment. That's why this post exists. Maybe I'll be able to fix my Matrix server returning that a database migration failed so that I can update my homelab machines because that failing makes the whole deploy rollback. I really hate synapse and I regret setting up my own Matrix homeserver.



Share



Facts and circumstances may have changed since publication. Please contact me before jumping to conclusions if something seems wrong or unclear.

Tags: nix, nixos, flakes

Copyright 2012-2024 Xe Iaso (Christine Dodrill). Any and all opinions listed here are my own and not representative of any of my employers, past, future, and/or present.

Like what you see? Donate on [Patreon](#) like [these awesome people!](#)

Served by xesite v4 (/nix/store/9rxy3y9y9ffwx6xdvb56pgciar4b5138-xesite_v4-20240128
/bin/xesite) with site version [a1841325](#) , source code available [here](#).



Install NixOS with Flake configuration on Git

This tutorial will walk you through the steps necessary to install **NixOS**, enable **flakes** while tracking the resulting system configuration in a **Git** repository.

Welcome to the tutorial series on **NixOS**

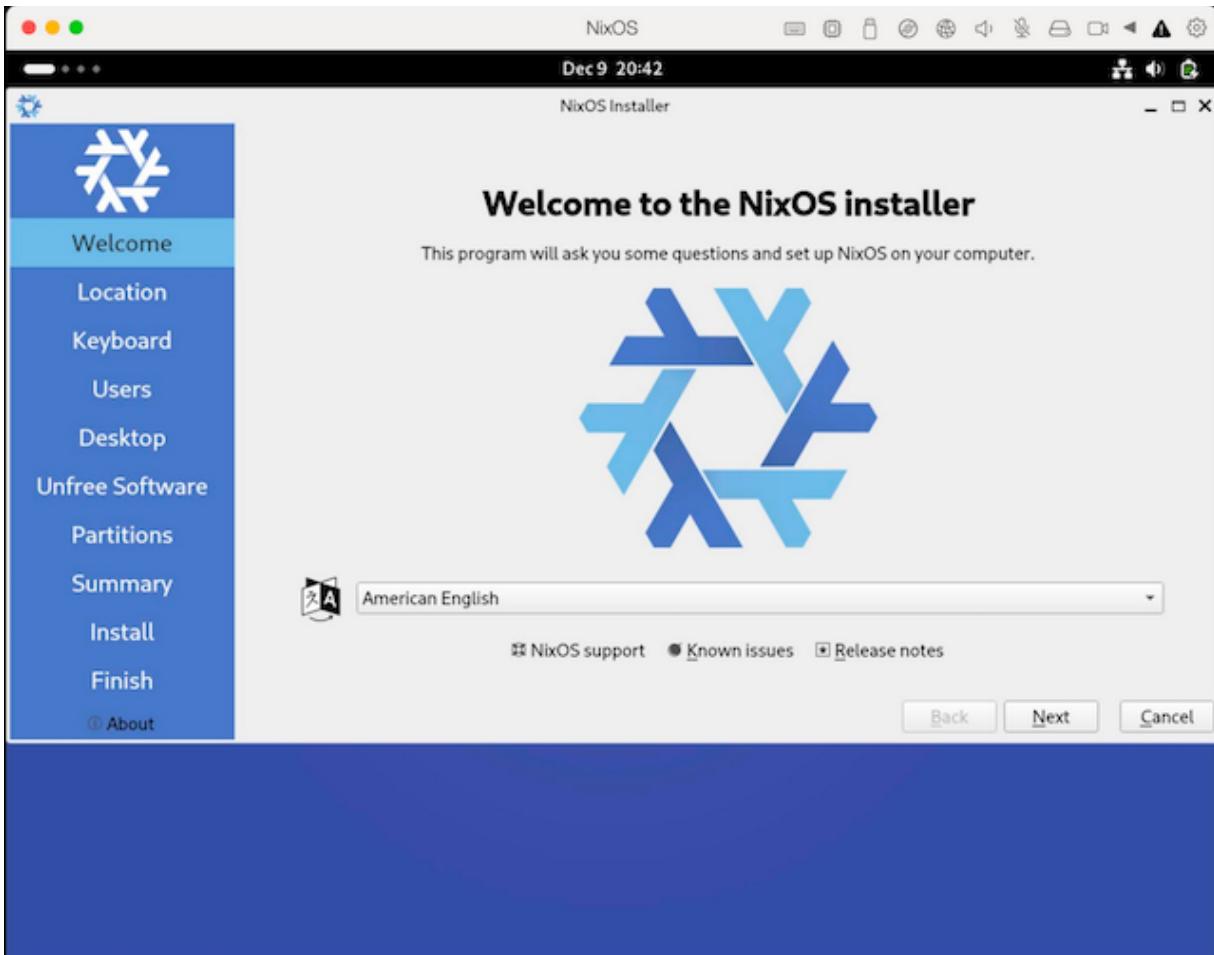
This page is the first in a planned series of tutorials aimed towards onboarding Linux/macOS users into comfortably using **NixOS** as their primary operating system.



Install NixOS

- Download the latest NixOS ISO from [here](#). Choose the GNOME (or Plasma) graphical ISO image for the appropriate CPU architecture.
- Create a bootable USB flash drive ([instructions here](#)) and boot the computer from it.

NixOS will boot into a graphical environment with the installer already running.



Go through the installation wizard; it is fairly similar to other distros. Once NixOS install is complete, reboot into your new system. You will be greeted with a login screen.

- Login as the user you created with the password you set during installation.
- Then open the “Console” application from the “Activities” menu.

Your first `configuration.nix` change

Your systems configuration includes everything from partition layout to kernel version to packages to services. It is defined in `/etc/nixos/configuration.nix`. The `/etc/nixos` directory looks like this:

```
$ ls -l /etc/nixos
-rw-r--r-- 1 root root 4001 Dec  9 16:03 configuration.nix
-rw-r--r-- 1 root root 1317 Dec  9 15:43 hardware-configuration.nix
```

ⓘ What is hardware-configuration.nix?

Hardware specific configuration (eg: disk partitions to mount) are defined in `/etc/nixos/hardware-configuration.nix` which is imported, as a **module**, by `configuration.nix`.

All system changes require a change to this `configuration.nix`. For example, in order to “install” or “uninstall” a package, we would edit this `configuration.nix` and activate it. Let’s do this now to install the `neovim` text editor. NixOS includes the nano editor by default:

```
sudo nano /etc/nixos/configuration.nix
```

🔮 Nix language

These `*.nix` files are written in the `Nix` language.

In the text editor, make the following changes:

- Add `neovim` under `environment.systemPackages`
- [Optional] uncomment `services.openssh.enable = true;` to enable the SSH server

Press `Ctrl+X` to exit nano.

Your `configuration.nix` should now look like:

```
# /etc/nixos/configuration.nix
{
  ...
  environment.systemPackages = with pkgs; [
    neovim
  ];
  ...
  services.openssh.enable = true;
  ...
}
```

```
}
```

Once the `configuration.nix` file has been saved to disk, you must activate that new configuration using the `nixos-rebuild` command:

```
sudo nixos-rebuild switch
```

This will take a few minutes to complete—as it will have to fetch neovim and its dependencies from the official [binary cache](#) (`cache.nixos.org`). Once it is done, you should expect to see something like this:

```

NixOS
Dec 9 16:07
srid@nixos:/etc/nixos
[etc/nixos]

building '/nix/store/z4i8vrbkxflka8npvhdm9wrik41v2rkk-sshd.conf-final.drv'...
copying path '/nix/store/szzz3mp9xs795bv0fgha5bz5147pa-lua5it-2.1.1693350652-env' from 'https://cache.nixos.org'...
copying path '/nix/store/374dlrwq2877vbaprim2kw3lqj1i53-neovim-ruby-env' from 'https://cache.nixos.org'...
copying path '/nix/store/14pdf8agfd26vblos775k4qdkyk6pm5m-python3-3.11.6-env' from 'https://cache.nixos.org'...
building '/nix/store/lntbkv3n52sakfsw8ibll1v24ni19mpv-unit-firewall.service.drv'...
building '/nix/store/27prwdnmnghy196a7n5w9jpra5dalbmw-X-Restart-Triggers-sshd.drv'...
building '/nix/store/s65kj0cdpxi25svaisgj4aj65xnq7xd-check-sshd-config.drv'...
copying path '/nix/store/xq4mazfkhk809qcfz7794h1354s5z8k4-neovim-unwrapped-0.9.4' from 'https://cache.nixos.org'...
building '/nix/store/s4c3q1j683jxvyl17kcx1s3jfcj3y42g-unit-sshd.service.drv'...
copying path '/nix/store/4mydmj1pfx0wc30jv88062nriyd05fq-neovim-0.9.4' from 'https://cache.nixos.org'...
building '/nix/store/w3pmhlnz7apq7d8819rm2ymwrk1jyy-system-path.drv'...
created 15393 symlinks in user environment
gtk-update-icon-cache: Cache file created successfully.
building '/nix/store/p06xqkzszyimrdds7f0bw15ax4rwjslb-X-Restart-Triggers-polkit.drv'...
building '/nix/store/fapd64vdgz0jwmmbjnplbzqwfwlj6-dbus-1.drv'...
building '/nix/store/4sj5j544241mvclpnccxwfg23fa95jns-etc-pam-environment.drv'...
building '/nix/store/cl45ljkw9iyhxha47jswxbjd1jwab0p9-set-environment.drv'...
building '/nix/store/gv6ndkwpqmx3kv1swmkd6zj4j7x96wp8-unit-accounts-daemon.service.drv'...
building '/nix/store/q060q15vbhk88f91202q891cd0wf3sk-X-Restart-Triggers-dbus.drv'...
building '/nix/store/gbwnfzvffhv0m2nzqplpk7sn5r2m7ds-etc-profile.drv'...
building '/nix/store/qp280cy1spwzd0habhgq5m6jfhx8kq-unit-polkit.service.drv'...
building '/nix/store/gqjv7swy45jvv155c58yrig2hj7qm40f-unit-dbus.service.drv'...
building '/nix/store/qc0icfblls1y56wf3vbq8s1m92f6bys-unit-dbus.service.drv'...
building '/nix/store/wffjbqchgivsqjxk6ldz0lx5fjkgajbj-user-units.drv'...
building '/nix/store/8rq3ixwdm9r50ba8a5898hp1bvy10n-system-units.drv'...
building '/nix/store/83m7j11jmvbxdw4y5rszk3j0vlg2xx-etc.drv'...
building '/nix/store/9sydwx15emqrkk1kg3q738b768afyxsw-nixos-system-nixos-23.11.1494.b4372c4924d9.drv'...
stopping the following units: accounts-daemon.service
activating the configuration...
setting up /etc...
reloading user units for srid...
setting up tmpfiles
reloading the following units: dbus.service, firewall.service
restarting the following units: polkit.service
starting the following units: accounts-daemon.service
the following new units were started: sshd.service

[srid@nixos:/etc/nixos]#

```

You can confirm that neovim is installed by running `which nvim`:

```
$ which nvim
/run/current-system/sw/bin/nvim
```

Remote access

Now that you have OpenSSH enabled, you may do the rest of the steps

from another machine by ssh'ing to this machine.

Flakeify

Convert `configuration.nix` to be a flake

A problem with the default NixOS `configuration.nix` generated by the official installer is that it is not “pure” and thus not reproducible (see [here](#)), as it still uses a mutable Nix channel (which is generally [discouraged](#)). For this reason (among others), it is recommended to immediately switch to using [Flakes](#) for our NixOS configuration. Doing this is pretty simple. Just add a `flake.nix` file in `/etc/nixos`:

```
sudo nvim /etc/nixos/flake.nix
```

Add the following:

```
# /etc/nixos/flake.nix
{
  inputs = {
    # NOTE: Replace "nixos-23.11" with that which is in system.stateVersion
    # configuration.nix. You can also use latter versions if you
    # upgrade.
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-23.11";
  };
  outputs = { self, nixpkgs }: {
    # NOTE: 'nixos' is the default hostname set by the installer
    nixosConfigurations.nixos = nixpkgs.lib.nixosSystem {
      # NOTE: Change this to aarch64-linux if you are on ARM
      system = "x86_64-linux";
      modules = [ ./configuration.nix ];
    };
  };
}
```

 **Make sure to change a couple of things in the above snippet:**

- Replace `nixos-23.11` with the version from `system.stateVersion` in your `/etc/nixos`

`/configuration.nix`. If you wish to upgrade right away, you can also use latter versions, or use `nixos-unstable` for the bleeding edge.

- `x86_64-linux` should be `aarch64-linux` if you are on ARM

Now, `/etc/nixos` is technically a `flake`. We can “inspect” this flake using the `nix flake show` command:

```
$ nix flake show /etc/nixos
error: experimental Nix feature 'nix-command' is disabled; use '...
```

Oops, what happened here? As flakes is a so-called “experimental” feature, you must manually enable it. We’ll *temporarily* enable it for now, and then enable it *permanently* latter. The `--extra-experimental-features` flag can be used to enable experimental features. Let’s try again:

```
$ nix --extra-experimental-features 'nix-command flakes' flake st
warning: creating lock file '/etc/nixos/flake.lock'
error:
... while updating the lock file of flake 'path:/etc/nixos?1
error: opening file '/etc/nixos/flake.lock': Permission de
```

Progress, but we hit another error—Nix understandably cannot write to root-owned directory (it tries to create the `flake.lock` file). One way to resolve this is to move the whole configuration to our home directory, which would also prepare the ground for storing it in `Git`. We will do this in the next section.

ⓘ `flake.lock`

Nix commands automatically generate a (or update the) `flake.lock` file. This file contains the exact pinned version of the inputs of the flake, which is important for reproducibility.

Move configuration to user

directory

Move the entire `/etc/nixos` directory to your home directory and gain control of it:

```
$ sudo mv /etc/nixos ~/nixos-config && sudo chown -R $USER ~/nixos
```

Your configuration directory should now look like:

```
$ ls -l ~/nixos-config/
total 12
-rw-r--r-- 1 srid root 4001 Dec  9 16:03 configuration.nix
-rw-r--r-- 1 srid root  224 Dec  9 16:12 flake.nix
-rw-r--r-- 1 srid root 1317 Dec  9 15:43 hardware-configuration.r
```

Now let's try `nix flake show` on it, and this time it should work:

```
$ cd ~/nixos-config
$ nix --extra-experimental-features 'nix-command flakes' flake status
warning: creating lock file '/home/srid/nixos-config/flake.lock'
path:/home/srid/nixos-config?lastModified=1702156518&narHash=sha256-...
└─nixosConfigurations
    └─nixos: NixOS configuration
```

Voila! Incidentally, this flake has a single output, `nixosConfigurations.nixos`, which is the NixOS configuration itself.

ⓘ More on Flakes

See [Rapid Introduction to Nix](#) for more information on flakes.

Once flake-ified, we can use the same command to activate the new configuration but we must additionally pass the `--flake` flag, viz.:

```
# The '.' is the path to the flake, which is current directory.  
$ sudo nixos-rebuild switch --flake .
```

If everything went well, you should see something like this:

```
[srid@nixos:~/nixos-config]$ sudo nixos-rebuild switch --flake .
building the system configuration...
```

```
stopping the following units: accounts-daemon.service
activating the configuration...
setting up /etc...
reloading user units for srid...
setting up tmpfiles
reloading the following units: dbus.service
restarting the following units: polkit.service
starting the following units: accounts-daemon.service

[srid@nixos:~/nixos-config]$
```

Excellent, now we have a flake-ified NixOS configuration that is pure and reproducible!

Let's [store our whole configuration](#) in a [Git](#) repository.

Store the configuration in Git

First we need to install [Git](#):

- add `git` to `environment.systemPackages`, and
- activate your new configuration using `sudo nixos-rebuild switch --flake ..`

Then, create a Git repository for your configuration:

```
$ cd ~/nixos-config
$ git config --global user.email "srid@srid.ca"
$ git config --global user.name "Sridhar Ratnakumar"
$ git init && git add . && git commit -m init
```

You may now [create a repository](#) on GitHub or your favourite Git host, and push your configuration repo to it.

Benefits of storing configuration on Git

- If you buy a new computer, and would like to reproduce your NixOS setup, all you have to do is clone your configuration repo, adjust your `hardware-configuration.nix` and run `sudo nixos-rebuild switch --flake ..`
- Version controlling configuration changes makes it straightforward to

point out problems and/or rollback to previous state.

Enable flakes

As a final step, let's permanently enable **Flakes** on our system, which is particular useful if you do a lot of **software development**. This time, instead of editing `configuration.nix` again, let's do it in a separate **module** (for no particular reasons other than pedagogic purposes). Remember the `modules` argument to `nixosSystem` function in our `flake.nix`? It is a list of modules, so we can add a second module there:

```
diff --git a/flake.nix b/flake.nix
index cc77fb9..4e84bdf 100644
--- a/flake.nix
+++ b/flake.nix
@@ -8,7 +8,14 @@
    # NOTE: 'nixos' is the default hostname
    nixosConfigurations.nixos = nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
-        modules = [ ./configuration.nix ];
+        modules = [
+            ./configuration.nix
+            {
+                nix = {
+                    settings.experimental-features = [ "nix-command" "flakes"
+                    ];
+                }
+            ];
        };
    };
}
```

🔥 NixOS options

You can see all the available options for NixOS in the [NixOS options](#) search engine.

As before, we must activate the new configuration using `sudo nixos-rebuild switch --flake ..`. Once that is done, we can verify that flakes is enabled by re-running `nix flake show` but without the `--extra-`

experimental-features flag:

```
$ nix flake show
warning: Git tree '/home/srid/nixos-config' is dirty
git+file:///home/srid/nixos-config
└─nixosConfigurations
    └─nixos: NixOS configuration
```

Recap

You have successfully installed NixOS. The entire system configuration is also stored in a Git repo, and can be reproduced at will during either a reinstallation or a new machine purchase. You can make changes to your configuration, commit them to Git, and push it to GitHub. Additionally we enabled **Flakes** permanently, which means you can now use all the modern `nix` commands, such as running a package directly from `nixpkgs` (same version pinned in `flake.lock` file):



Up Next

In part 2 of this tutorial, we will use [nixos-flake](#) for more convenience, as well as use [home-manager](#) (to manage home configuration). Then we'll describe several common NixOS workflows.



Managing your NixOS configuration with Flakes and Home Manager!



JOSIAH

SEP 8, 2023



Share



If you followed one of my [earlier posts about NixOS](#), then you should already be somewhat familiar with the basics of managing and configuring your NixOS.

However, Nix can be much more than a single configuration.nix file? In this guide we will be setting up not one, BUT TWO powerful tools. Flakes and Home Manager!

Ok, but what are Nix Flakes?

Flakes are an experimental feature of Nix that aims to fix some of the challenges around true reproducibility. They ensure that the same inputs always produce the same outputs.

You can manage your system configurations (yes, plural), distribute nixpkgs overlays, and modules more efficiently. Flakes make it easy to share your entire system down to the exact package version with anyone.

Flakes are stored in a file called flake.nix.

This file contains bidirectional Unicode text that may be interpreted or compiled differently than what appears below. To review, open the file in an editor that reveals hidden Unicode characters. [Learn more about bidirectional Unicode characters](#)

[Show hidden characters](#)

```
{  
description = "A simple NixOS flake";  
  
inputs = {  
nixpkgs.url = "github:NixOS/nixpkgs/nixos-unstable";  
};  
  
outputs = { self, nixpkgs, ... }: {  
nixosConfigurations = {  
hostName = nixpkgs.lib.nixosSystem {  
system = "x86_64-linux";  
modules = [ ./configuration.nix ];
```



```
};  
};  
};  
}  
}
```

[flake.nix](#) hosted with ❤ by GitHub

[view raw](#)

Above is one of the simplest flakes you can make for managing your home system.

The inputs section of the file specifies the flake's dependencies. Here we have nixpkgs defined. It points to the GitHub URL that contains the unstable branch of nixpkgs. You can look at it [here](#).

The second main part is the outputs. It inherits itself and nixpkgs from the inputs section above. The single output in this case is nixosConfigurations. As it sounds, this output defines the NixOS system configs.

hostName is a placeholder variable for the machine's name. In most cases, it is named after the machine's hostname (which you can find by typing `hostname` in the terminal). You can call it whatever, but if it's not the machine's hostname then you will have to call that name specifically when building.

It calls the `nixpkgs.lib.nixosSystem` function. That function specifies the system's architecture and the modules list. As you might have noticed, the modules list contains the relative path to the system configuration file.

When the flake file is built, it creates a `flake.lock` file. Just like how Javascript has a `package.json` file and a `package-lock.json` file, Nix has a `flake.nix` file and `flake.lock` file.

Before we move on to enabling and creating a flake file, let's touch on Home Manager and how we will be using it.

What is Home Manager?

While `/etc/nixos/configuration.nix` configures the system, Home Manager takes it a step further and allows you to manage user environments. With the Nix language, you can declaratively define user environmental variables, packages, and most importantly, dotfiles.

It can be used as standalone software, allowing users to update their dotfiles without root privileges, or as a NixOS module. If it's a Nix module then you will have to rebuild your system every time you want to update your dotfiles. I personally have it set as a module, but you can install it standalone as well.

Getting started

To get started you first need to enable flake support. It is technically an experimental feature, but this is because it's subject to breaking changes in the future. As of right now it is really stable and is becoming standard for most configurations.

Adding flakes is super simple. Copy this line and add it to your `configuration.nix` file.

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

Rebuild your system and then you can start writing flakes.



In the root of your configuration directory `/etc/nixos` run the command:

```
nix flake init
```

This will generate an extremely basic flake.nix file. Go ahead and change the default description to whatever you want and then remove everything from inside the outputs object.

You can copy the flake example from before for you starting flake. To build this flake you will need to run this command from the **directory the flake is in**.

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

The --flake flag points to the flake we are switching to. The period is the directory that the flake is stored in and the number sign specifies the hostname defined in the file. If you called the hostname in the flake the same as the machine host name then you can leave the symbol blank, otherwise, you will need to append the hostname defined in the flake to it. Like this .#hostName.

If everything went well then you should have a flake.lock file. If you want, open it up and look around. It will have a bunch of hashes and other stuff. Don't edit it.

To install home manager go ahead and use this flake file. It will install home manager as a module.

This file contains bidirectional Unicode text that may be interpreted or compiled differently than what appears below. To review, open the file in an editor that reveals hidden Unicode characters. [Learn more about bidirectional Unicode characters](#)

[Show hidden characters](#)

```
{  
  description = "NixOS configuration with Home Manager";  
  
  inputs = {  
    nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";  
    home-manager = {  
      url = "github:nix-community/home-manager";
```



[View raw](#)

Make sure to replace <yourUsername> with your actual username. In the inputs, you might notice that home manager is split into two parts. The GitHub URL points to the master branch. The reason for this is because it follows the unstable branch of nixpkgs. If you are using a stable release of Nix, update the home manager URL accordingly. The line below the URL tells home manager to follow the unstable branch.



Configuring things inside Home Manager

After rebuilding (using the command from before), you can start configuring the `./home.nix` file. You can also define the path elsewhere, just make sure it's relative to the flake.

Here is an example file. You can install user specific packages and define the dotfile for zsh.

This file contains bidirectional Unicode text that may be interpreted or compiled differently than what appears below. To review, open the file in an editor that reveals hidden Unicode characters. [Learn more about bidirectional Unicode characters](#)

Show hidden characters

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

{

# if you config gets too long, split it up into smaller modules
imports = [
  ./git # looks for ./git/defualt.nix
  ./hypr/hyprland.nix # looks for ./hypr/hyprland.nix
];

# The User and Path it manages
home.username = "<yourUsername>";
home.homeDirectory = "/home/<yourUsername>";

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

# List of user programs
home.packages = with pkgs; [
  zsh
  git
  firefox
];

# ZSH
programs.zsh = {
  enable = true;
  enableCompletion = true;
```



```
enableAutosuggestions = true
shellAliases = {
  _ = "sudo";
  h = "history";
  hg = "history | grep ";
};
};

}
```

[home.nix](#) hosted with ❤ by [GitHub](#)

[view raw](#)

There is a ton more you can do with Home Manager and Flakes too, but there is too much to cover in this one post. Here are some resources that you can use to learn more about them!

- [NixOS wiki Flakes](#)
- [Nix Flakes Book](#)
- [Home Manager guide](#)
- [Home Manager Option search](#)
- [Nixpkgs, options, and flakes search](#)
- [My personal dotfiles](#)

Type your email...

Subscribe

Comments



Write a comment...





Tags

Nix

Updated at

Apr 26, 2022 8:17 AM

Published at

May 9, 2020

NOTE: All of this is completely unstable so
please don't adopt this just yet

Nix Flakes is an experimental branch of the Nix project that adds:

- A central `flake.nix` entry-point to Nix projects.
- Built-in dependency management
- Is tied to Git
- Per-commit evaluation caching
- A new `nix` CLI.



Here are some notes that I took for myself on the subject.

Other sources

- <https://nixos.wiki/wiki/Flakes>
- [Summary of Nix Flakes vs original Nix](#)
- <https://edolstra.github.io/talks/nixcon-oct-2019.pdf>
- <https://www.tweag.io/blog/2020-05-25-flakes/>
- <https://www.tweag.io/blog/2020-06-25-eval-cache/>
- <https://www.tweag.io/blog/2020-07-31-nixos-flakes/>

Installation

NixOS

Add the following options to the NixOS configuration (on nixos-unstable):

```
{ pkgs, ... }:{  
    # Enable the nix 2.0 CLI and flakes support  
    feature-flags  
    nix.extraOptions = ''  
        experimental-features = nix-command flakes  
    '';  
}
```



Then run `nixos-rebuild switch` and that's it.

Other systems

Install Nix 2.5.0 or later. Then edit either `~/.config/nix/nix.conf` or `/etc/nix/nix.conf` and add:

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

This is needed to expose the Nix 2.0 CLI and flakes support that are hidden behind feature-flags.

Finally, if the Nix installation is in multi-user mode, don't forget to restart the nix-daemon (you can check that by running `ps aux | grep nix-daemon` to see if it's running).

Basic project usage

NOTE: flake makes a strong assumption that the folder is a git repository. It doesn't work outside of them.

In your repo, run `nix flake init` to generate the `flake.nix` file. Then run `git add flake.nix` to add it to the git staging area, otherwise nix will not recognize that the file exists.

TODO: add more usage examples here.



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

Flake schema

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

It has 3 top-level attributes:

- `description` which is self...describing
- `nixConfig` allows to set per-project Nix configuration.
- `input` is an attribute set of all the dependencies of the flake. The schema is described below.
- `output` is a function of one argument that takes an attribute set of all the realized inputs, and outputs another attribute set which schema is described below.

nixConfig schema

Eg (from the commit message):

```
{  
    nixConfig.bash-prompt-suffix = "ngi# ";  
    nixConfig.substituters = [  
        "https://cache.ngi0.nixos.org/" ];  
}
```



Input schema

This is not a complete schema but should be enough to get you started:

```
{  
    inputs.bar = {  
        # Source of the input. It supports  
        `github:` `gitlab:` and a number of      # other  
        schemes  
        url = "github:foo/bar/branch";  
        # Turn off if the target is not a flake.  
        flake = false;  
        # Used to override inputs of the target if  
        it is a flake.  
        inputs = {  
            # For example, here we declare to use  
            the same version as the parent  
            # nixpkgs. It's probably also possible  
            to override the URL attribute.  
            nixpkgs.follows = "nixpkgs";  
        };  
    };  
}
```

The `bar` input is then passes to the

Output schema



Here is what I found out while reading `src/nix/flake.cc` in `CmdFlakeCheck`.

Where:

- `<system>` is something like "x86_64-linux".
- `<machine>` is something like "mymachine".
- `<attr>` is an attribute name like "hello".
- `<job>` is a hydra job name like "release".
- `<flake>` is a flake name like "nixpkgs".
- `<store-path>` is a /nix/store.. path

```
{ self, ... }@inputs:  
{  
    # Executed by `nix flake check`  
    checks."<system>".<attr>" = derivation;  
    # Executed by `nix build .#<name>`  
    packages."<system>".<attr>" = derivation;  
    # Executed by `nix build .`  
    defaultPackage."<system>" = derivation;  
    # Executed by `nix run .#<name>`  
    apps."<system>".<attr>" = {  
        type = "app";  
        program = "<store-path>";  
    };  
    defaultApp."<system>" = { type = "app";  
    program = "..."; };  
  
    # TODO: Not sure how it's being used  
    legacyPackages = TODO;  
    # TODO: Not sure how it's being used  
    overlay = final: prev: { };  
    # TODO: Same idea as overlay but several.  
    overlays."<attr>" = final: prev: { };  
    # TODO: Not sure how it's being used  
    nixosModule = TODO;
```



```
# TODO: Same idea as nixosModule but several
nixosModules."<attr>" = TODO;
# TODO: Not sure how it's being used
nixosConfigurations."<machine>" = TODO;
# TODO: Similar idea as for nixosModules but
for hydra jobs.
hydraJobs."<job>" = TODO;
# Used by `nix flake init -t <flake>`
defaultTemplate = {
    path = "<store-path>";
    description = "template description goes
here?";
};
# Used by `nix flake init -t <flake>#<attr>`
templates."<attr>" = { path = "<store-
path>"; description = ""; };
}
```

See also:

- <https://github.com/NixOS/nix/blob/master/src/nix/flake-check.md>

Building NixOS configurations with Flakes



There is a special, undocumented way to build NixOS configurations with flakes.

First, change `flake.nix` to output a configuration. This uses the `nixosConfigurations` key. The `nixpkgs` flake includes

a helper for that:

```
{  
    outputs = { nixpkgs, ... }: {  
        nixosConfigurations.mymachine =  
nixpkgs.lib.nixosSystem {  
            modules = [  
                # Point this to your original  
                configuration.  
                ./machines/mymachine/configuration.nix  
            ];  
            # Select the target system here.  
            system = "x86_64-linux";  
        };  
    };  
}
```

Then to switch configurations, use `nixos-rebuild --flake .#mymachine switch`, from the same repository where the `flake.nix` file is located.

To switch a remote configuration, use:

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



NOTE: Remote building seems to be broken at the moment, which is why the build host is set to “localhost”.

Super fast nix-shell

One of the nix feature of the 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> { } }:
  with pkgs;
  mkShell {
    buildInputs = [
      nixpkgs-fmt
    ];

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



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

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

```
{  
    description = "my project description";  
  
    inputs.flake-utils.url =  
    "github:numtide/flake-utils";  
  
    outputs = { self, nixpkgs, flake-utils }:  
        flake-utils.lib.eachDefaultSystem  
        (system:  
            let pkgs =  
                nixpkgs.legacyPackages.${system}; in  
            {  
                devShell = import ./shell.nix {  
                    inherit pkgs; };  
            }  
        );  
}
```

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.



NOTE: 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.

Direnv integration

direnv 2.29.0 and later ship with the `use flake` builtin function. Just add that to your `.envrc` and you're good to go!

The nice thing about this approach is that evaluation is cached.

Optimize the reloads

Nix Flakes has a Nix evaluation caching mechanism. Is it possible to expose that somehow to automatically trigger direnv reloads?

With the previous solution, direnv would only reload iff the `flake.nix` or `flake.lock` files have changed. This is not completely precise as the `flake.nix` file might import other files in the repository.

Using with GitHub Actions

See <https://github.com/numtide/nix-flakes-installer#githhub-actions>



Pushing Flake inputs to Cachix

Flake inputs can also be cached in the Nix binary cache!

```
nix flake archive --json \ | jq -r '.path,  
.inputs|to_entries[].value.path)' \ | cachix  
push $cache_name
```

How to 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 all the derivations of a flake

Traditionally we would run `nix-build ci.nix` or something equivalent but flakes only support pointing `nix build` to a single derivation.



I am not 100% confident on this answer: it looks like exposing the derivation in the `checks` output attribute, and then

running `nix flake check` does the trick.

Some file is not found

Flakes only takes files into account if they are either in the git tree. You don't necessarily have to commit the files, adding them in the git staging area with `git add` is enough.

Pure evaluation

Because the evaluation in Flakes is “pure”, a few things are disabled.

Pure evaluation can also be enabled by using `--option pure-eval true` on standard nix CLIs. Eg:

```
$ nix-instantiate --option pure-eval true  
--eval --expr '(builtins.currentTime)'  
error: --- EvalError  
-----  
nix-instantiate  
at: (1:2) from string  
  
  1| (builtins.currentTime)  
   | ^  
  
attribute 'currentTime' missing
```



To find these out I searched for `evalSettings.pureEval` in

the “src/libexpr” folder of the Nix repo.

All these builtins are not defined in pure evaluation:

- `builtins.currentTime -> int`
- `builtins.currentSystem -> str`

Some more special behaviours:

- `builtins.getEnv str -> str` returns empty strings.
- `builtins.storePath` throws '`__storePath`' is not allowed in pure evaluation mode
- `builtins.filterSource` and `builtins.path` has some condition in it, I don't know exactly which.
- `builtins.fetchTree` also has some conditions.
- `<foo>` throws cannot look up '`<foo>`' in pure evaluation mode (use '`--impure`' to override)

Running unfree packages

Because Flakes are pure by default, something like `nix run nixpkgs#steam` will complain that it's unfree, even if `NIXPKGS_ALLOW_UNFREE=1` is set in the environment.

The workaround is to disable the pure evaluation with the `--impure` flag like so:



```
NIXPKGS_ALLOW_UNFREE=1 nix run --impure  
nixpkgs#steam
```



[Xe](#)[Blog](#)[Contact](#)[Resume](#)[Talks](#)[VODs](#)[Signalboost](#)

Nix Flakes: an Introduction

Published on 02/21/2022, 2771 words, 11 minutes to read

Nix is a package manager that lets you have a more deterministic view of your software dependencies and build processes. One of its biggest weaknesses out of the box is that there are very few conventions on how projects using Nix should work together. It's like having a build system but also having to configure systems to run software yourself. This could mean copying a NixOS module out of the project's git repo, writing your own or more. In contrast to this, [Nix flakes](#) define a set of conventions for how software can be built, run, integrated and deployed without having to rely on external tools such as [Niv](#) or [Lorri](#) to help you do basic tasks in a timely manner.

This is going to be a series of posts that will build on each other. This post will be an introduction to Nix flakes and serve as a "why should I care?" style overview of what you can do with flakes without going into too much detail. Most of these will get separate posts (some more than one post).

In my opinion, here are some of the big reasons you should care about Nix flakes:

- Flakes adds project templates to Nix
- Flakes define a standard way to say "this is a program you can run"
- Flakes consolidate development environments into project configuration
- Flakes can pull in dependencies from outside git repos trivially
- Flakes can work with people that don't use flakes too
- Flakes supports using private git repos
- Flakes let you define system configuration alongside your application code



- Flakes let you embed the git hash of your configurations repository into machines you deploy



<**Mara**> Something that may also help you understand why flakes matter is that Nix by itself is more akin to Dockerfiles. Dockerfiles help you build the software, but they don't really help you run or operate the software. Nix flakes is more akin to docker-compose, they help you compose packages written in Nix to run across machines.

Project Templates

One of the big annoying parts about getting into Nix is that setting up projects isn't totally a defined science. Nix configurations just tend to grow organically and can easily become weird or difficult to understand for people that didn't start the project. Nix flakes helps fix this by doing a few things:

1. Defining a `flake.nix` as the central "hub" for your project's dependencies, exposed packages, NixOS configuration modules and more.
2. Shipping a set of templates so that you can get projects started easily. Think something like Yeoman but built directly into Nix. You can also define your own templates in your `flake.nix`.

As an example that we will use for the rest of this post to help explain it, let's make a Go project with their Go template. First you will need to enable Nix flakes on your machine. If you are using NixOS, add this to your `configuration.nix` file:

```
nix = {  
  package = pkgs.nixFlakes;  
  extraOptions = ''  
    experimental-features = nix-command flakes  
  '';  
};
```



Then rebuild your system and you can continue along with the article.



<**Mara**> EDIT: You can use WSL for this. See [here](#) for more information.

If you are not on NixOS, you will need to either edit `~/.config/nix/nix.conf` or `/etc/nix/nix.conf` and add the following line to it:

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



<**Mara**> You may need to restart the Nix daemon here with `sudo systemctl restart nix-daemon.service`, but if you are unsure how Nix was set up on that non-NixOS machine feel free to totally restart your computer.

Now go to a temporary folder and run these commands to make a folder and create a new flake from a template:

```
mkdir ~/tmp/go-demo  
cd ~/tmp/go-demo  
nix flake new -t templates#go-hello .  
git init && git add .
```

This will create a few files in the folder:

```
$ ls  
flake.lock  flake.nix  go.mod  main.go
```

Then you can look at `flake.nix` to see what's up:

```
{  
  description = "A simple Go package";  
  
  # Nixpkgs / NixOS version to use.  
  inputs.nixpkgs.url = "nixpkgs/nixos-21.11";
```



```
outputs = { self, nixpkgs }:
let

# Generate a user-friendly version number.
version = builtins.substring 0 8 self.lastModifiedDate;

# System types to support.
supportedSystems = [ "x86_64-linux" "x86_64-darwin" "aarch64-linux" "aarch64-darwin" ];

# Helper function to generate an attrset '{ x86_64-linux = f "x86_64-linux"; ... }'.
forAllSystems = nixpkgs.lib.genAttrs supportedSystems;

# Nixpkgs instantiated for supported system types.
nixpkgsFor = forAllSystems (system: import nixpkgs { inherit system; });

in
{

# Provide some binary packages for selected system types.
packages = forAllSystems (system:
let
  pkgs = nixpkgsFor.${system};
  in
  {
    # The default package for 'nix build'. This makes sense if the
    # flake provides only one package or there is a clear "main"
    # package.
    default = pkgs.buildGoModule {
      pname = "go-hello";
      inherit version;
      # In 'nix develop', we don't need a copy of the source tree
      # in the Nix store.
      src = ./;

      # This hash locks the dependencies of this package. It is
      # necessary because of how Go requires network access to resolve
      # VCS. See https://www.tweag.io/blog/2021-03-04-gomod2nix/ for
      # details. Normally one can build with a fake sha256 and rely on native Go
      # mechanisms to tell you what the hash should be or determine what
      # it should be "out-of-band" with other tooling (eg. gomod2nix).
      # To begin with it is recommended to set this, but one must
      # remember to bump this hash when your dependencies change.
      #vendorSha256 = pkgs.lib.fakeSha256;

      vendorSha256 = "sha256-pQpattmS9Vm03ZIQUFn66az8GSmB4IvYhTTCFn6SUmo=";
    };
  };
};

}
```



```
};  
});  
};  
}
```

This defines a single Go package that is supported on macOS and Linux for 64 bit x86 processors and 64 bit ARM processors.



<Mara> In practice this spread should cover all of the main targets you'll need to care about for local development and cloud deployment.

You can then build the flake with `nix build`:

```
$ nix build
```

And then run it:

```
$ ./result/bin/go-hello  
Hello Nix!
```

Standard Default Package

Let's take a closer look at the higher level things in the flake:

```
{  
  description = "A simple Go package";  
  
  inputs.nixpkgs.url = "nixpkgs/nixos-21.11";  
  
  outputs = { self, nixpkgs }: {  
    packages = { ... };  
  };  
}
```





<**Cadey**> A note: in the rest of this article (and series of articles), when I refer to a "flake output", I am referring to an attribute in the `outputs` attribute of your `flake.nix`. Ditto with "flake input" referring to the `inputs` attribute of your `flake.nix`.

When you ran `nix build` earlier, it defaulted to building the `default` entry in `packages`. You can also build the `default` package by running this command:

```
$ nix build .#default
```

And if you want to build the copy I made for this post:

```
$ nix build github:Xe/gohello
$ ./result/bin/go-hello
Hello reader!
```

A standard default package means that you can more easily build software without having to read documentation on what file to build. `nix build` will Just Work™.

Exposing Packages as Applications

Additionally, you can expose a package as an application. This allows you to simplify that above `nix build` and `./result/bin/go-hello` cycle into a single `nix run` command. Open `flake.nix` in your favorite editor and let's configure `go-hello` to be the default app:

```
# below packages
```

```
apps = forAllSystems (system: {
  default = {
    type = "app";
    program = "${self.packages.${system}.default}/bin/go-hello";
  };
});
```



Then you can run it with `nix run`:

```
$ nix run  
Hello Nix!
```

Or you can run my copy:

```
$ nix run github:Xe/gohello/main  
Hello reader!
```



<Mara> What is that extra part of the URL path for? Is that a git branch?



<Cadey> Yes, you can use that syntax to set the git branch that Nix should build from. By default it will use the default branch (typically `main`), but sometimes you need to specify a branch or commit directly.

Development Environment Configuration

One of Nix's superpowers is the ability to declaratively manage the development environment for a project so that you can be sure that everyone working on the project is using the same tools.



<Cadey> I use this with all of my projects to the point that when I am outside of a project folder I do not have any development tools available.



Flakes has the ability to specify this using the `devShells` flake output. You can add it to your `flake.nix` using this:

```
# after apps
```

```
devShells = forAllSystems (system:  
  let pkgs = nixpkgsFor.${system};  
  in {  
    default = pkgs.mkShell {  
      buildInputs = with pkgs; [ go gopls gotools go-tools ];  
    };  
  });
```



<Mara> We consider this to be a basic Go development environment. It includes standard tools such as the language server, `goimports` for better formatting and tools like staticcheck. If you use staticcheck regularly at work, please consider throwing Dominik a couple bucks a month if you find it useful. It helps the project be more self-sustaining.

Then you can enter the development shell with `nix develop`:

```
$ nix develop
```

```
[cadey@pneuma:~/tmp/gohello]$ go version  
go version go1.16.9 linux/amd64
```

And then hack at your project all you want. You can send this git repo to a friend and they will have the same setup.

External Dependencies

Now let's talk about inputs. Flake inputs let you add external dependencies to a project. As an example, let's look at the `nixpkgs` input:



```
# Nixpkgs / NixOS version to use.  
inputs.nixpkgs.url = "nixpkgs/nixos-21.11";
```

This defines the release of nixpkgs that should be used for the project. This template defaults to NixOS 21.11's version of nixpkgs, however we can upgrade it to nixos-unstable

by changing it to this:

```
# Nixpkgs / NixOS version to use.  
inputs.nixpkgs.url = "nixpkgs/nixos-unstable";
```

Then we can run `nix flake update` and then `nix develop` and see that we are running a newer version of Go:

```
$ nix flake update  
warning: updating lock file '/home/cadey/tmp/gohello/flake.lock':  
• Updated input 'nixpkgs':  
  'github:NixOS/nixpkgs/77aa71f66fd05d9e7b7d1c084865d703a8008ab7' (2022-01-19)  
  → 'github:NixOS/nixpkgs/2128d0aa28edef51fd8fef38b132ffc0155595df' (2022-02-16)
```

```
$ nix develop
```

```
[cadey@pneuma:~/tmp/gohello]$ go version  
go version go1.17.7 linux/amd64
```

This also lets you pull in other Nix flakes projects, such as my CSS framework Xess:

```
inputs.xess.url = "github:Xe/Xess";  
inputs.xess.inputs.nixpkgs.follows = "nixpkgs";
```



<Mara> Why is that second line needed?



<Cadey> By default when you pull in another project with Nix flakes, it treats that project as an entirely separate universe and only interacts with the outputs of that flake. This means it pulls in its own version of nixpkgs, each dependency it has can pull in that own version of nixpkgs and vice versa ad infinitum. By making Xess' nixpkgs input follows our own one, we are saying "I understand this may be incompatible, but please use this version of nixpkgs instead". This can help larger projects with many inputs



(such as a nixos configs repo made by someone with too many throwaway side projects) evaluate and build faster. Nix flakes does have a cached evaluator, but still it helps to avoid the problem in the first place.

Or anything you want! A useful library to pull in is [flake-utils](#), that can help you simplify your `flake.nix` and get rid of those ugly `forAllSystems` and `nixpkgsFor` functions in the `flake.nix` that this post used by default. For an example of a flake that uses this library, see [this `flake.nix`](#) from the IRC bot that lives in `#xeserv`.



<[Mara](#)> Adapting this trivial example to use `flake-utils` is an excellent exercise for the reader! This library should really be shipped with flakes by default.

Backwards Compatibility

Normally you need to enable Nix flakes in your Nix daemon to take advantage of them. This is great for when you can do that, but sometimes you'll need to make things work for people without flakes enabled. This could happen when needing to graft in a Nix flakes project to one without flakes enabled. There is a library called [flake-compat](#) that makes this easy.

Create `default.nix` with the following contents:

```
(import (
  fetchTarball {
    url = "https://github.com/edolstra/flake-compat/archive/99f1c2157fba4bfe6211a321fd0ee43199";
    sha256 = "0x2jn3vrawwv9xp15674wjz9pixwjyj3j771izayl962zzivbx2"; }
) {
  src = ./;
}).defaultNix
```



And `shell.nix` with the following contents:

```
(import (
```

```
fetchTarball {  
    url = "https://github.com/edolstra/flake-compat/archive/99f1c2157fba4bfe6211a321fd0ee43199";  
    sha256 = "0x2jn3vrawwv9xp15674wjz9pixwjyj3j771izayl962zzivbx2"; }  
}  
src = ./;  
}).shellNix
```

Then you can use `nix-build` and `nix-shell` like you have in other Nix projects.

Private Git Repos

Nix flakes has native support for private git repositories as inputs. This can be useful when trying to build software you don't want to release as open to the world. To use a private repo, your flake input URL should look something like this:

```
git+ssh://git@github.com/user/repo?ref=main
```



<Cadey> I'm pretty sure you could use private git repos outside of flakes, however it was never really clear to me *how* you end up doing it.



<Mara> I'm told you can bash Niv into shape enough to do this, but yeah it's never really been clear how you do this.

Embed NixOS Modules in Flakes

The biggest ticket item for me is that it lets you embed NixOS modules in flakes themselves. This lets you define the system configuration for software right next to where the software is defined, thus shipping it as a unit. Using this you can make installing software a matter of adding it to your system's flake, adding the module and then enabling the settings you want to enable.

As an example, here is the NixOS module for that IRC bot I mentioned:



```
nixosModules.bot = { config, lib, ... }: {
  options.within.services.mara-bot.enable =
    lib.mkEnableOption "enable Mara bot";

  config = lib.mkIf config.within.services.mara-bot.enable {
    users.groups.mara-bot = { };

    users.users.mara-bot = {
      createHome = true;
      isSystemUser = true;
      home = "/var/lib/mara-bot";
      group = "mara-bot";
    };
  };

  systemd.services.mara-bot = {
    wantedBy = [ "multi-user.target" ];
    environment.RUST_LOG = "tower_http=debug,info";
    unitConfig.ConditionPathExists = "/var/lib/mara-bot/config.yaml";
    serviceConfig = {
      User = "mara-bot";
      Group = "mara-bot";
      Restart = "always";
      WorkingDirectory = "/var/lib/mara-bot";
      ExecStart = "${self.packages."${system}.default}/bin/mara";
    };
  };
};

};

};
```

The key important part here is the `ExecStart` line. It points back to the flake's default package (which is hopefully where the bot's code is defined), and then has systemd manage that.

I plan to use this to radically simplify my nixos-configs repo. Right now it has a lot of code that is very project-specific and if I can move that into the projects in question, I can eliminate a lot of code out of the core of my configs repo.



Embedding Configuration Git Hash into Systems

Finally, Nix flakes lets you see the configuration version of a system by embedding it at the

build step. Normally NixOS lets you see the following information with `nixos-version --json`:

```
{  
  "nixosVersion": "22.05pre348581.c07b471b52b",  
  "nixpkgsRevision": "c07b471b52be8fbc49a7dc194e9b37a6e19ee04d"  
}
```

You have the NixOS version and the nixpkgs hash. That doesn't tell you what configuration you are running or anything about it though. However with flakes you can embed the git hash of your configuration into the system config:

```
{  
  "configurationRevision": "f53891121ce4204f57409cbe9e6fce3b030a350",  
  "nixosVersion": "22.05.20220210.48d63e9",  
  "nixpkgsRevision": "48d63e924a2666baf37f4f14a18f19347fdb54a2"  
}
```

This can let you make a URL pointing to the commit in that output:

```
$ echo "https://tulpa.dev/cadey/nixos-configs/src/commit/$(ssh logos nixos-version --json | jc
```

Which will spit out a link to [cadey/nixos-configs@f53891121](https://tulpa.dev/cadey/nixos-configs@f53891121).

I'll cover more on how to do this in the NixOS deployment post.

There is a lot more to get into with each of these topics. I'm only really giving a very  high level overview on them while I learn more and migrate over my NixOS configurations to flakes piecemeal. This has also given me the opportunity to clean things up and chew out a lot of the fat from my NixOS configurations. More to come when it is ready.

[Share](#)

Facts and circumstances may have changed since publication. Please contact me before jumping to conclusions if something seems wrong or unclear.

Tags: nix, nixos

Copyright 2012-2024 Xe Iaso (Christine Dodrill). Any and all opinions listed here are my own and not representative of any of my employers, past, future, and/or present.

Like what you see? Donate on [Patreon](#) like [these awesome people!](#)

Served by xesite v4 (/nix/store/9rxy3y9y9ffwx6xdvb56pgciar4b5138-xesite_v4-20240128/bin/xesite) with site version [a1841325](#), source code available [here](#).



[Xe](#)[Blog](#)[Contact](#)[Resume](#)[Talks](#)[VODs](#)[Signalboost](#)

Nix Flakes: Exposing and using NixOS Modules

Published on 04/07/2022, 2100 words, 8 minutes to read

Nix flakes allow you to expose NixOS modules. NixOS modules are templates for system configuration and they are the basis of how you configure NixOS. Today we're going to take our Nix flake [from the last article](#) and write a NixOS module for it so that we can deploy it to a container running locally. In the next post we will deploy this to a server.



<Mara> If you haven't read [the other articles in this series](#), you probably should. This article builds upon the previous ones.

NixOS modules are building blocks that let you configure NixOS servers. Modules expose customizable options that expand out into system configuration. Individually, each module is fairly standalone and self-contained, but they build up together into your server configuration like a bunch of legos build into a house. Each module describes a subset of your desired system configuration and any options relevant to that configuration.



<Mara> You can think about them like Ansible playbooks, but NixOS modules describe the desired end state instead of the steps you need to get to that end state. It's the end result of evaluating all of your options against all of the modules that you use in your configuration.



NixOS modules are functions that take in the current state of the system and then return things to add to the state of the system. Here is a basic NixOS module that enables [nginx](#):

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

{
  config = {
    services.nginx.enable = true;
  };
}
```

This function takes in the state of the world and returns additions to the state of the world. This will use the nginx module that ships with NixOS to give you a basic nginx setup that has the upstream default configuration in it.

NixOS has a way to run other instances of NixOS with [NixOS containers](#). We can use them to test our NixOS module as we write it.

This probably won't work on a non-NixOS machine. You will need to install NixOS in order to test this. For an easy way to do this, see [nixos-infect](#), a script you can put into a cloudconfig when spinning up a new server. You can also [install NixOS manually](#) in a VM, but for now it may be better to use a cloud server as the path of least resistance. Installing NixOS with a flake will be a part of a future article in this series.

In Nix you can merge two attribute sets using the `//` operator. This allows you to add two attribute sets into one larger one, such as like this:

```
nix-repl> { foo = 1; } // { bar = 2; }
{ bar = 2; foo = 1; }
```

<Mara> Important pro tip: the merge operator is NOT recursive. If you try to do something like:

```
nix-repl> foo = { bar = { baz = "foo"; }; }
nix-repl> (foo // { bar = { spam = "eggs"; }; }).bar
```

You will get:





```
{ spam = "eggs"; }
```

And not:

```
{ baz = "foo"; spam = "eggs"; }
```

This is because the `//` operator prefers things in the right hand side over the left hand side if both conflict. To recursively merge two attribute sets (using all elements from both sides), use lib.recursiveUpdate:

```
nix-repl> (pkgs.lib.recursiveUpdate foo bar).bar
{ baz = "foo"; spam = "eggs"; }
```

We will use this to add the container configuration to the flake at the end of the flake.nix file. We need to do this because the upper part of the flake with the `forAllSystems` call will generate a bunch of system-specific attributes for each system we support. NixOS configurations don't support this level of granularity.

At the end of your flake.nix (just before the final closing `}`), there should be a line that looks like this:

```
});
```

This is what terminates the `outputs` declaration from all the way at the top. In order to add the container configuration, you should change this to look like this:

```
}) // {
};
```



Then we can add the container configuration to the flake:

```
}) // {
  nixosConfigurations.container = nixpkgs.lib.nixosSystem {
    system = "x86_64-linux";
    modules = [
      ({pkgs, ...}: {
        # Only allow this to boot as a container
        boot.isContainer = true;
        networking.hostName = "gohello";

        # Allow nginx through the firewall
        networking.firewall.allowedTCPPorts = [ 80 ];

        services.nginx.enable = true;
      })
    ];
  };
};
```

This will create a container (with the hostname "gohello") that starts nginx and allows traffic to go to nginx on TCP port 80. You can start up the container with the `nixos-container` command:

```
$ sudo nixos-container create gohello --flake .#container
host IP is 10.233.1.1, container IP is 10.233.1.2
```

Then you can start the container with this command:

```
$ sudo nixos-container start gohello
```

And then we can try to connect to nginx to see if it's working:

```
$ curl http://10.233.1.2
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```



```
body {}  
    width: 35em;  
    margin: 0 auto;  
    font-family: Tahoma, Verdana, Arial, sans-serif;  
}  
</style>  
</head>  
<body>  
<h1>Welcome to nginx!</h1>  
<p>If you see this page, the nginx web server is successfully installed and  
working. Further configuration is required.</p>  
  
<p>For online documentation and support please refer to  
<a href="http://nginx.org/">nginx.org</a>.<br/>  
Commercial support is available at  
<a href="http://nginx.com/">nginx.com</a>.</p>  
  
<p><em>Thank you for using nginx.</em></p>  
</body>  
</html>
```

We have nginx!

Now that we have our container to test with, let's write the configuration for the service. At a basic level we need the following things:

- A systemd unit for orchestrating the HTTP server process
- nginx configuration to reverse proxy to that HTTP server

Above the container definition, add this basic NixOS module template:

```
nixosModule = { config, lib, pkgs, ... }:  
  with lib;  
  let cfg = config.xeserv.services.gohello;  
  in {  
    options.xeserv.services.gohello = {  
      enable = mkEnableOption "Enables the gohello HTTP service";  
    };
```



```
config = mkIf cfg.enable {  
};  
};
```

This will create a NixOS module that will only be enabled when the configuration setting `xeserv.services.gohello.enable` is set to `true`. Everything else we do here will build on this.



<Mara> You can and probably do want to change the namespace `xeserv` here, it is a placeholder that is not likely to conflict with anything else.

Create a basic systemd service with this template:

```
config = mkIf cfg.enable {  
    systemd.services."xeserv.gohello" = {  
        wantedBy = [ "multi-user.target" ];  
  
        serviceConfig = let pkg = self.packages.${system}.default;  
        in {  
            Restart = "on-failure";  
            ExecStart = "${pkg}/bin/web-server";  
            DynamicUser = "yes";  
            RuntimeDirectory = "xeserv.gohello";  
            RuntimeDirectoryMode = "0755";  
            StateDirectory = "xeserv.gohello";  
            StateDirectoryMode = "0700";  
            CacheDirectory = "xeserv.gohello";  
            CacheDirectoryMode = "0750";  
        };  
    };  
};
```



<Mara> NOTE: If you have been following along since before this article was published, you will want to be sure to do the following things to your copy of gohello:

- Move the definition of `defaultPackage` into the `packages` attribute set with the name `default`





- Update `defaultApp` and the other entries to point to `self.packages.\${system}.default` instead of `self.defaultPackage.\${system}`

We have updated previous articles and the template accordingly. Annoyingly it seems that this change is new enough that it isn't totally documented on the NixOS wiki. We are working on fixing this.

This will do the following things:

- Start the service on boot (`multi-user.target` fires once the system is "fully booted" and the network is active)
- Automatically restarts the service when it crashes
- Starts our `web-server` binary when running the service
- Creates a random, unique user account for the service (see [here](#) for more information on how/why this works)
- Creates temporary, home and cache directories for the service, makes sure that random user has permission to use them (with the specified directory modes too)
- Enables the service automatically

Then you need to add the nginx configuration. We want this application to have its own virtual host, so we will need to add that as a configuration option under the `enable` option:

```
domain = mkOption rec {  
    type = types.str;  
    default = "gohello.local.cetacean.club";  
    example = default;  
    description = "The domain name for gohello";  
};
```



<Mara> Pro tip: `anything.local.cetacean.club` points to `127.0.0.1`. You can use this when testing things.



And then we can add the nginx configuration under the systemd service definition:

```
services.nginx.virtualHosts.${cfg.domain} = {
  locations."/" = { proxyPass = "http://127.0.0.1:3031"; };
};
```

Your module should look like this:

```
nixosModule = { config, lib, pkgs, ... }:
  with lib;
  let cfg = config.xeserv.services.gohello;
  in {
    options.xeserv.services.gohello = {
      enable = mkEnableOption "Enables the gohello HTTP service";

      domain = mkOption rec {
        type = types.str;
        default = "gohello.local.cetacean.club";
        example = default;
        description = "The domain name for gohello";
      };
    };

    config = mkIf cfg.enable {
      systemd.services."xeserv.gohello" = {
        wantedBy = [ "multi-user.target" ];

        serviceConfig = let pkg = self.packages.${pkgs.system}.default;
        in {
          Restart = "on-failure";
          ExecStart = "${pkg}/bin/web-server";
          DynamicUser = "yes";
          RuntimeDirectory = "xeserv.gohello";
          RuntimeDirectoryMode = "0755";
          StateDirectory = "xeserv.gohello";
          StateDirectoryMode = "0700";
          CacheDirectory = "xeserv.gohello";
          CacheDirectoryMode = "0750";
        };
      };
    };
  };
};
```



```
services.nginx.virtualHosts.${cfg.domain} = {
  locations."/" = { proxyPass = "http://127.0.0.1:3031"; };
};
```



<Mara> The service name is overly defensive. It's intended to avoid conflicting with any other unit on the system named `gohello.service`. Feel free to remove this part, it is really just defensive devops by design to avoid name conflicts.

Then you can add it to the container by importing our new module in its configuration and activating the gohello service:

```
nixosConfigurations.container = nixpkgs.lib.nixosSystem {
  system = "x86_64-linux";
  modules = [
    self.nixosModule
    ({ pkgs, ... }: {
      # Only allow this to boot as a container
      boot.isContainer = true;

      # Allow nginx through the firewall
      networking.firewall.allowedTCPPorts = [ 80 ];

      services.nginx.enable = true;

      xeserv.services.gohello.enable = true;
    })
  ];
};
```

Then you can update the container's configuration with this command:

```
$ sudo nixos-container update gohello --flake .#container
reloading container...
```



And finally make a request to the gohello service running in that container:

```
$ curl http://10.233.1.2 -H "Host: gohello.local.cetacean.club"  
hello world :)
```



<Mara> Exercises for the reader:

Try adding a nixos option that correlates to the `--bind` flag that `gohello` uses as the TCP address to serve HTTP from. You will want to have the type be `types.port`. If you are stuck, see here for inspiration.

Also try adding `AmbientCapabilities = "CAP_NET_BIND_SERVICE"` and `CapabilityBoundingSet = "CAP_NET_BIND_SERVICE"` to your `serviceConfig` and bind `gohello` to port 80 without nginx involved at all.

You can delete this container with `sudo nixos-container destroy gohello` when you are done with it.

These are the basics on how to use NixOS modules. Everything else you can do with them builds off of these fundamental ideas. Modules are templates that coordinate packages and configuration into your desired system state. Containers can let you test out modules without having to add them to your currently running system. Modules declare options and emit configuration based on those options.

You can also consume NixOS modules from flakes using the input system, however I will go into more details about this at a later date. If you want more examples of NixOS modules, I would suggest checking out my nixos-configs repository. I have nearly everything neatly modularized and configurable. If you see anything in there that is confusing to you, please reach out and ask. I am happy to answer your questions and your feedback will help me write future posts in this series.



I also have my "next generation" flakes-based configuration experiments here if you want to read through those. I have still been porting over things piecemeal, so it is not a complete replica of my existing configuration.

Next time I will cover how to install NixOS to a server and deploy system configurations

using [deploy-rs](#). This will allow you to have your workstation build configuration for your servers and push out all the changes from there.

Many thanks to Open Skies for being my fearless editor that helps make these things shine.

In part of this post I use my new Xexact-powered HTML component for some of the conversation fragments, but the sizing was off on my iPhone when I tested it. If you know what I am doing wrong, please [get in touch](#).



Facts and circumstances may have changed since publication. Please contact me before jumping to conclusions if something seems wrong or unclear.

Tags: nixos

Copyright 2012-2024 Xe Iaso (Christine Dodrill). Any and all opinions listed here are my own and not representative of any of my employers, past, future, and/or present.

Like what you see? Donate on [Patreon](#) like [these awesome people!](#)

Served by xesite v4 (/nix/store/9rxy3y9y9ffwx6xdvb56pgciar4b5138-xesite_v4-20240128/bin/xesite) with site version [a1841325](#), source code available [here](#).



[Xe](#)[Blog](#)[Contact](#)[Resume](#)[Talks](#)[VODs](#)[Signalboost](#)

Nix Flakes on WSL

Published on 05/01/2022, 1489 words, 6 minutes to read

About five years ago, Microsoft released the Windows Subsystem for Linux ([WSL](#)). This allows you to run Linux programs on a Windows machine. When they released WSL version 2 in 2019, this added support for things like Docker and systemd. As a result, this is enough to run NixOS on Windows.



<Mara> This will give you an environment to run Nix and Nix Flakes commands with. You can use this to follow along with this series without having to install NixOS on a VM or cloud server. This is going to retread a bunch of ground from the first article. If you have been following along through this entire series, once you get to the point where you convert the install to flakes there isn't much more new material here.

Installation

Head to the NixOS-WSL [CI feed](#) and download the `installer.zip` file to your Downloads folder. Extract the `nixos-wsl-installer.tar.gz` file from `installer.zip`. Do not extract `nixos-wsl-installer.tar.gz`, the `wsl` command will for you.

Then open Powershell and make a folder called `WSL`:

```
New-Item -Path .\WSL -ItemType Directory
```



<Mara> It's worth noting that Powershell does have a bunch of aliases for



common coreutils commands to the appropriate Powershell CMDlets. However these aliases are **NOT** flag-compatible and use the Powershell semantics instead of the semantics of the command it is aliasing. This will bite you when you use commands like `wget` out of instinct to download things. In order to avoid your muscle memory betraying you, the Powershell CMDlets are shown here in their full overly verbose glory.

Then enter the directory with `Set-Location`:

```
Set-Location -Path .\WSL
```



<Mara> This directory is where the NixOS root filesystem will live. If you want to put this somewhere else, feel free to. Somewhere in `%APPDATA%` will work, just as long as it's on an NTFS volume somewhere.

Make a folder for the NixOS filesystem:

```
New-Item -Path .\NixOS -ItemType Directory
```

Then install the NixOS root image with the `wsl` command:

```
wsl --import NixOS .\NixOS\ ..\Downloads\nixos-wsl-installer-fixed.tar.gz --version 2
```

And start NixOS once to have it install itself:

```
wsl -d NixOS
```

Once that finishes, press control-D (or use the `exit` command) to exit out of NixOS and restart the WSL virtual machine:



```
exit  
wsl --shutdown  
wsl -d NixOS
```

And then you have yourself a working NixOS environment! It's very barebones, but we can use it to test the `nix run` command against our gohello command:

```
$ nix run github:Xe/gohttp  
Hello reader!
```

Local Services

We can also use this NixOS environment to run a local nginx server. Open `/etc/nixos/configuration.nix`:



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

with lib;
let
  nixos-wsl = import ./nixos-wsl;
in
{
  imports = [
    "${modulesPath}/profiles/minimal.nix"

    nixos-wsl.nixosModules.wsl
  ];

  wsl = {
    enable = true;
    automountPath = "/mnt";
    defaultUser = "nixos";
    startMenuLaunchers = true;

    # Enable integration with Docker Desktop (needs to be installed)
    # docker.enable = true;
  };

  # Enable nix flakes
  nix.package = pkgs.nixFlakes;
  nix.extraOptions = ''
    experimental-features = nix-command flakes
  '';
}
}
```

Right after the `wsl` block, add this nginx configuration to the file:

```
services.nginx.enable = true;
services.nginx.virtualHosts."test.local.cetacean.club" = {
  root = "/srv/http/test.local.cetacean.club";
};
```



This will create an nginx configuration that points the domain `test.local.cetacean.club` to the contents of the folder `/srv/http/test.local.cetacean.club`.

<Mara> The `/srv` folder is set aside for site-specific data, which is code for



"do whatever you want with this folder". In many cases people make a separate `/srv/http` folder and put each static subdomain in its own folder under that, however I am also told that it is idiomatic to put stuff in `/var/www`. Pick your poison.

Then you can test the web server with the `curl` command:

```
$ curl http://test.local.cetacean.club
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

This is good! Nginx is running and since we haven't created the folder with our website content yet, this 404 means that it can't find it! Let's create the folder so that nginx has permission to it and we can modify things in it:

```
sudo mkdir -p /srv/http/test.local.cetacean.club
sudo chown nixos:nginx /srv/http/test.local.cetacean.club
```

Finally we can make an amazing website. Open `/srv/http/test.local.cetacean.club/index.html` in nano:

```
nano /srv/http/test.local.cetacean.club/index.html
```



And paste in this HTML:

```
<title>amazing website xD</title>
<h1>look at my AMAZING WEBSITE</h1>
It's so cool *twerks*
```



<Mara> This doesn't have to just be artisanal handcrafted HTML in bespoke folders either. You can set the `root` of a nginx virtual host to point to a Nix package as well. This will allow you to automatically generate your website somehow and deploy it with the rest of the system. Including being able to roll back changes.

And then you can see it show up with `curl`:

```
$ curl http://test.local.cetacean.club
<title>amazing website xD</title>
<h1>look at my AMAZING WEBSITE</h1>
It's so cool *twerks*
```

You can also check this out in a browser:



look at my AMAZING WEBSITE

It's so cool *twerks*

Installing `gohello`

To install the `gohello` service, first we will need to convert this machine to use NixOS flakes. We can do that really quick and easy by adding this file to `/etc/nixos/flake.nix`:



<Mara> Do this as root!



```
{
  inputs = {
```

```
nixpkgs.url = "nixpkgs/nixos-unstable";  
};  
  
outputs = { self, nixpkgs, ... }: {  
    nixosConfigurations.nixos = nixpkgs.lib.nixosSystem {  
        system = "x86_64-linux";  
        modules = [  
            ./configuration.nix  
  
            # add things here  
        ];  
    };  
};  
}
```

Then run `nix flake check` to make sure everything is okay:

```
sudo nix flake check /etc/nixos
```

And finally activate the new configuration with flakes:

```
sudo nixos-rebuild switch
```



<Mara> Why don't you have the `--flake` flag here? Based on what I read in the documentation, I thought you had to have it there.



<Cadey> `nixos-rebuild` will automatically detect flakes in `/etc/nixos`. The only major thing it cares about is the hostname matching. If you want to customize the hostname of the WSL VM, change the `nixos` in `nixosConfigurations.nixos` above and set `networking.hostName` to the value you want to use. To use flakes explicitly, pass `--flake /etc/nixos#hostname` to your `nixos-rebuild` call.



After it thinks for a bit, you should notice that nothing happened. This is good, we have just converted the system over to using Nix flakes instead of the classic `nix-channel`

rebuild method.

To get `gohello` in the system, first we need to add `git` to the commands available on the system in `configuration.nix`:

```
environment.systemPackages = with pkgs; [ git ];
```

Then we can add `gohello` to our system flake:

```
{
  inputs = {
    nixpkgs.url = "nixpkgs/nixos-unstable";
    # XXX(Xe): this URL may change for you, such as github:Xe/gohello-http
    gohello.url = "git+https://tulpa.dev/cadey/gohello-http?ref=main";
  };

  outputs = { self, nixpkgs, gohello, ... }: {
    nixosConfigurations.nixos = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        ./configuration.nix

        # add things here
        gohello.nixosModule
        ({ pkgs, ... }: {
          xeserv.services.gohello.enable = true;
        })
      ];
    };
  };
}
```



<Mara> The block of code under `gohello.nixosModule` is an inline NixOS module. If we put `gohello.nixosModule` before the `./configuration.nix` reference, we could put the `xeserv.services.gohello.enable = true;` line inside `./configuration.nix`. This is an exercise for the reader.



And rebuild the system with `gohello` enabled:

```
sudo nixos-rebuild switch
```

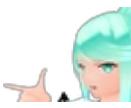
Finally, poke it with `curl`:

```
$ curl http://gohello.local.cetacean.club  
hello world :)
```

To update it, update the flake inputs in `/etc/nixos` and run `nixos-rebuild`:

```
sudo nix flake update /etc/nixos  
sudo nixos-rebuild switch
```

And from here you can do whatever you want with NixOS. You can use containers, set up arbitrary services, or plan for world domination as normal.



<Numa> I thought it was "to save the world from devastation", not "to plan for world domination". Who needs a monopoly on violence for world domination when you have Nix expressions?



<Cadey> Siiiiiiiiiiiiiiigh.

I will use this setup in future posts to make this more accessible and easy to hack at without having to have a dedicated NixOS machine laying around.



[Share](#)

Facts and circumstances may have changed since publication. Please contact me before jumping to conclusions if something seems wrong or unclear.

Tags: nixos, wsl

Copyright 2012-2024 Xe Iaso (Christine Dodrill). Any and all opinions listed here are my own and not representative of any of my employers, past, future, and/or present.

Like what you see? Donate on [Patreon](#) like [these awesome people!](#)

Served by xesite v4 (/nix/store/9rxy3y9y9ffwx6xdvb56pgciar4b5138-xesite_v4-20240128/bin/xesite) with site version [a1841325](#) , source code available [here](#).



[Xe](#)[Blog](#)[Contact](#)[Resume](#)[Talks](#)[VODs](#)[Signalboost](#)

Nix Flakes: Packages and How to Use Them

Published on 02/27/2022, 2903 words, 11 minutes to read



<Cadey> Nix flakes are still marked as experimental. This documentation has a small chance of bitrotting. I will make every attempt to update it if things change, however flakes have been fairly consistent for a few years now.

EDIT(20220327 14:13): A previous version of this article said to use `defaultPackage` for the default package. This is deprecated and you should use `packages.default` instead.



<Mara> What is a package? I've seen this term thrown around with phrases like "Nix is a package manager" or "language-specific package manager" or even "download the debian package and install it", but it's not really clear to me what a package is. What is a package?

A package is a bundle of files. These files could be program executables, resources such as stylesheets or images, or even a container image. Most of the time you don't deal with packages directly and instead you use a *package manager* (a program whose sole goal in life is to deal with packages) to do actions for you. This post is going to cover how to define packages in Nix and how Nix flakes let you manage multiple packages per project more easily.



What is a Package?

In Nix, you build packages by creating *derivations* that define the build steps and

associated inputs (such as the compiler) to end up with the resulting outputs (derivation being the product of deriving something). Consider a package like this:

```
# hello-shell.nix
with import <nixpkgs> {};
stdenv.mkDerivation {
  name = "hello-HEAD";
  src = ./.;
  installPhase = ''
    echo "Hello" > $out
  '';
}
```

Then we can build this package with `nix-build hello-shell.nix` and a `result` symlink will show up in your current working directory. Then you can view what it says with `cat`:

```
$ cat ./result
Hello
```

This is all it takes to make a Nix package. You need to name the package, give it input source code somehow, and potentially give it build instructions. Everything else we'll cover today will build on top of this.

Let's look back at the Go [example package](#) I walked us through in [the last post](#):

```
# ...
packages = forAllSystems (system:
  let pkgs = nixpkgsFor.${system};
  in {
    go-hello = pkgs.buildGoModule {
      pname = "go-hello";
      inherit version;
      # In 'nix develop', we don't need a copy of the source tree
      # in the Nix store.
      src = ./.;

      # This hash locks the dependencies of this package. It is
      # necessary because of how Go requires network access to resolve
      # VCS. See https://www.tweag.io/blog/2021-03-04-gomod2nix/ for
```



```
# details. Normally one can build with a fake sha256 and rely on native Go
# mechanisms to tell you what the hash should be or determine what
# it should be "out-of-band" with other tooling (eg. gomod2nix).
# To begin with it is recommended to set this, but one must
# remember to bump this hash when your dependencies change.
vendorSha256 =
  "sha256-pQpattmS9Vm03ZIQUFn66az8GSmB4IVYhTTCFn6SUmo=";
};

});
#
# ...
```

This uses a different builder, one called `\pkgs.buildGoModule`. This is like the ``stdenv.mkDerivation`` builder, except it is explicitly made to handle Go projects. There are some other flags that you can set in `buildGoModule` that can be useful. You can see examples in the NixOS manual page [here](#).

Another useful builder is [Naersk](#). Naersk will automatically derive build instructions for Rust projects using the `Cargo.toml` and `Cargo.lock` files. This means that your build step can look as small as this:

naersk-lib.buildPackage [./](#)



<Mara> You can think of these builders as templates for doing larger builds. This is kinda like [the ONBUILD Dockerfile instruction(<https://docs.docker.com/engine/reference/builder/#onbuild>)], but it isn't limited to Docker. The main difference is that Nix builds are more like functions (inputs and outputs) and Docker builds focus on the individual commands you run to get the result you want. Both eventually compile down to shell commands anyways!



A More Useful Package

This "hello world" program isn't very useful on its own, however we can use it as the basis for making something a bit more useful. I have made a template for a "Hello world" HTTP server [here](#). Let's make a new folder for it and then initialize it:



<**Mara**> If you want to make your own templates, see how to do that [here](#).

```
mkdir -p ~/tmp/gohello-http
cd ~/tmp/gohello-http
git init
nix flake init -t github:Xe/templates#go-web-server
```



<**Mara**> You may see a message from [direnv](#) about needing to approve its content. This will use Nix flake's cached interpreter to give you all the advantages of something like [Lorri](#) without having to install and run a daemon.

Then make an initial commit and run it:

```
git add .
git commit -sm "initial commit"
nix build
./result/bin/web-server
```



<**Mara**> Why are you using `git add .` everywhere? Shouldn't the files be picked up implicitly?



<**Cadey**> Not always. Nix flakes only deals with files that are tracked by git when you use it in a git repository. This means that if you want the changes to be observed by Nix, you need to add them to git somehow. `git add` is good enough for this.



Or you can run it directly with `nix run`:

```
nix run
```

Docker Images

Most of the time you will build software with Nix, however that doesn't stop you from building things like Docker images with Nix. Remember that you can have the output of any shell commands be run in a Nix build (the only catch is that they can't access the internet directly), so you can build a Docker image out of that web server template by defining another package:

```
# flake.nix

packages = {
  default = ...;
  docker = let
    web = self.packages.${system}.default;
  in pkgs.dockerTools.buildLayeredImage {
    name = web.pname;
    tag = web.version;
    contents = [ web ];

    config = {
      Cmd = [ "/bin/web-server" ];
      WorkingDir = "/";
    };
  };
};
```

This will build a Docker image with the web-server binary in it. To build it, run these commands:

```
git add .
nix build .#docker
```



<Mara> What's with that last argument to `nix build`, won't that be read as a shell comment?



<Cadey> It's a reference to the package in the flake. Shell only parses comments when the `#` is the first character after whitespace, so this is



more of a URL fragment than a comment. It's telling `nix build` to build the flake package named `docker`.

It will put the resulting docker image in `./result`. To load it into docker use the following command:

```
$ docker load < result  
Loaded image: web-server:20220227
```



<Mara> Your image tag may differ depending on when you build this image. This is deterministic because that date is derived from the date that the current git commit was made.

Then you can run it with `docker run`:

```
docker run -itp 3031:3031 web-server:20220227
```

Then poke it with curl:

```
$ curl http://[::]:3031  
hello from nix!
```

You can push this image to the Docker hub like any other image. Another cool thing about this is that when you update the program, it'll only actually load the images that changed. Let's edit the hello world message:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {  
    fmt.Fprintln(w, "hello from nix building a docker image!")  
})
```



And then re-build and load it into Docker:

```
git add .
```

```
nix build .#docker  
docker load < result
```



<Mara> Woah, when I did that it only updated 2 layers. The first time that I loaded it there were something like 7 layers. What's up with that?

When you use `buildLayeredImage`, each Nix package that contributes to the image gets put in its own Docker layer. This means that only the things that have changed actually need to be considered, so when you push an updated image to another machine, the only things that will actually be pushed are the application binary and the symlink farm pointing to the `contents` of the Docker image. It uses Docker more efficiently than Docker ever could!

systemd Portable Services

EDIT(20220227 17:24): It seems that these are even more experimental than I thought. systemd Portable Services don't seem to work properly with the `StateDirectory` and `CacheDirectory` directives unless you are running a git HEAD version of systemd. Maybe you should wait a few years before trying to use them for anything serious. By then most of the kinks should be worked out.

systemd Portable Services function like Docker, but they work at the systemd level and allow you to integrate into systemd instead of running on the side of it. This gives you access to systemd's readiness signaling, logging pipeline and dependency graph so that you can integrate like a native service. They are like containers, but without a lot of the headaches around networking, stateful storage and logging. They are just systemd services at their core.



<Mara> These are kinda like Ubuntu's Snaps or Flatpaks, but they operate purely at the system level and are focused at providing things for system services instead of user-facing applications. Ubuntu's Snaps do let you create system services, but they are basically exclusively used on Ubuntu. systemd Portable Services let you target more than just Ubuntu. In the next few years with more releases of systemd, Portable Services should be



easier to use and will be more integrated with the system than Docker is.

There is currently an [open pull request](#) for adding Portable Service building support to nixpkgs, however we can mess around with it today thanks to [my portable-svc overlay](#) that copies in the contents of that pull request.



<Mara> In Nix, an overlay is a set of additional packages or functions that is put on top of nixpkgs. This overlay defines the `portableService` function that is needed to build portable services.

To make this into a portable service, first we need to add my overlay to the flake inputs:

```
# flake.nix
inputs = {
    nixpkgs.url = "nixpkgs/nixos-unstable";
    utils.url = "github:numtide/flake-utils";
    portable-svc.url = "git+https://tulpa.dev/cadey/portable-svc.git?ref=main";
};
```

Then add it as an argument to the `outputs` function:

```
outputs = { self, nixpkgs, utils, portable-svc }:
```

And then change how we are importing the `pkgs` variable. The `pkgs` variable we're currently using is imported like this:

```
let pkgs = nixpkgs.legacyPackages.${system};
```

This works, however there isn't a way to specify an overlay into this. We need to change this into a manual import of nixpkgs with the overlay specified, like this:

```
let pkgs = import nixpkgs {
    overlays = [ portable-svc.overlay ];
    inherit system;
};
```



This will let us use the `portableService` function in Nix package definitions.

Next we need to make a systemd service unit for the web server. The exact path to the program binary can and will change with every build, so it would be good to have this templated. Make a folder called `systemd`:

```
mkdir systemd
```

And put the following contents in `systemd/web-server.service.in`:

```
[Unit]
Description=A web service

[Service]
DynamicUser=yes
ExecStart=@web@/bin/web-server

[Install]
WantedBy=multi-user.target
```

Then under the docker package definition, add the package that will template out the systemd unit:

```
web-service = pkgs.substituteAll {
  name = "web-server.service";
  src = ./systemd/web-server.service.in;
  web = self.packages.${system}.default;
};
```

You can build it with `nix build .#web-service`, the output will look something like this:



```
[Unit]
Description=A web service

[Service]
DynamicUser=yes
ExecStart=/nix/store/y1863jm907wfr7gq9j0c4bd3d4bdc4vp-web-server-20220227/bin/web-server
```

[Install]

WantedBy=multi-user.target



<Mara> The `@web@` in the template was replaced with the nix store path for the web server!

Then you can add the bit that builds the portable service:

```
portable = let
  web = self.packages.${system}.default;
in pkgs.portableService {
  inherit (web) version;
  name = web.pname;
  description = "A web server";
  units = [ self.packages.${system}.web-service ];
};
```

Then you can build it with `nix build`:

```
nix build .#portable
```

And then take a look at `./result`:

```
$ file $(readlink ./result)
/nix/store/1da6b90i75n03kqlzzfdwxii0j0bzxaf-web-server_20220227.raw:
Squashfs filesystem,
little endian,
version 4.0,
xz compressed,
9555806 bytes,
2010 inodes,
blocksize: 1048576 bytes,
created: Tue Jan 1 00:00:00 1980
```



At the time of writing this article, the most reliable way to test portable services is to use Arch Linux. So you could use something like [waifud](#) to spin up an Arch Linux VM:

```
$ waifuctl create -d arch -h logos -s 20
created instance jangmo-o on logos
jangmo-o: running
jangmo-o: init: IP address: 10.77.129.208
```

Then copy it over with `scp`:

```
$ scp (readlink ./result) xe@10.77.129.208:web-server_20220227.raw
```

Then you can use `portablectl` to attach it to the system:

```
$ sudo portablectl attach ./web-server_20220227.raw
[...]
Created symlink /etc/portables/web-server_20220227.raw → /home/xe/web-server_20220227.raw.
```

And then start it like any systemd service:

```
$ sudo systemctl start web-server
```

If you want the service to start automatically, add `--enable --now` to the `portablectl attach` command. That will enable the service in systemd and then start it, like when you run `systemctl enable --now something.service`.



And then inspect the service's status with `systemctl`:

```
$ sudo systemctl status web-server
● web-server.service - A web service
   Loaded: loaded (/etc/systemd/system.attached/web-server.service; disabled; vendor preset:
           Loaded: loaded (/etc/systemd/system.attached/web-server.service; disabled; vendor preset:
```

```
Drop-In: /etc/systemd/system.attached/web-server.service.d
          └─10-profile.conf, 20-portable.conf
Active: active (running) since Sun 2022-02-27 18:21:01 UTC; 20s ago
Main PID: 960 (web-server)
Tasks: 5 (limit: 513)
Memory: 8.1M
CPU: 189ms
CGroup: /system.slice/web-server.service
         └─960 /nix/store/y1863jm907wfr7gq9j0c4bd3d4bdc4vp-web-server-20220227/bin/web-se
```

Feb 27 18:21:01 jangmo-o systemd[1]: Started A web service.

Feb 27 18:21:01 jangmo-o web-server[960]: 2022/02/27 18:21:01 listening for HTTP on :3031

And finally poke it with curl:

```
$ curl http://[::]:3031
hello from nix building a docker image!
```

And then you can change the handler to something like:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "PORTABLE=%s\n", os.Getenv("PORTABLE"))
})
```

Rebuild the image with `nix build`:

```
$ git add .
$ nix build .#portable
```

Copy it to the arch VM with `scp`:

```
$ scp $(readlink ./result) xe@10.77.129.208:web-server_20220227.raw
```

And finally run `portablectl reattach` to upgrade it:



```
$ sudo portectl reattach --now ./web-server_20220227.raw
Queued /org/freedesktop/systemd1/job/858 to call RestartUnit on portable service
web-server.service.
```

Then you can see that it restarted the unit with `systemctl status`:

```
$ sudo systemctl status web-server
● web-server.service - A web service
  Loaded: loaded (/etc/systemd/system.attached/web-server.service; disabled; vendor preset:
  Drop-In: /etc/systemd/system.attached/web-server.service.d
            └─10-profile.conf, 20-portable.conf
  Active: active (running) since Sun 2022-02-27 18:30:04 UTC; 37s ago
    Main PID: 1074 (web-server)
      Tasks: 6 (limit: 513)
     Memory: 8.1M
       CPU: 182ms
      CGroup: /system.slice/web-server.service
              └─1074 /nix/store/j1mfz3ydn13qmvcgrql33zi0dw3x7dk-web-server-20220227/bin/web-s
```

Feb 27 18:30:04 jangmo-o systemd[1]: Started A web service.

Feb 27 18:30:04 jangmo-o web-server[1074]: 2022/02/27 18:30:04 listening for HTTP on :3031

And finally poke it with curl:

```
$ curl http://[::]:3031
PORTABLE=web-server_20220227.raw
```

And there you go! Nix created a portable system service, we spawned it on a newly created Arch Linux VM and then were able to update it so that we could replace the message.



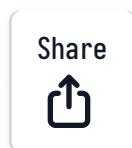
Nix builds can do more than just turn code into software. They can create Docker images, Portable Services, virtual machine images and more. The only real limit is what you can

imagine.

Flakes make it easier to pull in and munge about packages. Before flakes you'd need to have a few `*.nix` files like `docker.nix` for the docker image and `portable.nix` for the portable service. You'd also have to pull in something like [Niv](#) to make sure everything uses the same version of nixpkgs, and even then it's opt-in, not opt-out, so it's easy to mess things up and not use the pinned versions of things. Flakes make that explicit behavior implicit, so you can't bring in dependencies you aren't aware of.

If you want to see the code repo I developed while writing this post, see [cadey/gohello-http](#) on my git server.

Thanks for reading!



Facts and circumstances may have changed since publication. Please contact me before jumping to conclusions if something seems wrong or unclear.

Tags: nix, nixos, docker, systemd

Copyright 2012-2024 Xe Iaso (Christine Dodrill). Any and all opinions listed here are my own and not representative of any of my employers, past, future, and/or present.

Like what you see? Donate on [Patreon](#) like [these awesome people!](#)

Served by xesite v4 (/nix/store/9rxy3y9y9ffwx6xdvb56pgciar4b5138-xesite_v4-20240128 /bin/xesite) with site version [a1841325](#), source code available [here](#).





NIX FLAKES, PART 1: AN INTRODUCTION AND TUTORIAL

25 May 2020 — by Eelco Dolstra



This is the first in a series of blog posts intended to provide a gentle introduction to *flakes*, a new Nix feature that improves reproducibility, composability and usability in the Nix ecosystem. This blog post describes why flakes were introduced, and give a short tutorial on how to use them.

Flakes were developed at Tweag and funded by Target Corporation and Tweag.

WHAT PROBLEMS DO FLAKES SOLVE?

Once upon a time, Nix pioneered reproducible builds: it tries hard to ensure



This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)

/nixpkgs/config.nix), environment variables, Git repositories, files in the Nix search path (\$NIX_PATH), command-line arguments (--arg) and the system type (builtins.currentSystem). In other words, *evaluation isn't as hermetic as it could be*. In practice, ensuring reproducible evaluation of things like NixOS system configurations requires special care.

Furthermore, there is no *standard way to compose Nix-based projects*. It's rare that everything you need is in Nixpkgs; consider for instance projects that use Nix as a build tool, or NixOS system configurations. Typical ways to compose Nix files are to rely on the Nix search path (e.g. import <nixpkgs>) or to use fetchGit or fetchTarball . The former has poor reproducibility, while the latter provides a bad user experience because of the need to manually update Git hashes to update dependencies.

There is also no easy way to *deliver* Nix-based projects to users. Nix has a “channel” mechanism (essentially a tarball containing Nix files), but it's not easy to create channels and they are not composable. Finally, Nix-based projects lack a standardized structure. There are some conventions (e.g. shell.nix or release.nix) but they don't cover many common use cases; for instance, there is no way to discover the NixOS modules provided by a repository.

Flakes are a solution to these problems. A flake is simply a source tree (such as a Git repository) containing a file named flake.nix that provides a standardized interface to Nix artifacts such as packages or NixOS modules. Flakes can have dependencies on other flakes, with a “lock file” pinning those dependencies to exact revisions to ensure reproducible evaluation.

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



want to play with flakes, you can get this version of Nix from [Nixpkgs](#):

```
$ nix-shell -I nixpkgs=channel:nixos-21.05 --packages nixUnstable
```

Since flakes are an experimental feature, you also need to add the following line to `~/.config/nix/nix.conf`:

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

or pass the flag `--experimental-features 'nix-command flakes'` whenever you call the `nix` command.

USING FLAKES

To see flakes in action, let's start with a simple Unix package named `dwarffs` (a FUSE filesystem that automatically fetches debug symbols from the Internet). It lives in a GitHub repository at [https://github.com/edolstra/dwarf`fs`](https://github.com/edolstra/dwarf<code>fs</code>); it is a flake because it contains a file named `flake.nix`. We will look at the contents of this file later, but in short, it tells Nix what the flake provides (such as Nix packages, NixOS modules or CI tests).

The following command fetches the `dwarffs` Git repository, builds its *default package* and runs it.

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



to build a specific revision:

```
$ nix shell github:edolstra/dwarfss/cd7955af31698c571c30b7a0f78e5
```

Nix tries very hard to ensure that the result of building a flake from such a URL is always the same. This requires it to restrict a number of things that Nix projects could previously do. For example, the `dwarfss` project requires a number of dependencies (such as a C++ compiler) that it gets from the Nix Packages collection (Nixpkgs). In the past, you might use the `NIX_PATH` environment variable to allow your project to find Nixpkgs. In the world of flakes, this is no longer allowed: flakes have to declare their dependencies explicitly, and these dependencies have to be *locked* to specific revisions.

In order to do so, `dwarfss`'s `flake.nix` file declares an explicit dependency on Nixpkgs, which is also a `flake`. We can see the dependencies of a flake as follows:

```
$ nix flake metadata github:edolstra/dwarfss
Description: A filesystem that fetches DWARF debug info from the
...
github:edolstra/dwarfss/d11b181af08bfda367ea5cf7fad103652dc0409f
|   └──nix: github:NixOS/nix/3aaceeb7e2d3fb8a07a1aa5a21df1dca6bbaa0e
|       └──nixpkgs: github:NixOS/nixpkgs/b88ff468e9850410070d4e0ccd6
└──nixpkgs: github:NixOS/nixpkgs/b88ff468e9850410070d4e0ccd68c70
```

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



FLAKE OUTPUTS

Another goal of flakes is to provide a standard structure for discoverability within Nix-based projects. Flakes can provide arbitrary Nix values, such as packages, NixOS modules or library functions. These are called its *outputs*. We can see the outputs of a flake as follows:

```
$ nix flake show github:edolstra/dwarfss
github:edolstra/dwarfss/d11b181af08bfda367ea5cf7fad103652dc0409f
├─checks
│  ├─aarch64-linux
│  │  └─build: derivation 'dwarfss-0.1.20200409'
│  ├─i686-linux
│  │  └─build: derivation 'dwarfss-0.1.20200409'
│  └─x86_64-linux
    └─build: derivation 'dwarfss-0.1.20200409'
└─defaultPackage
  ├─aarch64-linux: package 'dwarfss-0.1.20200409'
  ├─i686-linux: package 'dwarfss-0.1.20200409'
  └─x86_64-linux: package 'dwarfss-0.1.20200409'
└─nixosModules
  └─dwarfss: NixOS module
└─overlay: Nixpkgs overlay
```

While a flake can have arbitrary outputs, some of them, if they exist, have a special meaning to certain Nix commands and therefore must have a specific type. For example, the output `defaultPackage.<system>`

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



By the way, the standard `checks` output specifies a set of derivations to be built by a continuous integration system such as Hydra. Because flake evaluation is hermetic and the lock file locks all dependencies, it's guaranteed that the `nix build` command above will evaluate to the same result as the one in the CI system.

THE FLAKE REGISTRY

Flake locations are specified using a URL-like syntax such as `github:edolstra/dwarfes` or `git+https://github.com/NixOS/patchelf`. But because such URLs would be rather verbose if you had to type them all the time on the command line, there also is a [flake registry](#) that maps symbolic identifiers such as `nixpkgs` to actual locations like `https://github.com/NixOS/nixpkgs`. So the following are (by default) equivalent:

```
$ nix shell nixpkgs#cowsay --command cowsay Hi!  
$ nix shell github:NixOS/nixpkgs#cowsay --command cowsay Hi!
```

It's possible to override the registry locally. For example, you can override the `nixpkgs` flake to your own Nixpkgs tree:

```
$ nix registry add nixpkgs ~/my-nixpkgs
```

or pin it to a specific revision:

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)

Unlike Nix channels, creating a flake is pretty simple: you just add a `flake.nix` and possibly a `flake.lock` to your project's repository. As an example, suppose we want to create our very own Hello World and distribute it as a flake. Let's create this project first:

```
$ git init hello
$ cd hello
$ echo 'int main() { printf("Hello World"); }' > hello.c
$ git add hello.c
```

To turn this Git repository into a flake, we add a file named `flake.nix` at the root of the repository with the following contents:

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)

```
{  
    description = "A flake for building Hello World";  
  
    inputs.nixpkgs.url = github:NixOS/nixpkgs/nixos-20.03;  
  
    outputs = { self, nixpkgs }: {  
  
        defaultPackage.x86_64-linux =  
            # Notice the reference to nixpkgs here.  
            with import nixpkgs { system = "x86_64-linux"; };  
            stdenv.mkDerivation {  
                name = "hello";  
                src = self;  
                buildPhase = "gcc -o hello ./hello.c";  
                installPhase = "mkdir -p $out/bin; install -t $out/bin he  
            };  
    };  
}
```

The command `nix flake init` creates a basic `flake.nix` for you.

Note that any file that is not tracked by Git is invisible during Nix evaluation, in order to ensure hermetic evaluation. Thus, you need to make `flake.nix` visible to Git:

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



```
$ nix build  
warning: creating lock file '/home/eelco/Dev/hello/flake.lock'  
warning: Git tree '/home/eelco/Dev/hello' is dirty  
  
$ ./result/bin/hello  
Hello World
```

or equivalently:

```
$ nix shell --command hello  
Hello World
```

It's also possible to get an interactive development environment in which all the dependencies (like GCC) and shell variables and functions from the derivation are in scope:

```
$ nix develop  
$ eval "$buildPhase"  
$ ./hello  
Hello World
```

So what does all that stuff in `flake.nix` mean?

The `description` attribute is a one-line description shown by `nix`



This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)

produces an attribute set. The function arguments are the flakes specified in `inputs`.

The `self` argument denotes *this* flake. Its primarily useful for referring to the source of the flake (as in `src = self;`) or to other outputs (e.g. `self.defaultPackage.x86_64-linux`).

The attributes produced by `outputs` are arbitrary values, except that (as we saw above) there are some standard outputs such as `defaultPackage.${system}`.

Every flake has some metadata, such as `self.lastModifiedDate`, which is used to generate a version string like `hello-20191015`.

You may have noticed that the dependency specification

`github:NixOS/nixpkgs/nixos-20.03` is imprecise: it says that we want to use the `nixos-20.03` branch of Nixpkgs, but doesn't say which Git revision. This seems bad for reproducibility. However, when we ran `nix build`, Nix automatically generated a lock file that precisely states which revision of `nixpkgs` to use:

```
$ cat flake.lock
{
  "nodes": {
    "nixpkgs": {
      "info": {
        "lastModified": 1587398327,
        "narHash": "sha256-mEKkeLgUrzAsdEaJ/1wdvYn0YZBAKEG3AN21ko
      },
      "locked": {
        "nixpkgs": {
          "info": {
            "lastModified": 1587398327,
            "narHash": "sha256-mEKkeLgUrzAsdEaJ/1wdvYn0YZBAKEG3AN21ko
          }
        }
      }
    }
  }
}
```

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



```
"owner": "NixOS",
"ref": "nixos-20.03",
"repo": "nixpkgs",
"type": "github"
},
},
"root": {
  "inputs": {
    "nixpkgs": "nixpkgs"
  }
},
"root": "root",
"version": 5
}
```

Any subsequent build of this flake will use the version of `nixpkgs` recorded in the lock file. If you add new inputs to `flake.nix`, when you run any command such as `nix build`, Nix will automatically add corresponding locks to `flake.lock`. However, it won't replace existing locks. If you want to update a locked input to the latest version, you need to ask for it:

```
$ nix flake lock --update-input nixpkgs
$ nix build
```

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



```
$ git remote add origin git@github.com:edolstra/hello.git  
$ git push -u origin master
```

Other users can then use this flake:

```
$ nix shell github:edolstra/hello -c hello
```

NEXT STEPS

In the next blog post, we'll talk about typical uses of flakes, such as managing NixOS system configurations, distributing Nixpkgs overlays and

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)

COMPANY

[About](#)
[Open Source](#)
[Careers](#)
[Contact Us](#)

WHAT WE DO

[Strategy](#)
[Product Development](#)
[Platform Modernization](#)
[Digital Operations](#)
[Teamwork with Atlassian](#)
[Work](#)

INSIGHTS

[Modus Blog](#)
[Tweag Blog](#)
[Teamwork Blog](#)
[Research](#)
[Innovation podcast](#)

CONNECT WITH US



This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)



NIX FLAKES, PART 2: EVALUATION CACHING

25 June 2020 — by Eelco Dolstra



Nix evaluation is often quite slow. In this blog post, we'll have a look at a nice advantage of the hermetic evaluation model enforced by flakes: the ability to cache evaluation results reliably. For a short introduction to flakes, see our [previous blog post](#).

WHY NIX EVALUATION IS SLOW

Nix uses a simple, interpreted, purely functional language to describe package dependency graphs and NixOS system configurations. So to get any information about those things, Nix first needs to *evaluate* a substantial



This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)

```
$ command time nix-env -qa | wc -l  
5.09user 0.49system 0:05.59elapsed 99%CPU (0avgtext+0avgdata 1522  
28012
```

Evaluating individual packages or configurations can also be slow. For example, using `nix-shell` to enter a development environment for [Hydra](#), we have to wait a bit, even if all dependencies are present in the Nix store:

```
$ command time nix-shell --command 'exit 0'  
1.34user 0.18system 0:01.69elapsed 89%CPU (0avgtext+0avgdata 4348
```

That might be okay for occasional use but a wait of one or more seconds may well be unacceptably slow in scripts.

Note that the evaluation overhead is completely independent from the time it takes to actually build or download a package or configuration. If something is already present in the Nix store, Nix won't build or download it again. But it still needs to re-evaluate the Nix files to determine *which* Nix store paths are needed.

CACHING EVALUATION RESULTS

So can't we speed things up by *caching* evaluation results? After all, the Nix language is purely functional, so it seems that re-evaluation should produce the same result, every time. Naively, maybe we can keep a cache

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)

hermetic. For example, a `.nix` file can import other Nix files through relative or absolute paths (such as `~/ .config/nixpkgs/config.nix` for `Nixpkgs`) or by looking them up in the Nix search path (`$NIX_PATH`). So unless we perfectly keep track of *all* the files used during evaluation, a cached result might be inconsistent with the current input.

(As an aside: for a while, Nix has had an experimental replacement for `nix-env -qa` called `nix search`, which used an ad hoc cache for package metadata. It had exactly this cache invalidation problem: it wasn't smart enough to figure out whether its cache was up to date with whatever revision of `Nixpkgs` you were using. So it had a manual flag `--update-cache` to allow the user to force cache invalidation.)

FLAKES TO THE RESCUE

Flakes solve this problem by ensuring fully hermetic evaluation. When you evaluate an output attribute of a particular flake (e.g. the attribute `defaultPackage.x86_64-linux` of the `dwarffs` flake), Nix disallows access to any files outside that flake or its dependencies. It also disallows impure or platform-dependent features such as access to environment variables or the current system type.

This allows the `nix` command to aggressively cache evaluation results without fear of cache invalidation problems. Let's see this in action by running Firefox from the `nixpkgs` flake. If we do this with an empty evaluation cache, Nix needs to evaluate the entire dependency graph of Firefox, which takes a quarter of a second:

```
$ command time nix shell nixpkgs#firefox -c firefox --version
```



This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)

```
$ command time nix shell nixpkgs#firefox -c firefox --version
Mozilla Firefox 75.0
0.01user 0.01system 0:00.03elapsed 93%CPU (0avgtext+0avgdata 2584
```

The cache is implemented using a simple SQLite database that stores the values of flake output attributes. After the first command above, the cache looks like this:

```
$ sqlite3 ~/.cache/nix/eval-cache-v1/302043eedfbce13ecd8169612849
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE Attributes (
    parent      integer not null,
    name        text,
    type        integer not null,
    value       text,
    primary key (parent, name)
);
INSERT INTO Attributes VALUES(0, '', 0, NULL);
INSERT INTO Attributes VALUES(1, 'packages', 3, NULL);
INSERT INTO Attributes VALUES(1, 'legacyPackages', 0, NULL);
INSERT INTO Attributes VALUES(3, 'x86_64-linux', 0, NULL);
INSERT INTO Attributes VALUES(4, 'firefox', 0, NULL);
INSERT INTO Attributes VALUES(5, 'type', 2, 'derivation');
INSERT INTO Attributes VALUES(5, 'drvPath', 2, '/nix/store/7mz8pkgpl
INSERT INTO Attributes VALUES(5, 'outPath', 2, '/nix/store/5x1i2gp8' 
```

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)

{type,drvPath,outPath,outputName} . It also stores negative lookups, that is, attributes that don't exist (such as packages).

The name of the SQLite database, 302043eedf...sqlite in this example, is derived from the contents of the top-level flake. Since the flake's lock file contains content hashes of all dependencies, this is enough to efficiently and completely capture all files that might influence the evaluation result. (In the future, we'll optimise this a bit more: for example, if the flake is a Git repository, we can simply use the Git revision as the cache name.)

The nix search command has been updated to use the new evaluation cache instead of its previous ad hoc cache. For example, searching for Blender is slow the first time:

```
$ command time nix search nixpkgs blender
* legacyPackages.x86_64-linux.blender (2.82a)
  3D Creation/Animation/Publishing System
  5.55user 0.63system 0:06.17elapsed 100%CPU (0avgtext+0avgdata 149
```

but the second time it is pretty fast and uses much less memory:

```
$ command time nix search nixpkgs blender
* legacyPackages.x86_64-linux.blender (2.82a)
  3D Creation/Animation/Publishing System
  0.41user 0.00system 0:00.42elapsed 99%CPU (0avgtext+0avgdata 2110
```

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)

There is only one way in which cached results can become “stale”, in a way. Nix evaluation produces store derivations such as `/nix/store/7mz8pkgpl24wyab8nnny0zclvca7ki2m8-firefox-75.0.drv` as a side effect. (`.drv` files are essentially a serialization of the dependency graph of a package.) These store derivations may be garbage-collected. In that case, the evaluation cache points to a path that no longer exists. Thus, Nix checks whether the `.drv` file still exist, and if not, falls back to evaluating normally.

FUTURE IMPROVEMENTS

Currently, the evaluation cache is only created and used locally. However, Nix could automatically *download* precomputed caches, similar to how it has a binary cache for the contents of store paths. That is, if we need a cache like `302043eedf....sqlite`, we could first check if it's available on `cache.nixos.org` and if so fetch it from there. In this way, when we run a command such as `nix shell nixpkgs#firefox`, we could even avoid the need to fetch the actual source of the flake!

Another future improvement is to populate and use the cache in the evaluator itself. Currently the cache is populated and cached in the user interface (that is, the `nix` command). The command `nix shell nixpkgs#firefox` will create a cache entry for `firefox`, but not for the dependencies of `firefox`; thus a subsequent `nix shell nixpkgs#thunderbird` won't see a speed improvement even though it shares most of its dependencies. So it would be nice if the evaluator had knowledge of the evaluation cache. For example, the evaluation of thunks that represent attributes like `nixpkgs.legacyPackages.x86_64-linux.<package name>` could check and update the cache.

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

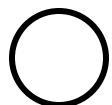
[Do Not Sell My Personal Information](#)

[Accept](#)



Part 3: Managing NixOS systems

ABOUT THE AUTHORS



Eelco
Dolstra

If you enjoyed this article, you might be interested in [joining the Tweag team](#).

This article is licensed under a [Creative Commons Attribution 4.0 International license](#).

← [Linear types are merged in GHC](#)
[Splittable pseudo-random number generators in Haskell: random v1.1 and v1.2](#) →

COMPANY

[About](#)
[Open Source](#)
[Careers](#)
[Contact Us](#)

WHAT WE DO

[Strategy](#)
[Product Development](#)
[Platform Modernization](#)
[Digital Operations](#)
[Teamwork with Atlassian](#)
[Work](#)

INSIGHTS

[Modus Blog](#)
[Tweag Blog](#)

CONNECT WITH US



[Do Not Sell My Personal Information](#)

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Accept](#)



© 2023 Modus Create, LLC [Privacy Policy](#) [Sitemap](#)

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)



NIX FLAKES, PART 3: MANAGING NIXOS SYSTEMS

31 July 2020 — by Eelco Dolstra



nix



devops

This is the third in a series of blog posts about *Nix flakes*. The [first part](#) motivated why we developed flakes — to improve Nix's reproducibility, composable and usability — and gave a short tutorial on how to use flakes. The [second part](#) showed how flakes enable reliable caching of Nix evaluation results. In this post, we show how flakes can be used to manage NixOS systems in a reproducible and composable way.

WHAT PROBLEMS ARE WE TRYING TO SOLVE?

Lack of reproducibility

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)

previously validated in a test environment.

However, the default NixOS workflow doesn't provide reproducible system configurations out of the box. Consider a typical sequence of commands to upgrade a NixOS system:

You edit `/etc/nixos/configuration.nix`.

You run `nix-channel --update` to get the latest version of the `nixpkgs` repository (which contains the NixOS sources).

You run `nixos-rebuild switch`, which evaluates and builds a function in the `nixpkgs` repository that takes `/etc/nixos/configuration.nix` as an input.

In this workflow, `/etc/nixos/configuration.nix` might not be under configuration management (e.g. point to a Git repository), or if it is, it might be a dirty working tree. Furthermore, `configuration.nix` doesn't specify what Git revision of `nixpkgs` to use; so if somebody else deploys the same `configuration.nix`, they might get a very different result.

Lack of traceability

The ability to reproduce a configuration is not very useful if you can't tell what configuration you're actually running. That is, from a running system, you should be able to get back to its specification. So there is a lack of *traceability*: the ability to trace derived artifacts back to their sources. This is an essential property of good configuration management, since without it, we don't know *what* we're actually running in production, so reproducing or fixing problems becomes much harder.



This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)

```
$ nixos-version --json | jq -r .nixpkgsRevision  
a84b797b28eb104db758b5cb2b61ba8face6744b
```

Unfortunately, this doesn't allow you to recover `configuration.nix` or any other external NixOS modules that were used by the configuration.

Lack of composability

It's easy to enable a package or system service in a NixOS configuration if it is part of the `nixpkgs` repository: you just add a line like `environment.systemPackages = [pkgs.hello];` or `services.postgresql.enable = true;` to your `configuration.nix`. But what if we want to use a package or service that *isn't* part of `Nixpkgs`? Then we're forced to use mechanisms like `$NIX_PATH`, `builtins.fetchGit`, imports using relative paths, and so on. These are not standardized (since everybody uses different conventions) and are inconvenient to use (for example, when using `$NIX_PATH`, it's the user's responsibility to put external repositories in the right directories).

Put another way: NixOS is currently built around a *monorepo* workflow — the entire universe should be added to the `nixpkgs` repository, because anything that isn't, is much harder to use.

It's worth noting that any NixOS system configuration already violates the monorepo assumption: your system's `configuration.nix` is not part of the `nixpkgs` repository.

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



This solves the problems described above:

Reproducibility: the entire system configuration (including everything it depends on) is captured by the flake and its lock file. So if two people check out the same Git revision of a flake and build it, they should get the same result.

Traceability: `nixos-version` prints the Git revision of the top-level configuration flake, not its `nixpkgs` input.

Composability: it's easy to pull in packages and modules from other repositories as flake inputs.

PREREQUISITES

Flake support has been added as an experimental feature to NixOS 20.03. However, flake support is not part of the current stable release of Nix (2.3). So to get a NixOS system that supports flakes, you first need to switch to the `nixUnstable` package and enable some experimental features. This can be done by adding the following to `configuration.nix`:

```
nix.package = pkgs.nixUnstable;
nix.extraOptions = ''
  experimental-features = nix-command flakes
';
'
```

CREATING A NIXOS CONFIGURATION FLAKE

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)

```
$ git commit -a -m 'Initial version'
```

Note that the `-t` flag to `nix flake init` specifies a *template* from which to copy the initial contents of the flake. This is useful for getting started. To see what templates are available, you can run:

```
$ nix flake show templates
```

For reference, this is what the initial `flake.nix` looks like:

```
{  
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-20.03";  
  
  outputs = { self, nixpkgs }: {  
  
    nixosConfigurations.container = nixpkgs.lib.nixosSystem {  
      system = "x86_64-linux";  
      modules =  
        [ ( { pkgs, ... }: {  
          boot.isContainer = true;  
  
          # Let 'nixos-version --json' know about the Git revision  
          # of this flake.  
          system.configurationRevision = nixpkgs.lib.mkIf (self  
    
```

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



```
        enable = true;
        adminAddr = "morty@example.org";
    );
}
];
};

}
```

That is, the flake has one input, namely `nixpkgs` - specifically the 20.03 branch. It has one output, `nixosConfigurations.container`, which evaluates a NixOS configuration for tools like `nixos-rebuild` and `nixos-container`. The main argument is `modules`, which is a list of NixOS configuration modules. This takes the place of the file `configuration.nix` in non-flake deployments. (In fact, you can write `modules = [./configuration.nix]` if you're converting a pre-flake NixOS configuration.)

Let's create and start the container! (Note that `nixos-container` currently requires you to be `root`.)

```
# nixos-container create flake-test --flake /path/to/my-flake
host IP is 10.233.4.1, container IP is 10.233.4.2

# nixos-container start flake-test
```

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



As an aside, if you just want to *build* the container without the `nixos-container` command, you can do so as follows:

```
$ nix build /path/to/my-flake#nixosConfigurations.container.config
```

Note that `system.build.toplevel` is an internal NixOS option that evaluates to the “system” derivation that commands like `nixos-rebuild`, `nixos-install` and `nixos-container` build and activate. The symlink `/run/current-system` points to the output of this derivation.

HERMETIC EVALUATION

One big difference between “regular” NixOS systems and flake-based NixOS systems is that the latter record the Git revisions from which they were built. We can query this as follows:

```
# nixos-container run flake-test -- nixos-version --json
{"configurationRevision": "9190c396f4dcfc734e554768c53a81d1c231c6a",
 "nixosVersion": "20.03.20200622.13c15f2",
 "nixpkgsRevision": "13c15f26d44cf7f54197891a6f0c78ce8149b037"}
```

Here, `configurationRevision` is the Git revision of the repository `/path/to/my-flake`. Because evaluation is hermetic, and the lock file locks all flake inputs such as `nixpkgs`, knowing the revision `9190c39...`

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)

```
--flake /path/to/my-flake?rev=9190c396f4dcfc734e554768c53a81d
```

DIRTY CONFIGURATIONS

It's not required that you commit all changes to a configuration before deploying it. For example, if you change the `adminAddr` line in `flake.nix` to

```
adminAddr = "rick@example.org";
```

and redeploy the container, you will get:

```
# nixos-container update flake-test
warning: Git tree '/path/to/my-flake' is dirty
...
reloading container...
```

and the container will no longer have a configuration Git revision:

```
# nixos-container run flake-test -- nixos-version --json | jq .co
null
```

While this may be convenient for testing, in production we really want to



This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)

Git tree '/path/to/my-flake' is dirty

Another is to require a clean Git tree in `flake.nix`, for instance by adding a check to the definition of `system.configurationRevision`:

```
system.configurationRevision =  
  if self ? rev  
  then self.rev  
  else throw "Refusing to build from a dirty Git tree!";
```

ADDING MODULES FROM THIRD-PARTY FLAKES

One of the main goals of flake-based NixOS is to make it easier to use packages and modules that are not included in the `nixpkgs` repository. As an example, we'll add [Hydra](#) (a continuous integration server) to our container.

Here's how we add it to our container. We specify it as an additional input:

```
inputs.hydra.url = "github:NixOS/hydra";
```

and as a corresponding function argument to the `outputs` function:

```
outputs = { self, nixpkgs, hydra }: {
```

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)

```
[ hydra.nixosModules.hydraTest

  ({ pkgs, ... }: {
    ... our own configuration ...

    # Hydra runs on port 3000 by default, so open it in t
    networking.firewall.allowedTCPPorts = [ 3000 ];
  })
];
```

Note that we can discover the name of this module by using `nix flake show`:

```
$ nix flake show github:NixOS/hydra
github:NixOS/hydra/d0deebc4fc95dbeb0249f7b774b03d366596fbed
├─...
├─nixosModules
|  ├─hydra: NixOS module
|  ├─hydraProxy: NixOS module
|  └─hydraTest: NixOS module
└─overlay: Nixpkgs overlay
```

After committing this change and running `nixos-container update`, we can check whether `hydra` is working in the container by visiting <http://flake-test:3000/> in a web browser.

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



```
$ nixos-container update flake-test --update-input nixpkgs --comm
```

updates the `nixpkgs` input to the latest revision on the `nixos-20.03` branch, and commits the new lock file with a commit message that records the input change.

A useful flag during development is `--override-input`, which allows you to point a flake input to another location, completely overriding the input location specified by `flake.nix`. For example, this is how you can build the container against a local Git checkout of Hydra:

```
$ nixos-container update flake-test --override-input hydra /path/
```

ADDING OVERLAYS FROM THIRD-PARTY FLAKES

Similarly, we can add Nixpkgs overlays from other flakes. (Nixpkgs overlays add or override packages in the `pkgs` set.) For example, here is how you add the overlay provided by the `nix` flake:

```
outputs = { self, nixpkgs, nix }: {
  nixosConfigurations.container = nixpkgs.lib.nixosSystem {
    ...
    modules =
    [
      ({ pkgs, ... }: {
```

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)

[Accept](#)



}

USING NIXOS-REBUILD

Above we saw how to manage NixOS containers using flakes. Managing “real” NixOS systems works much the same, except using `nixos-rebuild` instead of `nixos-container`. For example,

```
# nixos-rebuild switch --flake /path/to/my-flake#my-machine
```

builds and activates the configuration specified by the flake output `nixosConfigurations.my-machine`. If you omit the name of the configuration (`#my-machine`), `nixos-rebuild` defaults to using the current host name.

PINNING NIXPKGS

It’s often convenient to *pin* the `nixpkgs` flake to the exact version of `nixpkgs` used to build the system. This ensures that commands like `nix shell nixpkgs#<package>` work more efficiently since many or all of the dependencies of `<package>` will already be present. Here is a bit of NixOS configuration that pins `nixpkgs` in the system-wide flake registry:

```
nix.registry.nixpkgs.flake = nixpkgs;
```

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)

CONCLUSION

In this blog post we saw how Nix flakes make NixOS configurations hermetic and reproducible. In a future post, we'll show how we can do the same for cloud deployments using NixOps.

Acknowledgment: The development of flakes was partially funded by Target Corporation.

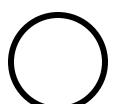
NIX FLAKES SERIES

[Part 1: An introduction and tutorial](#)

[Part 2: Evaluation caching](#)

[Part 3: Managing NixOS systems](#)

ABOUT THE AUTHORS



Eelco
Dolstra

If you enjoyed this article, you might be interested in [joining the Tweag team](#).

This article is licensed under a [Creative Commons Attribution 4.0 International license](#).

COMPANY

[About](#)

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

WHAT WE DO

[Strategy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)

INSIGHTS

Modus Blog
Tweag Blog
Teamwork Blog
Research
Innovation podcast

CONNECT WITH US



© 2023 Modus Create, LLC [Privacy Policy](#) [Sitemap](#)

This website uses cookies to enhance user experience and to analyze performance and traffic on our website. We also share information about your use of our site with our social media, advertising and analytics partners. [Privacy Policy](#)

[Do Not Sell My Personal Information](#)



[Accept](#)

jd

[blog](#) [about](#) [projects](#) [github](#) 

NixOS Configuration with Flakes

2021.09.11 :: 19 min read :: ^[NixOS Desktop](#) :: #[NixOS](#) #[Nix](#) #[X11](#) #[Linux](#)

My Flakes Philosophy

In this blog post I will overview how to setup flakes with [NixOS](#) and [home-manager](#), and my approach to system configuration. I will also take care to introduce the Nix language and provide links to learning more. This is the first post in my *NixOS Desktop* series which I will use to explore my Nix and Linux journey. I installed Linux/NixOS for the first time one month ago so it is following me in real time from beginner to eventually, hopefully master :). This post is heavy on the Nix as I am laying the foundation for future posts. But in the future expect it to be much more balanced. For a sneak peek of what is to come check out my [dotfiles](#).

NixOS and home-manager have extensive options but most of them only need to be configured once. Things like sound settings, boot, etc. can be abstracted into groups. Therefore I leverage the Nix module system, which I will go into in detail, to create high level options that makes setting up new systems and profiles easy.

Before I get into the details, I have to thank [Wil Taylor](#) for his amazing NixOS series (see below) and his [dotfiles](#) repository for providing a baseline. My configuration started off as a copy of his and has since evolved but much remains similar. His name will appear often for credit in this post. He uses a *role* system which involves importing configuration files, while I prefer to use the built-in module system which provides more flexibility.

Note: While flakes are technically unstable, I have been daily driving them with no issues.



Setting up NixOS

There has been much written on setting up NixOS so if you don't have it installed check out these resources:

- * [Graham Christensen Dell Setup](#)

- * Setting up partitions and encryption (luks)

- * [Wil Taylor's NixOS series](#)

- * Overview of how to install NixOS and basic flake introduction.

- Also overviews how to set up an initial NixOS flake

- configuration. I used the series to get started with

- NixOS/Flakes and highly recommend watching the full series.

Introduction to Flakes

I will be going into a little of how flakes work but there are already some great writeups. My go-to when I need to brush up is [Practical Nix Flakes](#). Also Wil Taylor's series provides an overview.

See the [wiki](#) for installation and more information. I recommend following the system-wide installation as the `nixFlakes` installation option will not work as we need access to the `nixos-rebuild **** --flake` command.

Writing flake.nix

Assuming you have Nix installed with flakes enabled we will start our journey to creating a configuration. The core of the system configuration is `flake.nix`. This is the file that all our nix flake commands look for.

Initialize flake

First create your directory that will host your configuration. I followed Wil Taylor's lead (see NixOS series) and created `.dotfiles` folder in my home directory. In your folder call `nix flake init` which will create a basic flake. You should see:



```
{ # .dotfiles/flake.nix
  description = "A very basic flake";
```

```
outputs = { self, nixpkgs }: {

    packages.x86_64-linux.hello = nixpkgs.legacyPackages.x86_64-linux.h
    defaultPackage.x86_64-linux = self.packages.x86_64-linux.hello;

};

}
```

We can update the description to signify that this is our system config

```
config description = "System config"; .
```

Setting inputs

Inputs are implicit in `nix flake init`. But we want to make it explicit and add extra inputs such as home-manager.

I am a believer in home-manager for configuring my user environment. It allows me to have my user config follow the same philosophies of my NixOS config. Combined with its support of flakes, all configuration can be in a single github repository and is reproducible.

I also use unstable nixpkgs (see channels) as the default which is a preference of mine. I have home-manager follow my nixpkgs so everything is following my own `flake.lock`. For more information on inputs check out the manual.

URLs can be in any format as described by the schema. By default it assumes that any repository as input is a `flake`. Therefore it looks for a `flake.nix` file in the base directory.

```
{ # .dotfiles/flake.nix
  #...

  inputs = {
    nixpkgs.url = "nixpkgs/nixos-unstable";
    home-manager = {
```



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

};

#...
}
```

All external inputs should be placed in the inputs. These inputs are what Nix uses to generate the `flake.lock` file. The lock file sets the versions being used which is what provides the reproducibility. As our home-manager and system configuration are in a single flake, updating our packages will be as simple as `nix flake update`.

Setting up outputs

The outputs of a flake are where our actual configuration will be. However, keeping our entire configuration in `flake.nix` will quickly get unwieldy. So this is where we will be importing our own function for building users and systems.

```
{ # .dotfiles/flake.nix
#...

outputs = { nixpkgs, home-manager, ... }@inputs:
let
  inherit (nixpkgs) lib;

  util = import ./lib {
    inherit system pkgs home-manager lib; overlays = (pkgs.overlays);
  };

  inherit (util) user;
  inherit (util) host;

  pkgs = import nixpkgs {
    inherit system;
    config.allowUnfree = true;
    overlays = [];
  };
}
```



```
system = "x86_64-linux";
in {
  homeManagerConfigurations = {
    jd = user.mkHMUser {
      # ...
    };
  };
}

nixosConfigurations = {
  laptop = host.mkHost {
    # ...
  };
};
};

}
```

Nix Language

There is a lot to unpack here so I'll start with what the Nix Language is doing. If you are experienced with Nix, you can skip this section. I will go over the language briefly as we go along and provide links but I highly recommend James Fisher's [Nix by example](#) for learning Nix.

The `let` section:

- * The `output` is a [function](#) that is called with the inputs declared earlier.
- * We declare local variables using the `let ... in` syntax. Nix pills has a good [overview](#).
- * In our let we utilize `inherit` which lets us copy variables easily. The manual has a nice [description](#).
- * We create our own `pkgs` which imports `nixpkgs` with our configuration of choice. I install non-free packages like *obsidian* and *zoom-us* so I make sure it is allowed (see [wiki](#))
- * We import our custom functions using `import ./lib { #parameters }`. Importing is like calling a function, see the [Nix Pills](#)
- * We will not cover `overlays` in this post as they are not important



to the task at hand, but if you are interested check out the [wiki](#).

The `in` section:

- * We call our homemade imported functions `user.mkHUser` and `host.mkHost` with parameters. We will write out these functions and their parameters next.

Homemade Functions

Our homemade funtions `mkHost` and `mkHUser` are our abstraction away from base option settings. Our configuration options will be parameters to this function which will take them act on them build our system and home-manager flakes. You can have as many `homeManagerConfigurations` and `nixosConfigurations` as you want enabling you to have every OS config and User config in one reproducible github repository \(^▽^)/.

Library Functions

Before I jump into writing our functions, I have to credit the one and only **Wil Taylor** for providing the idea and original code (see [here](#))

./lib default.nix

When you import a directory, Nix automatically looks for a `default.nix` file to run. So this is where we will import our functions.

```
# .dotfiles/lib/default.nix
{ pkgs, home-manager, system, lib, overlays, ... }:
rec {
  user = import ./user.nix { inherit pkgs home-manager lib system overlays; };
  host = import ./host.nix { inherit system pkgs home-manager lib user; };
}
```



Notice we use `rec` around our imports. This lets us self reference so we can call `util.user` in our `flake.nix`. See [wiki](#).

System Configuration

We will start with learning how to make a system configuration. In this section I will introduce you to Nix Modules which are fundamental to NixOS.

mkHost

This function is what we use to make system configurations. There are three parts to the parameters, hardware/kernel options and the `systemConfig`, and `users`. The hardware/kernel options are mapped one-to-one to NixOS options, while `systemConfig` will be our abstracted configuration module.

```
# .dotfiles/lib/host.nix
{ system, pkgs, home-manager, lib, user, ... }:
with builtins;
{
  mkHost = { name, NICs, initrdMods, kernelMods, kernelParams, kernelPa
    systemConfig, cpuCores, users, wifi ? [],
    gpuTempSensor ? null, cpuTempSensor ? null
  }:
  let
    networkCfg = listToAttrs (map (n: {
      name = "${n}"; value = { useDHCP = true; };
    }) NICs);

    userCfg = {
      inherit name NICs systemConfig cpuCores gpuTempSensor cpuTempSens
    };

    sys_users = (map (u: user.mkSystemUser u) users);
  in lib.nixosSystem {
    inherit system;

    modules = [
      {
        imports = [ ..modules/system ] ++ sys_users;

        jd = systemConfig;

        environment.etc = {

```



```

    "hmsystemdata.json".text = toJSON userCfg;
};

networking.hostName = "${name}";
networking.interfaces = networkCfg;
networking.wireless.interfaces = wifi;

networking.networkmanager.enable = true;
networking.useDHCP = false;

boot.initrd.availableKernelModules = initrdMods;
boot.kernelModules = kernelMods;
boot.kernelParams = kernelParams;
boot.kernelPackages = kernelPackage;

nixpkgs.pkgs = pkgs;
nix.maxJobs = lib.mkDefault cpuCores;

system.stateVersion = "21.05";
}
];
};
}

```

with Statement

Another large piece of code to unpack! Starting at the beginning we have `with builtins;`. This lets us call builtin functions without using `builtins.**`. See [Nix Pills](#). All the builtin functions can be found in the [manual](#).

networkCfg

Next up is `networkCfg`. The `NICs` variable is a list of strings. So `builtins.map` turns the list of strings into a list of `{ name = "${nic}"; value = { useDHCP = true; } }`. Then we call `builtins.listToAttrs` which maps the list of name/value pairs to an attribute set `{ "${nic}" = { useDHCP = true; } }`.

Now in the output we have `networking.interfaces = networkCfg` (see



nixOS [options](#)). Our list of strings was turned into suitable networking interfaces. We have `useDHCP = true` so we can have IP addresses auto-assigned ([wikipedia](#)).

userCfg

`userCfg` doesn't actually configure anything. Instead it is used to pass information about the system to our user (aka home-manager) configurations. We use `builtins.toJSON` to save the attribute set to `/etc/hmsystemdata.json`. Passing data in between the system and user is important for setting up things like `.xinitrc` if system is not using a display manager.

sysUsers

All of our users need to be declared in the system configuration. Therefore we map our list of `user` attributes sets that we passed from `flake.nix` to the `user.mkSystemUser` function.

Now is a good time to set up our `user.nix` file with its first function.

```
# .dotfiles/lib/user.nix
{ pkgs, home-manager, lib, system, overlays, ... }:
with builtins;
{
  mkHMuUser = { # To be completed later };

  mkSystemUser = { name, groups, uid, shell, ... }:
  {
    users.users."${name}" = {
      name = name;
      isNormalUser = true;
      isSystemUser = false;
      extraGroups = groups;
      uid = uid;
      initialPassword = "helloworld";
      shell = shell;
    };
  };
}
```



{

These are some basic user settings. See [here](#) for info on `isNormalUser`, `isSystemUser`, and `uid`. We set an initial password but it should be changed immediately after setup. We set the shell to the chosen shell package. The groups is a list of any groups the user should be a part of.

Kernel Goodies, etc.

A whole bunch of kernel settings, some networking settings, etc. The philosophy behind not abstracting these settings is that they are essential to any system and should be explicitly chosen.

lib.nixosSystem

This is the function that produces our NixOS system flake. I was unable to find a high level description of what it does internally but if your interested here is the [source code](#). Warning: it is very complex. What follows is my understanding of what `lib.nixosSystem` does.

The structure is laid out by the [NixOS wiki](#). We set `system` and `modules`. Ooh! Our first code mentioning modules. NixOS takes in a list of module files. But what is a module!?

Modules!

According to the wiki `modules` are Nix files that declare *options* for other modules to *define*. But what exactly does that mean? It is illustrated well when you see the structure of a module.

```
{  
  imports = [  
    # paths to other modules  
  ];  
  
  options = {  
    # option declarations  
  };
```



```
config = {  
    # option definitions  
};  
}
```

Config

Working backwards lets start with the config. This is where you act on your configuration. Lets say a user set `wifi.enable = true`. If `wifi.enable = true`, then we need install packages and run systemd services (or whatever enabling wifi does). But wait, to do that we are setting other options. Thats why the NixOS wiki says other modules *define!* Each module acts on other modules. So then where are these options coming from? The options section!

Options

Every configuration option needs to be declared as an option. Attempting to access something in the config that isn't declared will result in an error. For all the declaration options check out, you guessed it, the [wiki](#).

Imports

As every option needs to be declared, we want modules to interact with each other. The imports section lets you bring in your other modules and combine them.

Back to lib.nixosSystem

Now that we have an understanding of Nix modules, lets re-examine what is happening with `modules`. If you have ever checked out [NixOS Options Search](#) you will see 10,000+ options advertised. NixOS is really just one massive community module for setting up a Linux system!

This means when we provide our NixOS configuration, we can also import in our own modules that define new options. Its as simple as



that: we provide our own high level options that set built-in NixOS options.

Imports

We import our custom module system from `./modules/system` and the config outputs from `mkSystemUser`. The `++` operator is just concatenating the lists. [Nix Operators](#)

jd & systemConfig

To prevent any accidental conflicts between the NixOS module system and my abstracted module system, all custom options are in the `jd.**` attribute set. However, from `flake.nix` you would never know this as I set `jd = systemConfig`. Now that you know how to make a system flake lets go back to our `flake.nix` and configure our first system!

System flake.nix

Here is an example laptop configuration that I have in use. I copied and pasted the `NICs`, `initrdMods`, and `kernelMods` from my system's auto-generated `hardware.nix` file. I am leaving `systemConfig` blank as that is your freedom spot to create your own settings. At the end of the blog I will have an example module for inspiration.

```
{# .dotfiles/flake.nix
laptop = host.mkHost {
  name = "laptop";
  NICs = [ "enp0s31f6" "wlp2s0" ];
  kernelPackage = pkgs.linuxPackages;
  initrdMods = [ "xhci_pci" "nvme" "usb_storage" "sd_mod" "rtsx_pci_sdm" ];
  kernelMods = [ "kvm-intel" ];
  kernelParams = [];
  systemConfig = {
    # your abstracted system config
  };
  users = [
    { name = "jd";
      groups = [ "wheel" "networkmanager" "video" ];
    }
  ];
}
```



```
    uid = 1000;
    shell = pkgs.zsh;
  }];
  cpuCores = 4;
};

}
```

User Configuration

Now that you know how Nix Modules work, this next part should be a breeze. `home-manager` is just a massive community module for configuring user environments rather than the operating system.

mkHmUser

`mkHmUser` is the function used to build our home manager flake. We have two parameters, `username` and `userConfig`. `userConfig` is the user equivalent of `systemConfig` which is our settings passed to our custom module.

```
# .dotfiles/lib/user.nix
{ pkgs, home-manager, lib, system, overlays, ... }:
with builtins;
{
  mkHmUser = {userConfig, username}:
    home-manager.lib.homeManagerConfiguration {
      inherit system username pkgs;
      stateVersion = "21.05";
      configuration =
        let
          trySettings = tryEval (fromJSON (readFile /etc/hmsystemdata.j
          machineData = if trySettings.success then trySettings.value e
          machineModule = { pkgs, config, lib, ... }: {
            options.machineData = lib.mkOption {
              default = {};
              description = "Settings passed from nixos system configur
            };
            config.machineData = machineData;
          };
        };
      
```



```
};

in {
    jd = userConfig;

    nixpkgs.overlays = overlays;
    nixpkgs.config.allowUnfree = true;

    systemd.user.startServices = true;
    home.stateVersion = "21.05";
    home.username = username;
    home.homeDirectory = "/home/${username}";

    imports = [ .. /modules/users machineModule ];
};

homeDirectory = "/home/${username}";
};

# ...

}
```

machineModule

This is our first encounter of a custom built module! It is very simple but shows how all `config.***` need to be declared. While we never manually change what `config.machineData` is, it still needs to be declared as an option. The point of `machineModule` is so we can access our exported `hmsystemdata.json` in our custom modules. When accessing the data from our custom modules, all we need to do is access `config.machineData.***` !

Home Manager Flake

I am using the function `home-manager.lib.homeManagerConfiguration` from `home-manager`'s `flake.nix`. I copied the relevant code below.



```
{ # home-manager flake.nix
homeManagerConfiguration = { configuration, system, homeDirectory
, username, extraModules ? [ ], extraSpecialArgs ? { }
, pkgs ? builtins.getAttr system nixpkgs.outputs.legacyPackages
, check ? true, stateVersion ? "20.09" }@args:
```

```
assert nixpkgs.lib.versionAtLeast stateVersion "20.09";

import ./modules {
  inherit pkgs check extraSpecialArgs;
  configuration = { ... }: {
    imports = [ configuration ] ++ extraModules;
    home = { inherit homeDirectory stateVersion username; };
    nixpkgs = { inherit (pkgs) config overlays; };
  };
};

}
```

The `home-manager` function `home-manager.lib.homeManagerConfiguration` is similar to `lib.nixosSystem`. Rather than a list of `modules` it takes in a single `configuration`. But the outcome is the same, we can import our custom module at `.dotfiles/modules/users` and `machineModule` to expand the options. Just like with the system config, we set `jd = userConfig`.

Make sure to pass in the other parameters (`system`, `homeDirectory`, `username`, and `pkgs`). As you can see, home manager defaults to `stateVersion = "21.09"` so make sure to set `stateVersion = "21.05"` if it is 21.05 or else an error will occur.

That was much faster to get through then the system configuration setting because the concepts repeat! Its now time to start writing your own modules for configuration.

Writing Modules

When writing your own modules you will be abstracting existing NixOS and `home-manager` modules. Therefore you should have all the options handy. Use [NixOS Search](#) when doing system configurations, and [home-manager Appendix A](#) when doing user configurations.

If you want inspiration for modules to write check out my [modules folder](#). A good understanding of Nix and the builtin functions makes writing modules much easier. A great resource for builtin functions is teu5us' [website](#). Also don't be afraid to google and read `nixpkgs` github source. Most of the time its not actually that complicated.



Example Module

This module is taken from my user setting for git.

```
# .dotfiles/modules/users/default.nix
{ pkgs, config, lib, ... }:

{
  imports = [
    ./git
  ];
}
```

We are utilizing module imports to group each module file into nice categories. Remember that when importing a directory, Nix looks for `default.nix`

```
# .dotfiles/modules/users/git/default.nix
{ pkgs, config, lib, ... }:
with lib;

let
  cfg = config.jd.git;
in {
  options.jd.git = {
    enable = mkOption {
      description = "Enable git";
      type = types.bool;
      default = false;
    };
    userName = mkOption {
      description = "Name for git";
      type = types.str;
      default = "Jordan Isaacs";
    };
    userEmail = mkOption {
      description = "Email for git";
    };
  };
}
```



```
type = types.str;
default = "github@jdisaacs.com";
};

};

config = mkIf (cfg.enable) {
  programs.git = {
    enable = true;
    userName = cfg.userName;
    userEmail = cfg.userEmail;
    extraConfig = {
      credential.helper = "${pkgs.git.override { withLibsecret = true; }/bin/git-credential-libsecret";
    };
  };
};

};

}
```

I create a shortcut to the current config, by setting `cfg = config.jd.git`.

There is an `enable` option that allows us to easily turn on and off git. This way instead of having to remove any custom configuration, just set `enable = false` the attribute set won't be made anymore due to `mkIf`. The `userName` and `userEmail` have defaults that I should never have to change, but are there in case I do. And I have git setup to always use my keyring (what is a keyring, and setting it up will be in a future blogpost!) instead of plaintext (from [wiki](#)).

Now from `flake.nix` all we have to do is the following. Notice the `jd.git.**` is hidden.

```
{ # flake.nix
  jd = user.mkHmUser {
    userConfig = {
      git.enable = true;
    };
  };
};
```



```
}
```

Flakes Actions

Now that you have written the modules you want, you probably are wondering how you actually interact with your flakes. However, before we can do that we need to get a git repository set up.

Git & Flakes

Nix requires your flakes to be in a git repository and all new files that are accessed by the flake to be staged (but not necessarily committed). Attempting to access a newly created file that was not staged will result in the flake saying the file was not found. However, once the file is staged any changes can be made to it without staging or committing. If you do not commit you will see a warning: `warning: Git tree '*dir-path*' is dirty`. It is just letting you know you are building a flake with uncommitted changes.

Applying Flake

While writing modules is fun, the whole point is to create a usable system! That is only possible by building and activating your flakes.

System

Applying your system flakes with your system is as easy as appending `--flake '.'` to your `sudo nixos-rebuild **` command. Make sure to run the command when your working directory is `.dotfiles` or wherever your flake is located.

User

Applying your user flake is a little more complicated. First you need to build the flake then activate it.



```
nix build --impure .#homeManagerConfigurations.$USER.activationPackage  
./result/activate
```

`--impure` is required because we are importing `hmsystemdata.json`.

It means we are not *fully* reproducible because the file (and thus output) could change without updating the flake.

Updating Flake

As our system and user are all in the same flake, updating the flake is as easy as calling `nix flake update` when in the directory.

Cleaning System

It is still a normal NixOS system so you can call `nix-store --gc` and `nix-store --optimize`.

Finishing up

© 2022 Jordan Isaacs :: Powered by [Zola](#)

[Like this site?](#) [see source code here](#)

Well now you hopefully have a functioning NixOS and home-manager flake with a better understanding of Nix! If you have any questions, or find any issues with anything in this post feel free to leave a github issue at my [dotfiles repo](#)



NixOS /
nixpkgs

Code

Issues 5k+

Pull requests 5k+

Actions

Projects 35

Sec



dadada nixos/doc: add documentation on using FIDO2 toke...



4 months ago



115 lines (88 loc) · 3.94 KB

[nixpkgs / nixos / doc / manual / configuration / luks-file-systems.section.md](#)

↑ Top

Preview

Code Blame

Raw



LUKS-Encrypted File Systems {#sec-luks-file-systems}

NixOS supports file systems that are encrypted using *LUKS* (Linux Unified Key Setup). For example, here is how you create an encrypted Ext4 file system on the device `/dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d` :

```
# cryptsetup luksFormat /dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d
```

WARNING!

=====

This will overwrite data on `/dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d`

Are you sure? (Type uppercase yes): YES

Enter LUKS passphrase: ***

Verify passphrase: ***

```
# cryptsetup luksOpen /dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d
```

Enter passphrase for `/dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d`

```
# mkfs.ext4 /dev/mapper/crypted
```



The LUKS volume should be automatically picked up by `nixos-generate-config`, but you might want to verify that your `hardware-configuration.nix` looks correct. To manually ensure that the system is automatically mounted at boot time as `/`, add the following to `configuration.nix`:

```
boot.initrd.luks.devices.crypted.device = "/dev/disk/by-uuid/3f6b0024"   
fileSystems."/".device = "/dev/mapper/crypted"; 
```

Should grub be used as bootloader, and `/boot` is located on an encrypted partition, it is necessary to add the following grub option:

```
boot.loader.grub.enableCryptodisk = true; 
```

FIDO2 {#sec-luks-file-systems-fido2}

NixOS also supports unlocking your LUKS-Encrypted file system using a FIDO2 compatible token.

Without systemd in initrd {#sec-luks-file-systems-fido2-legacy}

In the following example, we will create a new FIDO2 credential and add it as a new key to our existing device `/dev/sda2`:

```
# export FID02_LABEL="/dev/sda2 @ $HOSTNAME"  
# fido2luks credential "$FID02_LABEL"  
f1d00200108b9d6e849a8b388da457688e3dd653b4e53770012d8f28e5d3b26986503  
  
# fido2luks -i add-key /dev/sda2 f1d00200108b9d6e849a8b388da457688e3d  
Password:  
Password (again):  
Old password:  
Old password (again):  
Added to key to device /dev/sda2, slot: 2 
```

To ensure that this file system is decrypted using the FIDO2 compatible key, add the following to `configuration.nix`:

```
boot.initrd.luks.fido2Support = true;  
boot.initrd.luks.devices."/dev/sda2".fido2.credential = "f1d00200108b   
2" 
```

You can also use the FIDO2 passwordless setup, but for security reasons, you might want to enable it only when your device is PIN protected, such as [Trezor](#).

```
boot.initrd.luks.devices."/dev/sda2".fido2.passwordLess = true;
```



systemd Stage 1 {#sec-luks-file-systems-fido2-systemd}

If systemd stage 1 is enabled, it handles unlocking of LUKS-encrypted volumes during boot. The following example enables systemd stage1 and adds support for unlocking the existing LUKS2 volume `root` using any enrolled FIDO2 compatible tokens.

```
boot.initrd = {
  luks.devices.root = {
    crypttabExtraOpts = [ "fido2-device=auto" ];
    device = "/dev/sda2";
  };
  systemd.enable = true;
};
```



All tokens that should be used for unlocking the LUKS2-encrypted volume must first be enrolled using [systemd-cryptenroll](#). In the following example, a new key slot for the first discovered token is added to the LUKS volume.

```
# systemd-cryptenroll --fido2-device=auto /dev/sda2
```



Existing key slots are left intact, unless `--wipe-slot=` is specified. It is recommended to add a recovery key that should be stored in a secure physical location and can be entered wherever a password would be entered.

```
# systemd-cryptenroll --recovery-key /dev/sda2
```



 [NixOS / nix](#) Public

<> [Code](#) (2.7k) Issues 2.7k Pull requests 360 Actions Projects 2 Security

 [iFreilicht](#) docs: clarify flake types and implied defaults ✓ 4 months ago ... ↻

 706 lines (544 loc) · 23.3 KB

[Preview](#) Code Blame Raw Copy Download Edit ☰

```
R""(
```

Description

`nix flake` provides subcommands for creating, modifying and querying *Nix flakes*. Flakes are the unit for packaging Nix code in a reproducible and discoverable way. They can have dependencies on other flakes, making it possible to have multi-repository Nix projects.

A flake is a filesystem tree (typically fetched from a Git repository or a tarball) that contains a file named `flake.nix` in the root directory. `flake.nix` specifies some metadata about the flake such as dependencies (called *inputs*), as well as its *outputs* (the Nix values such as packages or NixOS modules provided by the flake).

Flake references

Flake references (*flakeref*s) are a way to specify the location of a flake. These have two different forms:

Attribute set representation

Example:

```
{  
  type = "github";  
  owner = "NixOS";  
  repo = "nixpkgs";  
}
```



The only required attribute is `type`. The supported types are listed below.

URL-like syntax

Example:

```
github:NixOS/nixpkgs
```



These are used on the command line as a more convenient alternative to the attribute set representation. For instance, in the command

```
# nix build github:NixOS/nixpkgs#hello
```



`github:NixOS/nixpkgs` is a flake reference (while `hello` is an output attribute). They are also allowed in the `inputs` attribute of a flake, e.g.

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



is equivalent to

```
inputs.nixpkgs = {  
  type = "github";  
  owner = "NixOS";  
  repo = "nixpkgs";  
};
```



Following [RFC 3986](#), characters outside of the allowed range (i.e. neither [reserved characters](#) nor [unreserved characters](#)) must be percent-encoded.



Examples























Here are some examples of flake references in their URL-like representation:

- `nixpkgs` : The `nixpkgs` entry in the flake registry.
- `nixpkgs/a3a3dda3bacf61e8a39258a0ed9c924eeeca8e293` : The `nixpkgs` entry in the flake registry, with its Git revision overridden to a specific value.
- `github:NixOS/nixpkgs` : The `master` branch of the `NixOS/nixpkgs` repository

on GitHub.

- `github:NixOS/nixpkgs/nixos-20.09` : The `nixos-20.09` branch of the `nixpkgs` repository.
- `github:NixOS/nixpkgs/a3a3dda3bacf61e8a39258a0ed9c924eeaca8e293` : A specific revision of the `nixpkgs` repository.
- `github:edolstra/nix-warez?dir=blender` : A flake in a subdirectory of a GitHub repository.
- `git+https://github.com/NixOS/patchelf` : A Git repository.
- `git+https://github.com/NixOS/patchelf?ref=master` : A specific branch of a Git repository.
- `git+https://github.com/NixOS/patchelf?ref=master&rev=f34751b88bd07d7f44f5cd3200fb4122bf916c7e` : A specific branch *and* revision of a Git repository.
- `https://github.com/NixOS/patchelf/archive/master.tar.gz` : A tarball flake.

Path-like syntax

Flakes corresponding to a local path can also be referred to by a direct path reference, either `/absolute/path/to/the/flake` or `./relative/path/to/the/flake`. Note that the leading `./` is mandatory for relative paths. If it is omitted, the path will be interpreted as [URL-like syntax](#), which will cause error messages like this:

```
error: cannot find flake 'flake:relative/path/to/the/flake' in the fl ↗
```

The semantic of such a path is as follows:

- If the directory is part of a Git repository, then the input will be treated as a `git+file:` URL, otherwise it will be treated as a `path: url`;
- If the directory doesn't contain a `flake.nix` file, then Nix will search for such a file upwards in the file system hierarchy until it finds any of:
 - i. The Git repository root, or
 - ii. The filesystem root (`/`), or
 - iii. A folder on a different mount point.

Contrary to URL-like references, path-like flake references can contain arbitrary unicode characters (except `#` and `?`).



Examples

- `..` : The flake to which the current directory belongs to.
- `/home/alice/src/patchelf` : A flake in some other directory.
- `./.../sub directory/with Üñî©ôð€` : A flake in another relative directory that has Unicode characters in its name.

Flake reference attributes

The following generic flake reference attributes are supported:

- `dir` : The subdirectory of the flake in which `flake.nix` is located. This parameter enables having multiple flakes in a repository or tarball. The default is the root directory of the flake.
- `narHash` : The hash of the NAR serialisation (in SRI format) of the contents of the flake. This is useful for flake types such as tarballs that lack a unique content identifier such as a Git commit hash.

In addition, the following attributes are common to several flake reference types:

- `rev` : A Git or Mercurial commit hash.
- `ref` : A Git or Mercurial branch or tag name.

Finally, some attribute are typically not specified by the user, but can occur in *locked* flake references and are available to Nix code:

- `revCount` : The number of ancestors of the commit `rev`.
- `lastModified` : The timestamp (in seconds since the Unix epoch) of the last modification of this version of the flake. For Git/Mercurial flakes, this is the commit time of commit `rev`, while for tarball flakes, it's the most recent timestamp of any file inside the tarball.

Types

Currently the `type` attribute can be one of the following:

- `indirect` : *The default*. Indirection through the flake registry. These have the form



```
[flake:]<flake-id>(/<rev-or-ref>(/rev)?)?
```



These perform a lookup of `<flake-id>` in the flake registry. For example, `nixpkgs` and `nixpkgs/release-20.09` are indirect flake references. The specified `rev` and/or `ref` are merged with the entry in the registry; see [nix registry](#) for details.

For example, these are valid indirect flake references:

- `nixpkgs`
- `nixpkgs/nixos-unstable`
- `nixpkgs/a3a3dda3bacf61e8a39258a0ed9c924eeeca8e293`
- `nixpkgs/nixos-unstable/a3a3dda3bacf61e8a39258a0ed9c924eeeca8e293`
- `sub/dir` (if a flake named `sub` is in the registry)
- `path` : arbitrary local directories. The required attribute `path` specifies the path of the flake. The URL form is

```
path:<path>(\?<params>)?
```



where `path` is an absolute path to a directory in the file system containing a file named `flake.nix`.

If the flake at `path` is not inside a git repository, the `path:` prefix is implied and can be omitted.

`path` generally must be an absolute path. However, on the command line, it can be a relative path (e.g. `.` or `./foo`) which is interpreted as relative to the current directory. In this case, it must start with `.` to avoid ambiguity with registry lookups (e.g. `nixpkgs` is a registry lookup; `./nixpkgs` is a relative path).

For example, these are valid path flake references:

- `path:/home/user/sub/dir`
- `/home/user/sub/dir` (if `dir/flake.nix` is *not* in a git repository)
- `./sub/dir` (when used on the command line and `dir/flake.nix` is *not* in a git repository)



- `git` : Git repositories. The location of the repository is specified by the attribute `url`.

They have the URL form

```
git(+http|+https|+ssh|+git|+file):(//<server>)?:<path>  
(\?<params>)?
```



If `path` starts with `/` (or `./` when used as an argument on the command line) and is a local path to a git repository, the leading `git:` or `+file` prefixes are implied and can be omitted.

The `ref` attribute defaults to resolving the `HEAD` reference.

The `rev` attribute must denote a commit that exists in the branch or tag specified by the `ref` attribute, since Nix doesn't do a full clone of the remote repository by default (and the Git protocol doesn't allow fetching a `rev` without a known `ref`). The default is the commit currently pointed to by `ref`.

When `git+file` is used without specifying `ref` or `rev`, files are fetched directly from the local `path` as long as they have been added to the Git repository. If there are uncommitted changes, the reference is treated as dirty and a warning is printed.

For example, the following are valid Git flake references:

- `git:/home/user/sub/dir`
- `/home/user/sub/dir` (if `dir/flake.nix` is in a git repository)
- `./sub/dir` (when used on the command line and `dir/flake.nix` is in a git repository)
- `git+https://example.org/my/repo`
- `git+https://example.org/my/repo?dir=flake1`
- `git+ssh://git@github.com:NixOS/nix?ref=v1.2.3`
- `git://github.com/edolstra/dwarfss?ref=unstable&rev=e486d8d40e626a20e06d792db8cc5ac5aba9a5b4`
- `git+file:///home/my-user/some-repo/some-repo`

-



`mercurial` : Mercurial repositories. The URL form is similar to the `git` type, except that the URL schema must be one of `hg+http` , `hg+https` , `hg+ssh` or `hg+file` .

- `tarball` : Tarballs. The location of the tarball is specified by the attribute `url` .

In URL form, the schema must be `tarball+http://` , `tarball+https://` or `tarball+file://` . If the extension corresponds to a known archive format (`.zip` , `.tar` , `.tgz` , `.tar.gz` , `.tar.xz` , `.tar.bzz` or `.tar.zst`), then the `tarball+` can be dropped.

- `file` : Plain files or directory tarballs, either over http(s) or from the local disk.

In URL form, the schema must be `file+http://` , `file+https://` or `file+file://` . If the extension doesn't correspond to a known archive format (as defined by the `tarball` fetcher), then the `file+` prefix can be dropped.

- `github` : A more efficient way to fetch repositories from GitHub. The following attributes are required:

- `owner` : The owner of the repository.
- `repo` : The name of the repository.

These are downloaded as tarball archives, rather than through Git. This is often much faster and uses less disk space since it doesn't require fetching the entire history of the repository. On the other hand, it doesn't allow incremental fetching (but full downloads are often faster than incremental fetches!).

The URL syntax for `github` flakes is:

`github:<owner>/<repo>(/<rev-or-ref>)?(\\?<params>)?`



`<rev-or-ref>` specifies the name of a branch or tag (`ref`), or a commit hash (`rev`). Note that unlike Git, GitHub allows fetching by commit hash without specifying a branch or tag.



You can also specify `host` as a parameter, to point to a custom GitHub Enterprise server.

Some examples:

- `github:edolstra/dwarfss`
- `github:edolstra/dwarfss/unstable`
- `github:edolstra/dwarfss/d3f2baba8f425779026c6ec04021b2e927f61e31`
- `github:internal/project?host=company-github.example.org`
- `gitlab` : Similar to `github`, is a more efficient way to fetch GitLab repositories. The following attributes are required:
 - `owner` : The owner of the repository.
 - `repo` : The name of the repository.

Like `github`, these are downloaded as tarball archives.

The URL syntax for `gitlab` flakes is:

`gitlab:<owner>/<repo>(/<rev-or-ref>)?(\?<params>)?`

`<rev-or-ref>` works the same as `github`. Either a branch or tag name (`ref`), or a commit hash (`rev`) can be specified.

Since GitLab allows for self-hosting, you can specify `host` as a parameter, to point to any instances other than `gitlab.com`.

Some examples:

- `gitlab:veloren/veloren`
- `gitlab:veloren/veloren/master`
- `gitlab:veloren/veloren/80a4d7f13492d916e47d6195be23acae8001985a`
- `gitlab:openldap/openldap?host=git.openldap.org`

When accessing a project in a (nested) subgroup, make sure to URL-encode any slashes, i.e. replace `/` with `%2F`:

- `gitlab:veloren%2Fdev/rfc5`
- `sourcehut` : Similar to `github`, is a more efficient way to fetch SourceHut repositories. The following attributes are required:
 - `owner` : The owner of the repository (including leading `~`).
 - `repo` : The name of the repository.



Like `github`, these are downloaded as tarball archives.

The URL syntax for `sourcehut` flakes is:

```
sourcehut:<owner>/<repo>(/<rev-or-ref>)?(\?<params>)?
```

`<rev-or-ref>` works the same as `github`. Either a branch or tag name (`ref`), or a commit hash (`rev`) can be specified.

Since SourceHut allows for self-hosting, you can specify `host` as a parameter, to point to any instances other than `git.sr.ht`.

Currently, `ref` name resolution only works for Git repositories. You can refer to Mercurial repositories by simply changing `host` to `hg.sr.ht` (or any other Mercurial instance). With the caveat that you must explicitly specify a commit hash (`rev`).

Some examples:

- `sourcehut:~misterio/nix-colors`
- `sourcehut:~misterio/nix-colors/main`
- `sourcehut:~misterio/nix-colors?host=git.example.org`
- `sourcehut:~misterio/nix-colors
/182b4b8709b8ffe4e9774a4c5d6877bf6bb9a21c`
- `sourcehut:~misterio/nix-colors
/21c1a380a6915d890d408e9f22203436a35bb2de?host=hg.sr.ht`

Flake format

As an example, here is a simple `flake.nix` that depends on the `Nixpkgs` flake and provides a single package (i.e. an [installable](#) derivation):



```
{  
    description = "A flake for building Hello World";  
  
    inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-20.03";  
  
    outputs = { self, nixpkgs }: {  
  
        packages.x86_64-linux.default =  
            # Notice the reference to nixpkgs here.  
            with import nixpkgs { system = "x86_64-linux"; };  
            stdenv.mkDerivation {  
                name = "hello";  
                src = self;  
                buildPhase = "gcc -o hello ./hello.c";  
                installPhase = "mkdir -p $out/bin; install -t $out/bin hello"  
            };  
    };  
}
```

The following attributes are supported in `flake.nix`:

- `description` : A short, one-line description of the flake.
- `inputs` : An attrset specifying the dependencies of the flake (described below).
- `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. Thus, in the example above, `inputs.nixpkgs` contains the result of the call to the `outputs` function of the `nixpkgs` flake.

In addition to the outputs of each input, each input in `inputs` also contains some metadata about the inputs. These are:

- `outPath` : The path in the Nix store of the flake's source tree. This way, the attribute set can be passed to `import` as if it was a path, as in the example above (`import nixpkgs`).
- `rev` : The commit hash of the flake's repository, if applicable.
- `revCount` : The number of ancestors of the revision `rev`. This is not available for `github` repositories, since they're fetched as tarballs rather than as Git repositories.

- `lastModifiedDate` : The commit time of the revision `rev`, in the format `%Y%m%d%H%M%S` (e.g. `20181231100934`). Unlike `revCount`, this is available for both Git and GitHub repositories, so it's useful for generating (hopefully) monotonically increasing version strings.
- `lastModified` : The commit time of the revision `rev` as an integer denoting the number of seconds since 1970.
- `narHash` : The SHA-256 (in SRI format) of the NAR serialization of the flake's source tree.

The value returned by the `outputs` function must be an attribute set. The attributes can have arbitrary values; however, various `nix` subcommands require specific attributes to have a specific value (e.g. `packages.x86_64-linux` must be an attribute set of derivations built for the `x86_64-linux` platform).

- `nixConfig` : a set of `nix.conf` options to be set when evaluating any part of a flake. In the interests of security, only a small set of options is allowed to be set without confirmation so long as [`accept-flake-config`](#) is not enabled in the global configuration:
 - [`bash-prompt`](#)
 - [`bash-prompt-prefix`](#)
 - [`bash-prompt-suffix`](#)
 - [`flake-registry`](#)
 - [`commit-lockfile-summary`](#)

Flake inputs

The attribute `inputs` specifies the dependencies of a flake, as an attrset mapping input names to flake references. For example, the following specifies a dependency on the `nixpkgs` and `import-cargo` repositories:



```
# A GitHub repository.  
inputs.import-cargo = {  
    type = "github";  
    owner = "edolstra";  
    repo = "import-cargo";  
};  
  
# An indirection through the flake registry.  
inputs.nixpkgs = {  
    type = "indirect";  
    id = "nixpkgs";  
};
```



Alternatively, you can use the URL-like syntax:

```
inputs.import-cargo.url = "github:edolstra/import-cargo";  
inputs.nixpkgs.url = "nixpkgs";
```



Each input is fetched, evaluated and passed to the `outputs` function as a set of attributes with the same name as the corresponding input. The special input named `self` refers to the outputs and source tree of *this* flake. Thus, a typical `outputs` function looks like this:

```
outputs = { self, nixpkgs, import-cargo }: {  
    ... outputs ...  
};
```



It is also possible to omit an input entirely and *only* list it as expected function argument to `outputs`. Thus,

```
outputs = { self, nixpkgs }: ...;
```



without an `inputs.nixpkgs` attribute is equivalent to



```
inputs.nixpkgs = {  
    type = "indirect";  
    id = "nixpkgs";  
};
```



Repositories that don't contain a `flake.nix` can also be used as inputs, by setting the input's `flake` attribute to `false`:

```
inputs.grcov = {  
    type = "github";  
    owner = "mozilla";  
    repo = "grcov";  
    flake = false;  
};  
  
outputs = { self, nixpkgs, grcov }: {  
    packages.x86_64-linux.grcov = stdenv.mkDerivation {  
        src = grcov;  
        ...  
    };  
};
```



Transitive inputs can be overridden from a `flake.nix` file. For example, the following overrides the `nixpkgs` input of the `nixops` input:

```
inputs.nixops.inputs.nixpkgs = {  
    type = "github";  
    owner = "my-org";  
    repo = "nixpkgs";  
};
```



It is also possible to "inherit" an input from another input. This is useful to minimize flake dependencies. For example, the following sets the `nixpkgs` input of the top-level flake to be equal to the `nixpkgs` input of the `dwarffs` input of the top-level flake:

```
inputs.nixpkgs.follows = "dwarffs/nixpkgs";
```



The value of the `follows` attribute is a `/`-separated sequence of input names denoting the path of inputs to be followed from the root flake.



Overrides and `follows` can be combined, e.g.

```
inputs.nixops.inputs.nixpkgs.follows = "dwarffs/nixpkgs";
```



sets the `nixpkgs` input of `nixops` to be the same as the `nixpkgs` input of `dwarf`. It is worth noting, however, that it is generally not useful to eliminate transitive `nixpkgs` flake inputs in this way. Most flakes provide their functionality through Nixpkgs overlays or NixOS modules, which are composed into the top-level flake's `nixpkgs` input; so their own `nixpkgs` input is usually irrelevant.

Lock files

Inputs specified in `flake.nix` are typically "unlocked" in the sense that they don't specify an exact revision. To ensure reproducibility, Nix will automatically generate and use a *lock file* called `flake.lock` in the flake's directory. The lock file contains a graph structure isomorphic to the graph of dependencies of the root flake. Each node in the graph (except the root node) maps the (usually) unlocked input specifications in `flake.nix` to locked input specifications. Each node also contains some metadata, such as the dependencies (outgoing edges) of the node.

For example, if `flake.nix` has the inputs in the example above, then the resulting lock file might be:



```
{  
  "version": 7,  
  "root": "n1",  
  "nodes": {  
    "n1": {  
      "inputs": {  
        "nixpkgs": "n2",  
        "import-cargo": "n3",  
        "grcov": "n4"  
      }  
    },  
    "n2": {  
      "inputs": {},  
      "locked": {  
        "owner": "edolstra",  
        "repo": "nixpkgs",  
        "rev": "7f8d4b088e2df7fdb6b513bc2d6941f1d422a013",  
        "type": "github",  
        "lastModified": 1580555482,  
        "narHash": "sha256-OnpEWzNxF/AU4KlqBXM2s5PWvfI5/BS6xQrPvkF5t0  
      },  
      "original": {  
        "id": "nixpkgs",  
        "type": "indirect"  
      }  
    },  
    "n3": {  
      "inputs": {},  
      "locked": {  
        "owner": "edolstra",  
        "repo": "import-cargo",  
        "rev": "8abf7b3a8cbe1c8a885391f826357a74d382a422",  
        "type": "github",  
        "lastModified": 1567183309,  
        "narHash": "sha256-wIXWOpX9rRjK5NDsL6WzuuBJl2R0kUCnlpZUrASyks  
      },  
      "original": {  
        "owner": "edolstra",  
        "repo": "import-cargo",  
        "type": "github"  
      }  
    },  
    "n4": {  
      "inputs": {},  
      "locked": {  
        "owner": "mozilla",  
        "repo": "grcov",  
        "rev": "989a84bb29e95e392589c4e73c29189fd69a1d4e",  
        "type": "github"  
      }  
    }  
  }  
}
```



```
        "type": "github",
        "lastModified": 1580729070,
        "narHash": "sha256-235uMxYlHxJ5y92EXZWAYEsEb6mm+b069GAd+B0I0x
    },
    "original": {
        "owner": "mozilla",
        "repo": "grcov",
        "type": "github"
    },
    "flake": false
}
}
}
```

This graph has 4 nodes: the root flake, and its 3 dependencies. The nodes have arbitrary labels (e.g. `n1`). The label of the root node of the graph is specified by the `root` attribute. Nodes contain the following fields:

- `inputs` : The dependencies of this node, as a mapping from input names (e.g. `nixpkgs`) to node labels (e.g. `n2`).
- `original` : The original input specification from `flake.lock`, as a set of `builtins.fetchTree` arguments.
- `locked` : The locked input specification, as a set of `builtins.fetchTree` arguments. Thus, in the example above, when we build this flake, the input `nixpkgs` is mapped to revision `7f8d4b088e2df7fdb6b513bc2d6941f1d422a013` of the `edolstra/nixpkgs` repository on GitHub.

It also includes the attribute `narHash`, specifying the expected contents of the tree in the Nix store (as computed by `nix hash-path`), and may include input-type-specific attributes such as the `lastModified` or `revCount`. The main reason for these attributes is to allow flake inputs to be substituted from a binary cache: `narHash` allows the store path to be computed, while the other attributes are necessary because they provide information not stored in the store path.

- `flake` : A Boolean denoting whether this is a flake or non-flake dependency. Corresponds to the `flake` attribute in the `inputs` attribute in `flake.nix`.

The `original` and `locked` attributes are omitted for the root node. This is because we cannot record the commit hash or content hash of the root flake, since modifying `flake.lock` will invalidate these.



The graph representation of lock files allows circular dependencies between flakes.

For example, here are two flakes that reference each other:

```
{  
    inputs.b = ... location of flake B ...;  
    # Tell the 'b' flake not to fetch 'a' again, to ensure its 'a' is  
    # *this* 'a'.  
    inputs.b.inputs.a.follows = "";  
    outputs = { self, b }: {  
        foo = 123 + b.bar;  
        xyzzy = 1000;  
    };  
}
```

and

```
{  
    inputs.a = ... location of flake A ...;  
    inputs.a.inputs.b.follows = "";  
    outputs = { self, a }: {  
        bar = 456 + a.xyzzy;  
    };  
}
```

Lock files transitively lock direct as well as indirect dependencies. That is, if a lock file exists and is up to date, Nix will not look at the lock files of dependencies. However, lock file generation itself *does* use the lock files of dependencies by default.

)""





RESEARCH

PROGRAMMING

ARTIFICIAL INTELLIGENCE

INTERVIEWS

OTHER

Practical Nix Flakes

Article by Alexander Bantyev
May 3rd, 2021

19 min read



191

Flakes are a new feature in the Nix ecosystem. Flakes replace stateful channels (which cause much confusion among novices) and introduce a more intuitive and consistent CLI, making them a perfect opportunity to start using Nix.

This post is a quick introduction to Nix itself and the flakes feature specifically. It contains examples and advice on using flakes for a real-life use case: building applications in various languages.

What is Nix?



Briefly speaking, Nix is a package manager and a build system. Its most important aspect is allowing to write declarative scripts for reproducible software builds. It also helps to test and deploy software systems while using the functional programming paradigm. There is a vast repository of packages for Nix called [nixpkgs](#), and a GNU/Linux distribution that extends the ideas of Nix to the OS level called [NixOS](#).

Nix building instructions are called “derivations” and are written in the [Nix programming language](#).

Derivations can be written for packages or even entire systems. After that, they can then be deterministically “realised” (built) via Nix, the package manager. Derivations can only depend on a pre-defined set of inputs, so they are somewhat reproducible.

You can read more about the benefits of Nix in my blog post [on Nix](#).



What are Nix flakes?



Flakes are self-contained units that have inputs (dependencies) and outputs (packages, deployment instructions, Nix functions for use in other flakes). You can think about them as Rust crates or Go modules but language-independent. Flakes have great reproducibility because they are only allowed to depend on their inputs and they pin the exact versions of said inputs in a lockfile.

If you're already familiar with Nix, flakes are to Nix expressions what derivations are to build instructions.



Getting started with Nix

In order to do anything with flakes, you will first have to get “unstable” Nix up and running on your machine. Don’t mind that it is called unstable: it is not generally dangerous to run on your machine, it simply changes more often than “stable”. The easiest way is to use the official installer, which will work on any Linux distro, macOS, or Windows Subsystem for Linux:

```
curl -L https://nixos.org/nix/install | sh
```

Follow the instructions until you have Nix working on your machine, and then update to unstable with:

```
nix-env -f '<nixpkgs>' -iA nixUnstable
```

And enable experimental features with:

```
mkdir -p ~/.config/nix
echo 'experimental-features = nix-command flakes' >> ~/.config/nix/nix.conf
```

If you are using NixOS or have some trouble with installation, consult the NixOS wiki: <https://nixos.wiki/w/index.php?title=Flakes&oldid=100000>

Getting a feel for flakes

Now that you have a “flaky” Nix installed, it’s time to use it!

nix shell

First, let’s enter a shell that has GNU Hello from nixpkgs’ branch `nixpkgs-unstable` in it:

```
nix shell github:nixos/nixpkgs/nixpkgs-unstable#hello
```



Note that this will start the same shell as you are running, but add a directory containing the `hello` executable to your `$PATH`. The shell shouldn’t look any different from how it was outside the `nix shell`, so don’t panic if it looks like nothing is happening! The executable is not installed anywhere per se, it gets downloaded and unpacked in what you can consider a cache directory.

Now, inside that shell, try running `hello`.



through what this command does. `nix shell` is a nix subcommand that is used to run a shell with some packages available in `$PATH`. Those packages can be specified as arguments in the “installable” format. Each installable contains two parts: the URL (`github:nixos/nixpkgs/master` in this case) and an “attribute path” (`hello` here).



There are a few URL schemes supported:

- `github:owner/repo/[revision or branch]` and `gitlab:owner/repo/[revision or branch]` (for public repositories on [github.com](#) and [gitlab.com](#); note that the branch name cannot contain slashes).
- `https://example.com/path/to/tarball.tar.gz` for tarballs.
- `git+https://example.com/path/to/repo.git` and `git+ssh://example.com/path/to/repo.git` for plain git repositories (you can, of course, use this for GitHub and GitLab). You can specify the branch or revision by adding `?ref=<branch name here>`.
- `file:///path/to/directory` or `/path/to/directory` or `./path/to/relative/directory` for a local directory.
- `flake-registry-value` for a value from a flake registry (I won’t talk about flake registries in this article).

So, there are some other ways to get the same shell:

```
nix shell https://github.com/nixos/nixpkgs/archive/nixpkgs-unstable.tar.gz#hello
nix shell 'git+https://github.com/nixos/nixpkgs?ref=nixpkgs-unstable#hello'
nix shell nixpkgs#hello # nixpkgs is specified in the default registry to be github:ni
```

As for the attribute path, for now, just know that it’s a period-separated list of Nix “attribute names” that selects a flake output according to some simple logic.

Note that in this case, Nix did not have to build anything since it could just fetch GNU Hello and its dependencies from the binary cache. To achieve this, Nix evaluates a *derivation* from the expression, hashes its contents, and queries all the caches it knows to see if someone has the derivation with this hash cached. Nix uses all the dependencies and all the instructions as the input for this hash! If some binary cache has a version ready, it can be *substituted* (downloaded). Otherwise, Nix will build the derivation by first realising (substituting or building) all the dependencies and then executing the build instructions.



You might be wondering where exactly is the executable installed. Well, try `command -v hello` to see that it is located in a subdirectory of `/nix/store`. In fact, all Nix *derivations* have “store paths” (paths located in `/nix/store`) as inputs and outputs.

nix build



If you just want to build something instead of entering a shell with it, try `nix build`:

```
$ nix build nixpkgs#hello
```

This will build Hello (or fetch it from the binary cache if available) and then symlink it to `result` in your current directory. You can then explore `result`, e.g.

```
$ ./result/bin/hello  
Hello, world!
```

nix develop



Despite the use of binary caches, Nix is a sourcecode-first package manager. This means that it has the ability to provide a build environment for its derivations. So, you can use Nix to manage your build environments for you! To enter a shell with all runtime and buildtime dependencies of GNU Hello, use:

```
$ nix develop nixpkgs#hello
```

Inside that shell, you can call `unpackPhase` to place GNU Hello sources in the current directory, then `configurePhase` to run `configure` script with correct arguments and finally `buildPhase` to build.

nix profile



Nix implements stateful “profiles” to allow users to “permanently” install stuff.

For example:

```
nix profile install nixpkgs#hello  
nix profile list  
nix profile update hello  
nix profile remove hello
```



If you’re already familiar with Nix, this is a replacement for `nix-env`.

nix flake



`nix flake` set of subcommands is used to observe and manipulate flakes themselves rather than their

outputs.



nix flake show



This command takes a flake URI and prints all the outputs of the flake as a nice tree structure, mapping attribute paths to the types of values.

For example:

```
$ nix flake show github:nixos/nixpkgs
github:nixos/nixpkgs/d1183f3dc44b9ee5134fcbcd45555c48aa678e93
├── checks
│   └── x86_64-linux
│       └── tarball: derivation 'nixpkgs-tarball-21.05pre20210407.d1183f3'
├── htmlDocs: unknown
└── legacyPackages
    ├── aarch64-linux: omitted (use '--legacy' to show)
    ├── armv6l-linux: omitted (use '--legacy' to show)
    ├── armv7l-linux: omitted (use '--legacy' to show)
    ├── i686-linux: omitted (use '--legacy' to show)
    ├── x86_64-darwin: omitted (use '--legacy' to show)
    └── x86_64-linux: omitted (use '--legacy' to show)
├── lib: unknown
└── nixosModules
    └── notDetected: NixOS module
```

nix flake clone

`nix flake clone` will clone the flake source to a local directory, similar to `git clone`.

Let's clone some simple flake and use some other `nix flake` subcommands on it:

```
nix flake clone git+https://github.com/balsoft/hello-flake/ -f hello-flake
cd hello-flake
```

**nix flake lock (previously nix flake update)**

Every time you call a Nix command on some flake in a local directory, Nix will make sure that the contents of `flake.lock` satisfy the `inputs` in `flake.nix`. If you want to do just that, without actually building (or even evaluating) any outputs, use `nix flake lock`.

There are also some arguments for flake input manipulation that can be passed to most Nix commands:

- `--override-input` takes an input name that you have specified in `inputs` of `flake.nix` and a like URI to provide as this input; `--update-input` will take an input name and update it with the latest version satisfying the flake URI from `flake.nix`.



Writing your own



Now that you know how to interact with flakes, it's time to write one.

Nix language refresher



The widely used data type in Nix is an attribute set: a data type for storing key-value pairs. It is similar to a JSON object or a hashmap in many languages. Its syntax is confusingly similar to a list of statements in C-like languages:

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

The set above is equivalent to this JSON object:

```
{  
  "hello": "world",  
  "foo": "bar"  
}
```

`hello` and `foo` are commonly referred to as “attributes” or “attribute names”; `"world"` and `"bar"` are “attribute values”.

To get an attribute value from an attribute set, use `..`. For example:

```
let  
  my_attrset = { foo = "bar"; };  
in my_attrset.foo
```



(`let ... in` is a way to create bindings; the syntax inside it is identical to that of an attribute set)

You can also abbreviate your attribute set by setting specific attributes with `.` instead of defining the entire set:



```
.bar = "baz";
```



Is equivalent to

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

Other types include strings (`"foo"`), numbers (1, 3.1415), heterogenous lists (`[1 2 "foo"]`) and – quite importantly – functions (`x: x + 1`).

Functions support pattern matching on attribute sets. For example, this function:

```
{ a, b }: a + b
```

When called with `{ a = 10; b = 20; }` will return 30.

Function application is done in ML style:

```
let  
  f = { a, b }: a + b;  
in f { a = 10; b = 20; }
```

The function itself comes first. Then there is a whitespace-separated list of arguments.

If you want to have a function of multiple arguments, use currying:

```
let  
  f = a: b: a + b;  
in f 10 20
```



In this example, `f 10` evaluates to `b: 10 + b`, and then `f 10 20` evaluates to `30`.

If you want to learn more about Nix, check out the [corresponding manual section](#) and [Nix Pills](#).

Basic flake structure



The language description you got above is far from complete or formal, but it should help you understand

and, more importantly, write some simple Nix expressions and, even more importantly, a flake.



A flake is a directory that contains a `flake.nix` file. That file must contain an attribute set with one required attribute – `outputs` – and optionally `description` and `inputs`.

`outputs` is a function that takes an attribute set of inputs (there's always at least one input – `self` – which refers to the flake that Nix is currently evaluating; this is possible due to laziness). So, the most trivial flake possible is this:

```
{  
  outputs = { self }: { };  
}
```

This is a flake with no external inputs and no outputs. Not very useful, huh?

Well, we can add an arbitrary output to it and evaluate it with `nix eval` to see that it works:

```
{  
  outputs = { self }: {  
    foo = "bar";  
  };  
}  
  
$ nix eval .#foo  
"bar"
```

Still not very useful, though.

Let's make a flake that does something useful! For that, we will most likely need some inputs:

```
{  
  inputs = {  
    nixpkgs.url = "github:nixos/nixpkgs";  
  };  
  
  outputs = { self, nixpkgs }: { };  
}
```



Still not very useful; we don't have any outputs! However, now there is an external `nixpkgs` input.

While the attribute set that `outputs` returns may contain arbitrary attributes, some standard outputs are un-

derstood by various `nix` utilities. For example, there is a `packages` output that contains packages. It



ell with the commands described in [Getting a feel for flakes](#). Let's add it!



```
{  
  inputs = {  
    nixpkgs.url = "github:nixos/nixpkgs";  
  };  
  
  outputs = { self, nixpkgs }: {  
    packages.x86_64-linux.hello = /* something here */;  
  };  
}
```

First of all, let's understand why we need `x86_64-linux` here. Flakes promise us *hermetic evaluation*, which means that the outputs of a flake should be the same regardless of the evaluator's environment. One particular property of the evaluating environment that's very relevant in a build system is the *platform* (a combination of architecture and OS). Because of this, all flake outputs that have anything to do with packages must specify the platform explicitly in some way. The standard way is to make the output be an attribute set with names being platforms and values being whatever the output semantically represents, but built specifically for that platform. In the case of `packages`, each per-platform value is an attribute set of packages.

You might now think: well then, how can we just write `nix build nixpkgs#hello` and get a package without explicitly specifying the platform? Well, that's because Nix actually does that for you behind the scenes. For `nix shell`, `nix build`, `nix profile`, and `nix develop` (among some other commands), Nix tries to figure out which output you want by trying multiple in a specific order. Let's say you do `nix build nixpkgs#hello` on an x86_64 machine running Linux. Then Nix will try:

- `hello`
- `packages.x86_64-linux.hello`
- `legacyPackages.x86_64-linux.hello`

We're already familiar with `packages` output; `legacyPackages` is designed specifically for `nixpkgs`. The `nixpkgs` repository is a lot older than flakes, so it is impossible to fit its arbitrary attribute format into neat `packages`. `legacyPackages` was devised to accomodate the legacy mess. In particular, `legacyPackages` allows per-platform packagesets to be arbitrary attribute sets rather than structured packages.



So, let's reexport `hello` from `nixpkgs` in our own flake:

```
{
```

```
inputs = {
  nixpkgs.url = "github:nixos/nixpkgs";
};

outputs = { self, nixpkgs }: {
  packages.x86_64-linux.hello = nixpkgs.legacyPackages.x86_64-linux.hello;
};

}
```



Now we can build the reexported package:

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

Or run it:

```
$ nix run .#hello
Hello, world!
```

(By default, `nix run` will execute the binary with the same name as the attribute name of the package.)

Hooray! Now we have a flake that outputs a package we can use or build.

Another thing we can add is a “development” shell containing some utilities that might be useful when working on our flake. In this example, maybe we want to have `hello` and `cowsay` in `$PATH` to print the friendly greeting and then make the cow say it. There is a special output for such development shells, called `devShell`. There is also a function for building such shells in `nixpkgs`. To prevent writing the unwieldy `nixpkgs.legacyPackages.x86_64-linux` multiple times, let’s extract it via a `let ... in` binding:

```
{
  inputs = { nixpkgs.url = "github:nixos/nixpkgs"; };

  outputs = { self, nixpkgs }: {
    let pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in {
      packages.x86_64-linux.hello = pkgs.hello;

      devShell.x86_64-linux =
        pkgs.mkShell { buildInputs = [ self.packages.x86_64-linux.hello pkgs.cowsay ];
    };
  }
}
```



Now we can enter the development environment with `nix develop`. If you want to run a shell other than

 that environment, you can use `nix shell -c $SHELL`.



```
$ nix develop -c $SHELL
$ hello | cowsay
-----
< Hello, world! >
-----
 \  ^__^
  \  (oo)\_____
    (__)\       )\/\
        ||----w |
        ||     ||
```

Let's examine our flake using `nix flake show`:

```
$ nix flake show
path:/path/to/flake
├── devShell
│   └── x86_64-linux: development environment 'nix-shell'
└── packages
    └── x86_64-linux
        └── hello: package 'hello-2.10'
```

Now that we've written our slightly useful "hello" flake, time to move to practical applications!

Some tips and tricks



direnv



You can use `direnv` with `nix-direnv` to automatically enter `devShell` when you change directory into the project which is packaged in a flake. The `.envrc` file is really simple in that case:

```
use flake
```



flake-utils



There is a library that helps you extract the boring per-platform attrsets away: `flake-utils`. If we use `flake-utils` in our example flake, we can make it support all the `nixpkgs` platforms with practically no extra code:



```
flake-utils = {  
    nixpkgs.url = "github:nixos/nixpkgs";  
    flake-utils.url = "github:numtide/flake-utils";  
};  
  
outputs = { self, nixpkgs, flake-utils }:  
    flake-utils.lib.eachDefaultSystem (system:  
        let pkgs = nixpkgs.legacyPackages.${system};  
        in {  
            packages.hello = pkgs.hello;  
  
            devShell = pkgs.mkShell { buildInputs = [ pkgs.hello pkgs.cowsay ]; };  
        );  
    }  

```



Note how now there are more platforms in the output of `nix flake show`:

```
path:/path/to/flake  
└── devShell  
    ├── aarch64-linux: development environment 'nix-shell'  
    ├── i686-linux: development environment 'nix-shell'  
    ├── x86_64-darwin: development environment 'nix-shell'  
    └── x86_64-linux: development environment 'nix-shell'  
└── packages  
    ├── aarch64-linux  
    │   └── hello: package 'hello-2.10'  
    ├── i686-linux  
    │   └── hello: package 'hello-2.10'  
    ├── x86_64-darwin  
    │   └── hello: package 'hello-2.10'  
    └── x86_64-linux  
        └── hello: package 'hello-2.10'
```

All the platform attributes are inserted automatically by `flake-utils`.



Packaging existing applications



The most obvious use-case for flakes is packaging up existing applications to get the benefits of the Nix ecosystem. To facilitate this, I have written a library of templates that you can use with `nix flake init -t`. This section will mostly be about tweaking the templates to build your application correctly.

Haskell (cabal)



Haskell has a prevalent presence in the Nix ecosystem. Nixpkgs contains a complete mirror of Hackage, and there are multiple tools that facilitate building Haskell applications using Nix.

The most popular way of building Haskell applications with Nix is `cabal2nix`. It extracts dependency information from your project's cabal file and uses the nixpkgs `haskellPackages` collection to resolve those dependencies.

To add it to your existing Haskell application, do:

```
nix flake init -t github:serokell/templates#haskell-cabal2nix
```

It will create a `flake.nix` similar to this (note the `throw <...>` replacement for a name; write the name of your project there).

```
{  
  description = "My haskell application";  
  
  inputs = {  
    nixpkgs.url = "github:NixOS/nixpkgs";  
    flake-utils.url = "github:numtide/flake-utils";  
  };  
  
  outputs = { self, nixpkgs, flake-utils }:  
    flake-utils.lib.eachDefaultSystem (system:  
      let  
        pkgs = nixpkgs.legacyPackages.${system};  
  
        haskellPackages = pkgs.haskellPackages;  
  
        jailbreakUnbreak = pkg:  
          pkgs.haskell.lib.doJailbreak (pkg.overrideAttrs (_: { meta = { }; }));  
  
          packageName = throw "put your package name here!";  
          in {  
            packages.${packageName} = # (ref:haskell-package-def)  
            haskellPackages.callCabal2nix packageName self rec {  
              # Dependency overrides go here  
            };  
          };  
  
          defaultPackage = self.packages.${system}.${packageName};  
        );  
      );  
    );  
  );  
};
```





```
devShell = pkgs.mkShell {  
    buildInputs = with haskellPackages; [  
        haskell-language-server  
        ghcid  
        cabal-install  
    ];  
    inputsFrom = builtins.attrValues self.packages.${system};  
};  
});  
}  
}
```



Using the flake



You can build the `defaultPackage` using `nix build`, and do all the other usual things: `nix shell`, `nix develop`, etc.

Potential modifications and troubleshooting



Unless you need some of the development tools provided in the `devShell` (such as the language server or `ghcid`), just remove them.

If you need a different version of GHC, choose it in `haskellPackages` definition, for example, to use GHC 9.0.1, do:

```
haskellPackages = pkgs.haskell.packages.ghc901;
```

In case some dependency fails to build, you need to override it in the package definition (the last argument of `callCabal2nix`, to be precise). For example, if your project depends on `gi-gtk-declarative` and it is broken in the current version of nixpkgs because some version constraints don't match, do (substituting the failing package name for `gi-gtk-declarative`):

```
# <...>  
packages.${packageName} =  
  haskellPackages.callCabal2nix packageName self rec {  
    gi-gtk-declarative =  
      jailbreakUnbreak haskellPackages.gi-gtk-declarative;  
  };  
# <...>
```



Note: `jailbreakUnbreak` only removes cabal version constraints. If there is an actual breaking change in some dependency, the build will still fail. In that case, there's little you can do but try to fix the actual issue

with the package, which requires some understanding of how `overrideAttrs` works, and that's out of scope of this tutorial. Don't be afraid to read the `nixpkgs` manual or ask for help in any of the Nix community channels you can find!



If some dependency fails to build because of failing tests (which may be the case when the tests require network access or a local resource unavailable from the Nix build), you can disable them like this:

```
# <...>
packages.${packageName} =
  HaskellPackages.callCabal2nix packageName self rec {
  gi-gtk-declarative =
    pkgs.haskell.lib.dontCheck HaskellPackages.gi-gtk-declarative;
};
# <...>
```

Other build options



Some other options for building Haskell applications and libraries with Nix:

- `haskell.nix`

Rust (Cargo)



Rust is also quite popular in the Nix community, although due to the relatively young age the integration isn't as good at this time. There are multiple competing ways to build crates. I prefer the way `crate2nix` works: it's very similar in operation to `cabal2nix`, but it's a bit more complicated due to some Cargo peculiarities.

To get a template that builds a single Cargo crate via `crate2nix`, run `nix flake init -t github:serokell/templates#rust-crate2nix`. The `flake.nix` is going to be similar to this file:

```
{
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs";
    crate2nix = {
      url = "github:kolloch/crate2nix";
      flake = false;
    };
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, crate2nix, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
```





```
let
  pkgs = nixpkgs.legacyPackages.${system};
  crateName = throw "Put your crate name here";

  inherit (import "${crate2nix}/tools.nix" { inherit pkgs; })
    generatedCargoNix;

  project = pkgs.callPackage (generatedCargoNix {
    name = crateName;
    src = ./.;
  }) {
    defaultCrateOverrides = pkgs.defaultCrateOverrides // {
      # Crate dependency overrides go here
    };
  };

in {
  packages.${crateName} = project.rootCrate.build;

  defaultPackage = self.packages.${system}.${crateName};

  devShell = pkgs.mkShell {
    inputsFrom = builtins.attrValues self.packages.${system};
    buildInputs = [ pkgs.cargo pkgs.rust-analyzer pkgs.clippy ];
  };
};

}
```



Troubleshooting and improvements >

Suppose your package requires some “native” (non-Rust) libraries, either itself or transitively via a dependency. In that case, you have to specify them manually to `defaultCrateOverrides`, unless they are already specified in `nixpkgs`’ `defaultCrateOverrides`. For example, if your application called `my-music-player` requires `libpulseaudio`,



```
# <...>
defaultCrateOverrides = pkgs.defaultCrateOverrides // {
  my-music-player = _: {
    buildInputs = [ pkgs.libpulseaudio ];
  };
}
#<...>
```

Alternatives

manually using `buildRustCrate` from `nixpkgs`

- [naersk](#)



Python (poetry)



Python ecosystem is somewhat supported in `nixpkgs` via `python3Packages`, but the support is not as complete as for Haskell. However, if your project is packaged with poetry, you're in luck as there is `poetry2nix` which allows you to build the application easily.

To initialize the template, `nix flake init -t github:serokell/templates#python-poetry2nix`. It looks something like this:

```
{  
    description = "My Python application";  
  
    inputs = {  
        nixpkgs.url = "github:NixOS/nixpkgs";  
        flake-utils.url = "github:numtide/flake-utils";  
    };  
  
    outputs = { self, nixpkgs, flake-utils }:  
        flake-utils.lib.eachDefaultSystem (system:  
            let  
                pkgs = nixpkgs.legacyPackages.${system};  
  
                customOverrides = self: super: {  
                    # Overrides go here  
                };  
  
                app = pkgs.poetry2nix.mkPoetryApplication {  
                    projectDir = ./.;  
                    overrides =  
                        [ pkgs.poetry2nix.defaultPoetryOverrides customOverrides ];  
                };  
  
                packageName = throw "put your package name here";  
                in {  
                    packages.${packageName} = app;  
  
                    defaultPackage = self.packages.${system}.${packageName};  
                };  
            };  
        );  
    };  
};
```





```
devShell = pkgs.mkShell {  
    buildInputs = with pkgs; [ poetry ];  
    inputsFrom = builtins.attrValues self.packages.${system};  
};  
});  
}
```



Troubleshooting



In case some dependency fails to build, you need to fix it by overriding it in `customOverrides`.

Up next



In an upcoming article, I'll explain how to use Flakes to build and run containers and set up infrastructure deployments. To make sure you hear about it, you can follow Serokell on [Twitter](#). Stay tuned!

TAGGED: nix

Share:

 191 upvotes

Get new articles via email

No spam – you'll only receive stuff we'd like to read ourselves.

Enter your e-mail

Accept [Privacy notice](#)

Subscribe





FP gone wild.

New shirts in Serokell Shop

[Serokell Shop →](#)



More from Serokell



CI/CD/CT WITH NIX



CI/CD/CT With Nix

Nix is a powerful, purely functional package manager that offers reliable and reproducible build and deployment processes, eliminating many of the challenges traditionally associated with software lifecycle management. In this article, we explore the integration of Nix with CI, CD, and CT pipelines and explain how it can streamline DevOps and MLOps processes.

October 9th, 2023 | 8 min read

HASKELL IN PRODUCTION CHANNLABLE



Haskell in Production: Channable

In this edition of our Haskell in Production series, we interview Fabian Thorand from Channable. Read the article to learn where Channable uses Haskell and why do they like it.

June 27th, 2022 | 9 min read



Biotech

Blockchain

Fintech



Elixir Development

Nix Development

Rust Development

How We Work

Managed IT Services

Smart Contract Audit



Haskell Development

Python Development

TypeScript Development

Privacy Policy

(+372) 699-1531

hi@serokell.io

Pille tn 7/5-13, Kesklinna linnaosa, Tallinn, Harju maakond, 10135, Estonia

Serokell: Rate 5.0 based on 9 Google Business reviews

© 2015–2024 Serokell

