Flakes aren't real and cannot hurt you: a guide to using Nix flakes the non-flake way

jade.fyi/blog/flakes-arent-real/

January 02, 2024 26 minute read

Inflammatory title out of the way, let's go.

I think that Nix flakes have some considerable benefits, such as:

- Convenient pinning of evaluation-time dependencies
- Eliminating pointless rebuilds of code by only including tracked files in builds
- Making Nix code, on average, much more reproducible by pervasive pinning
- Allegedly caching evaluation
- Possibly making Nix easier to learn by reducing the amount of poking at strange attribute sets and general `NIX_PATH` brokenness

However, at the same time, there are a few things that one might be led to think about flakes that are not the most effective way of doing things. I personally use flakes relatively extensively in my own work, but there are several ways I use them that are not standard, with reason.

Flakes are *optional*, and as much as some people whose salary depends on it might say otherwise, they are not the (only) future of Nix: they are simply a special entry point for Nix code with a built in pinning system, nothing more, nothing less.

Nix continues to gather a reputation for bad documentation, in part because the official documentation for nixpkgs and NixOS is *de facto* not allowed to talk about flakes, as a policy. This situation is certainly partially due to a divide between Nix developers and nixpkgs developers, which are groups with surprisingly little overlap.

Flakes also are a symptom or cause of much intra-community strife between "pro-flakes" and "anti-flakes" factions, but this situation is at some level a sign of broken consensus processes and various actors trying to sidestep them, an assumption by many people that the docs are "outdated" for not using flakes, and the bizarre proliferation of flakes everywhere in blog posts or tutorials leading to a belief that they are required for everything.

This post is about how to architect Nix projects in general, with a special eye on how to do so with flakes while avoiding their limitations. It tries to dispel misconceptions that can develop in such a monoculture.

"Flakes are the composition primitive in Nix"

The Nix language, functions, and nixpkgs utilities are an effective composition primitive, and are much better suited for putting parts of a project together, especially if it is in a monorepo.

The most flexible way of building large systems with Nix is to merely use flakes as an entry point, and develop the rest using "old" tools. This is for multitudinous reasons:

- Flakes couple version control integration, dependency management and lockfile management. In medium sized projects, even at the scale of my dotfiles, locking dependencies of subprojects is often highly undesirable.

  They're not ideal for either working in the large or in the small: in the small, there is too much overhead in writing a separate `flake.nix` for some tiny utility, and in the large, for example, in nixpkgs, if flakes were actually used for dependency management, `flake.nix` would be 100,000 lines of `inputs` long.

- In terms of making flexible builds, flakes don't support configuration except through hilarious abuses of `--override-input`. This means that all build configuration variants have to be anticipated ahead of time, or that traditional nixpkgs/Nix language primitives need to be used instead.

- Flakes as a composition primitive is completely incompatible with cross compilation. Due to the lack of configuration support, `packages.${system}` cannot be used for cross compilation: there is nowhere to specify the architecture to build with.

Because of all of this, even in a flakes world, to compose software in the large *and* in the small reusably and efficiently, the other composition primitives provided by Nix and nixpkgs remain the best choices to assemble software. A flake can then be relegated to merely an entry point and a way of acquiring dependencies that are required for evaluation (build-time dependencies should use `pkgs.fetchurl`, `fetchFromGitHub`, etc).

For example, to expose multiple configurations of a program, one might write it the traditional way, using a lambda accepting some configuration parameters, then call that lambda multiple times to expose multiple output attributes inside the flake itself. This separates the capability to configure the software from the actual defined configurations of the software, and avoids letting the configuration non-system of flakes define how the internals of the build definition work.

One of the largest simultaneous advantages and disadvantages of the Nix language is that it is a Turing complete language, which causes pain to static analysis, but is also one of its largest assets: you can program it. This can be seen as a problem, but it also is awesome: you can programmatically patch packages, define configuration dynamically, read files of arbitrary formats and more.

Nix is a functional programming language, which means that its fundamental composition primitive is the function. Even "fancy" objects like NixOS modules or overlays are just functions that can be moved into separate files, imported, or created through partial application of other functions (although, since `imports` in modules are deduplicated by file name, NixOS modules generally should be imported by path instead of generated by functions).

See the next section for concrete ways of composing software together.

"Flakes are where you put your Nix code"

Flakes are merely a fancy schema for making a standardized entry point into Nix code. Most of the Nix code in a project of any significant size should not be in `flake.nix`, for several reasons.

The most trivial reason to put as little code as possible in `flake.nix` is maintainability: there is as much rightward drift in `flake.nix` as in recent German and Dutch elections (concerningly much!), so from just that perspective, it's useful to move things out of it.

Let's talk about some standard patterns that have existed before flakes did, which still are relevant in a flakes world.

`package.nix`

I am using `package.nix` to refer to the standard way for writing packages in nixpkgs style, which are invoked with `callPackage`. This is as opposed to writing something directly in `flake.nix` using `pkgs`.

A `package.nix` file looks something like so:

```
{ # receives(*) pkgs.hello and pkgs.stdenv

  hello, stdenv,

  # can be overridden with `yourPackage.override { enableSomething = true; }`

  enableSomething ? false


}:


stdenv.mkDerivation (finalAttrs: {

  # optional finalAttrs to refer to the set below; preferred over using `rec` attr
sets

  # ...


})
```

Package definitions should be written with `callPackage` if possible, rather than inline in `flake.nix`, since using `package.nix` makes them into small, composable, configurable, and portable units of software. Also, by using `callPackage` and writing in nixpkgs style, it becomes a lot easier to move packages between projects, and indeed to upstream them to nixpkgs, since they look and work a familiar way.

Cross compilation

A lesser-known fact is that `callPackage` is load-bearing for cross compilation. If you write `pkgs.foo` in `nativeBuildInputs`, such a Nix expression will break under cross compilation, but `foo` as an argument from `callPackage` will not. This is because `callPackage` will magically resolve `foo` appearing inside `nativeBuildInputs` to mean `pkgs.buildPackages.foo`; that is, a package built for the build computer.

`callPackage` evaluates a Nix file multiple times with different arguments and splices the results together such that `buildInputs` magically receives target packages, and `nativeBuildInputs` receives build packages, even if the same package name appears in both. Magic ✨

That is, in the following intentionally-flawed-for-other-reasons `flake.nix`:

```
{...}: {
  outputs = { nixpkgs, ... }:
  let pkgs = nixpkgs.legacyPackages.x86_64-linux;
  in {
    packages.x86_64-linux.x = pkgs.callPackage ./package.nix { };
  };

}
```

then `package.nix`:

```
{ stdenv, hello, openssl }:

stdenv.mkDerivation {
  # ...
  # things used in the build go in here
  nativeBuildInputs = [ hello ];
  # libraries used by the resulting program go in here
  buildInputs = [ openssl ];

}
```

Incidentally, notice anything there? Yeah, it's flakes completely not supporting cross compilation. See the next point. :D

It's possible to use the `pkgs.buildPackages` attribute to pull things into `nativeBuildInputs`, and `pkgs` for `buildInputs` but it is not conventional to do so, and is quite verbose.

See the manual about these callPackage shenanigans for more details. See also: the manual about dependency categories.

Overlays

An overlay is a function overriding nixpkgs which is evaluated until it reaches a fixed point. An overlay takes two arguments, `final` and `prev` (sometimes also called `self` and `super`), and returns an attribute set that is shallowly replaced on top of nixpkgs with `//`.

Overlays are useful as a means for distributing sets of software outside of nixpkgs, and still are useful in that role in a flakes world, since overlays are simple functions that can be evaluated against any version of nixpkgs, and if written with `callPackage`, cross compilation works. One may notice that the `overlays` flake output is not architecture specific, which follows from their definition as functions that take package sets and return modifications to make; this is why they work properly here.

Evaluation to a fixed point means that it is evaluated as many times as necessary until it stops referring to the `final` argument (or overflows the stack). This idea appears in many places, including tables of contents in LaTeX, Typst or other typesetting programs: by generating the table of contents, you may affect the layout of subsequent pages and change their page numbers, but after the first run of that, the layout will probably not change, since the only change is the numbers, so *that* iteration likely converges to the final result.

`final` gives the *final* version of the attribute set, after overlays have been evaluated as far as they will go; your overlay may be run multiple times in evaluating an attribute in `final`, or even cause infinite recursion. `prev` gives the version of nixpkgs prior to the present overlay or any further overlays.

For example, we could write an overlay to override GNU Hello to be a wrapper that makes a reference to an excellent retrocomputing series. Content of `overlay.nix`:

```
final: prev: {

  hello = final.writeShellScriptBin "hello" ''

    ${prev.hello}/bin/hello -g "hellorld" "$@"

  '';

}
```

Then:

```
» nix run --impure --expr '(import <nixpkgs> { overlays = [ (import ./overlay.nix)
]; }).hello'
hellorld
```

Here, the attribute `hello` of our modified `nixpkgs` now is our script that calls the original `hello` to say "hellorld".

It's pretty easy to accidentally cause infinite recursion with overlays if their laziness isn't correct. For example, attribute sets' attribute names are evaluated strictly, with all names in an attribute set evaluated immediately, but the values of attributes are lazily evaluated. There have been attempts to change this but they were canned for performance reasons. Strict attribute names can be a foot-gun, causing confusing infinite recursion in some cases using `mapAttrs` or similar mechanisms on `prev` to generate the set of things to override.

Infinite recursion is not typically a problem if an overlay doesn't actually replace anything or contain self-references, as may be the case for overlays distributing very simple software, and we can take advantage of that as shown in the next section.

The place of overlays in a flakes world

Flakes don't support cross compilation.

I am being a little bit tricky with the wording here. Flakes don't *stop* you from doing cross compilation, but you have to do an end-run around flakes and do it the "old" way.

Because of this design fault in flakes, namely, the lack of support for parameters, the most compatible way of writing packaging in a flake project is to write the package definitions into an overlay first, then expose the packages from the overlay. Consumers that need cross compilation can use the overlay with their own copy of nixpkgs, and consumers that don't care can use it through `packages`.

Keeping in mind "1000 instances of nixpkgs", a reasonable way of writing a flake that *doesn't modify anything in nixpkgs* and just adds stuff is:

```
{
  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
  inputs.flake-utils.url = "github:numtide/flake-utils";
  outputs = { self, nixpkgs, flake-utils, ... }:
    let
      out = system:
        let
          pkgs = nixpkgs.legacyPackages.${system};
          appliedOverlay = self.overlays.default pkgs pkgs;
        in
        {
          packages.myPackage = appliedOverlay.myPackage;
        };
    in
    flake-utils.lib.eachDefaultSystem out // {
      overlays.default = final: prev: {
        myPackage = final.callPackage ./package.nix { };
      };
    };
}
```

Downstream consumers that need to cross compile in spite of flakes can use the overlay, and other consumers can use `packages` as normal.

This uses a cute trick of calling the overlay, which is just a function, with both `final` and `prev` as the final nixpkgs attribute set. This definitely does not work on all overlays, since overlays can make self-references using `final`, and indeed often need to do so, if they contain multiple derivations that depend on each other.

However, with a little bit more work, this can be overcome very cleanly, while also avoiding any possibility of name shadowing problems!

If you're thinking "just use a `rec` attribute set", that's unfortunately clever but flawed: `rec` will receive the version as of the execution of your file, but not any overridden version, which is not the case for `makeScope` and similar tools from nixpkgs.

In order to regain the ability to make self-references without being a real overlay that uses `prev`, consider using makeScope (example from nixpkgs) to create a smaller *scope*, within which self references to other things in the same scope are allowed.

For example, here we create a scope with a dependency between derivations. Content of `test.nix`, which could equivalently be an overlay:

```
let pkgs = import <nixpkgs> { };


in pkgs.callPackage ./scope.nix { makeScope = pkgs.lib.makeScope; }
```

and `scope.nix`:

```
{ makeScope, newScope, writeShellScriptBin }: makeScope newScope (self: {

  meow = writeShellScriptBin "meow" ''

    echo meow

  '';

  meow2 = writeShellScriptBin "meow2" ''

    echo "meow is at ${self.meow}"

  '';

})
```

Which gives the following result:

```
 » nix run --impure --expr '(import ./test.nix).meow'
meow
 » nix run --impure --expr '(import ./test.nix).meow2'
meow is at /nix/store/aj0fhn8is6w8q85h0ramnqz2di92plwc-meow
 » nix eval --impure --expr 'builtins.attrNames (import ./test.nix)'
[ "callPackage" "meow" "meow2" "newScope" "override" "overrideDerivation"
"overrideScope" "overrideScope'" "packages" ]
```

If you do have to use a real overlay that needs to replace things, import nixpkgs again from your flake with the overlay as an argument. It's fine. It's just a second of gratuitous evaluation time:

```
let pkgs = import nixpkgs { inherit system; overlays = [ self.overlays.default ];
};

in # ....
```

NixOS modules

NixOS modules are, like overlays and `package.nix`, fundamentally just functions which are invoked in a fancy way, and are not a flakes construct.

As used in flakes with the `nixosModules.*` output, they are *architecture independent* since they are just functions, and if defining a module for software that is built by the same flake, one would generally want to use an overlay in `nixpkgs.overlays` or the trick above, invoking the overlay with `pkgs` twice, to actually bring it in (again, to remain cross compilation compatible).

To keep with the theme of putting things outside of `flake.nix` to enable reusability, the code for the module can be placed in a separate file that is imported. Then, `flake.nix` is used to import that module and inject dependencies from its environment.

Injecting dependencies

There are a couple of ways to inject dependencies into NixOS modules from a flake, one of which is mildly uglier. Injecting values from `flake.nix` into NixOS is required for a couple of reasons, most notably, to use flakes-managed dependencies inside NixOS configurations. It is also necessary to properly configure `NIX_PATH` so `<nixpkgs>` resolves in a flake configuration, since you need the actual inputs from `flake.nix` to get a proper reference to nixpkgs suitable to create a dependency on the actual flake input.

The simplest (and most reasonable, in my view) way to inject dependencies from a flake is to write an inline module that has them in its lexical closure inside of `flake.nix`. If you want to be fancy, you could even make an option to store the injected dependencies:

```
let depInject = { pkgs, lib, ... }: {

  options.dep-inject = lib.mkOption {

    type = with lib.types; attrsOf unspecified;

    default = { };

  };

  config.dep-inject = {

    # inputs comes from the outer environment of flake.nix

    flake-inputs = inputs;

  };


};


in {

  nixosModules.default = { pkgs, lib, ... }: {

    imports = [ depInject ];

  };


}
```

The uglier and perhaps more well-known way to inject dependencies into NixOS modules from flakes is `specialArgs`. This is uglier, since it gets dumped into the arguments for every module, which is unlike how every other bit of data flow works in NixOS, and it also doesn't work outside of the flake that's actually invoking `nixpkgs.lib.nixosSystem`. The latter is the much more sinister part, and the reason I would strongly recommend inline modules with closures instead of `specialArgs`: they break flake composition.

To use `specialArgs`, an attribute set is passed into `nixpkgs.lib.nixosSystem`, which then land in the arguments of NixOS modules:

```
# ...


nixosConfigurations.something = nixpkgs.lib.nixosSystem {

  system = "x86_64-linux";

  specialArgs = {

    myPkgs = nixpkgs;

  };

  modules = {

    { pkgs, lib, myPkgs }: {

      # do something with myPkgs

    }

  };


}
```

Example

This could be equivalently done with the overlay invocation trick above on pkgs.

For example, this defines a very practical NixOS module that meows at the user on the console on boot:

```
{

  inputs.nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";

  outputs = { self, nixpkgs, ... }: {

    overlays.default = final: prev: {

      meow = final.writeShellScriptBin "meow" ''


        echo meow


      '';

    };



    nixosModules.default = { pkgs, config, lib, ... }: {

      imports = [ ./nixos-module.nix ];

      # inject dependencies from flake.nix, and don't do anything else

      config = lib.mkIf config.services.meow.enable {

        nixpkgs.overlays = [ self.overlays.default ];

        services.meow.package = lib.mkDefault pkgs.meow;

      };

    };



  };
}
```

and `nixos-module.nix` containing the actual code:

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


let cfg = config.services.meow; in {

  options = {

    services.meow = {

      enable = lib.mkEnableOption "meow";

      package = lib.mkOption {

        description = "meow package to use";

        type = lib.types.package;

      };

    };

  };

  config = lib.mkIf cfg.enable {

    systemd.services.meow = {

      description = "meow at the user on the console";

      serviceConfig = {

        Type = "oneshot";

        ExecStart = "${cfg.package}/bin/meow";

        StandardOutput = "journal+console";

      };
      wantedBy = [ "multi-user.target" ];

    };

  };


}
```

▶ How I tested the above

"Flakes are the future of Nix, and the only CLI"

Many words have been spilled on the new CLI and its design, mostly focusing on flakes. However, this is not the only mode of the new CLI: wherever it makes sense, it actually fully supports non-flake usage.

To get more exact equivalence with the old CLI, `-L` (`--print-build-logs`) and `--print-out-path` are useful. Equally, the *old* CLI can have its output improved to that of the new CLI by passing `--log-format bar-with-logs`. I would be remiss not to mention nix-output-monitor as a much nicer way of watching Nix builds, as well.

Here is a table of the equivalences:

Old CLI	Equivalent
`nix-build -A hello`	`nix build -f . hello`
`nix-shell -A blah`	`nix develop -f . blah`
-	`nix run -f . hello`
`nix-build -E '(import <nixpkgs> { config.allowUnfree = true; }).blah'`	`nix build --impure --expr '(import <nixpkgs> { config.allowUnfree = true; }).blah'`
`nix-instantiate --eval --strict -E 'blah'`	`nix eval --impure --expr 'blah'`

"Flakes are how to manage external dependencies"

Flakes are one way of managing external dependencies but they have many flaws in that role.

One flaw is that all the dependencies need to be listed in one file, and there is no way of scoping them into groups.

For fetching things that are needed to build but not needed to evaluate, flake inputs suffer from a poorly documented limitation of builtin fetchers, which is the reason they are banned in nixpkgs (in addition to `restrict-eval` making them not work), is that they block further evaluation while fetching. The alternative to this is to use a fixed-output derivation that performs the fetching at build time, such as is done with `pkgs.fetchFromGitHub`, `pkgs.fetchurl` and so on.

The blocking is not necessarily the biggest problem if the dependencies are Nix code required to evaluate the build, since you cannot avoid blocking with those, but it can be troublesome when the dependencies are not required to evaluate, since it slows down and serializes evaluation, downloading just one thing at a time. If the dependencies are required for evaluation, there is not much way to make this better, but for instance, for builds requiring many build-time inputs such as a pile of tree-sitter grammars, Haskell package sources, or such, switching to fixed-output derivation fetchers will save a lot of time.

If not flakes then what?

There's a perfectly reasonable argument to be made for just treating dependencies the same way as nixpkgs and directly calling `pkgs.fetchurl` and such inside Nix source. This works fine, is conventional, and avoids the evaluation-time-build-dependency ("import from derivation" (IFD)) problems.

It's nice to have tools to automatically update these and grab the appropriate hash, though.

There are several tools that can maintain a lock file with Nix hashes, such as Niv, npins, and gridlock. The first two sadly ship Nix files that use built-in fetchers and thus have the evaluation performance issues, and the latter doesn't ship any Nix code. This is not a knock on these projects: their primary purpose is in pinning Nix code, for which builtin fetchers are the right choice, but it does mean that the code they ship shouldn't be used for build dependencies.

Thus, the solution for build-time dependencies is to ignore any provided Nix code for whichever one you choose to use and write some code to read the tool's JSON file and pull the package URL and hashes out, and call `pkgs.fetchurl` with them. This is quite easy to do and we would recommend it.

Why are there five ways of getting software?

It's possible to get lost in the sheer number of ways of doing things and lose what any of it is for. I think this is actually worsened by flakes, *because* they make examples self-contained, making them more easy to just pick up and run without contemplation. Often a lot of blogs provide examples in flake form, potentially as NixOS modules or other forms which have a lot of fanciness that might be surplus to requirements for the desired application.

If one is new to Nix it gets very easy to think "Nix Nix Nix I shall reduce the world to nothingness, Nix" and convert everything to be tangled with Nix, and it helps to understand the available mechanisms and why one might need one particular one.

The various ways of installing things have different relationships to mutability and "effects". By effects, I mean, mutations to the computing system, which other systems might use post-install scripts for. Nix derivations don't support post-install scripts because "installing" doesn't mean anything. By persistent, I mean that they have some lasting effect on the system besides putting stuff in the Nix store.

This section perhaps deserves its own post, but I will briefly summarize:

`flake.nix` (`default.nix`, `shell.nix`) in project directories

These are *developer* packaging of projects: pinned tool versions, not caring as much about unifying dependencies with the system, etc. To this end, they provide dev shells to work on a project, and are versioned *with* the project. Additionally they may provide packaging to install a tool separately from nixpkgs.

There are a couple of things that make these notable compared to the packaging one might see in nixpkgs:

- In nixpkgs, more build time is likely tolerated, and there is little desire to do incremental compilation. Import from derivation is also banned from nixpkgs. For these reasons, packaging outside of nixpkgs likely uses different frameworks such as `crane`, `callCabal2nix` and other similar tools that reduce the burden of maintaining Nix packaging or speed up rebuilds.
- It's versioned with the software and so more crimes are generally tolerated: one might pin libraries or other such things, and update nixpkgs infrequently.
- They include the tools to *work on* a project, which may be a superset of the tools required to build it, for example in the case of checked-in generated code or other such things.
- They may include things like checks, post-commit hooks and other project infrastructure.

Shells are just made of environment variables and (without building one at least) don't create a single `bin` folder of all the things in the shell, for instance. Also, since they are made of environment variables, they don't have much ability to perform effects such as managing services or on-disk state.

Use this for tools specific to one project, such as compilers and libraries for that project. Depending on taste and circumstances, these may or may not be used for language servers. Generally these are not used for providing tools like `git` or `nix` that are expected to be on the system, unless they are required to actually compile the software.

- Declarative
- Persistent
- Effects

Ephemeral shells (`nix shell`, `nix-shell -p`)

The ephemeral shell is one of the superpowers of Nix since it can appear software from the ether without worrying about getting rid of it later. This essentially has exactly the same power as project-specific `flake.nix` files: you can bring packages into scope or do anything else that can be done there.

I would consider a project shell file to be simply a case of saving a `nix-shell -p` invocation, and the motivation to do so is about the same, just with more Software Engineering Maintainability Juice with pinning and such.

Use this for grabbing tools temporarily for whatever purpose that might have.

- Declarative
- Persistent
- Effects

Note that for Bad Reasons, `nix-shell -p` is not equivalent to `nix shell`: the latter does not provide a compiler or `stdenv` as would be necessary to build software. The technical reason here is that `nix shell` constructs the shell within C++ code in Nix, whereas `nix-shell -p` is more or less `nix develop` on a questionable string templated expression involving `pkgs.mkShell`.

`nix profile`, `nix-env`

`nix profile` and `nix-env` build a directory with `bin`, `share`, and such, symlinking to the actual packages providing the files. Under the hood, these are mostly `pkgs.buildEnv`, which is a script that builds such a structure out of symlinks. They then symlink it somewhere in PATH.

Personally, I don't think that `nix profile` and `nix-env` should ever be used except when convinced to operate in a declarative manner, because they are the exception in the Nix ecosystem as far as being both imperative and persistent, and doing it declaratively avoids various brokenness by fully specifying intent (they have some ugly edge cases in upgrades which are solved by simply writing the Nix code specifying where the packages come from into a file).

Use these for nothing. Or not, I'm not a cop.

- Declarative
- Persistent
- Effects

flakey-profile, `nix-env --install --remove-all --file`

The old Nix profile CLI actually supports declarative package installation, although I wouldn't suggest it because flakey-profile is just plainly more pleasant UX wise and is absolutely trivial in implementation. These do the same thing as `nix profile` and `nix-env` in terms of building a directory with `bin`, `share`, and such, and putting it somewhere in PATH.

Use these for lightweight declarative package management, perhaps on non-NixOS systems (there's nothing stopping you using it on NixOS but NixOS itself is right there).

- Declarative
- Persistent
- Effects

`home-manager`

Those who use `home-manager` generally use it to replace dotfile managers, as well as configuring services and installing user-specific packages. I don't use it because an approach based on symlinks into a git repo avoids adding an unnecessary Nix build and much complexity to the config file iteration cycle.

`home-manager` has essentially the power of NixOS in terms of being able to have effects such as services, activation scripts, etc, while being scoped to one user.

Use this for installing packages on one user, potentially not on a NixOS system, in a declarative manner, as well as configuring user-scoped services. Note that this overlaps with profiles as described above; it's just a heavier weight mechanism built with the same tools.

- Declarative
- Persistent
- Effects

NixOS/`nix-darwin`

NixOS and `nix-darwin` are system-wide configuration management systems built on top of Nix profiles, combined with activation scripts. They allow installing and configuring services, and managing config files in a declarative manner. In terms of both implementation and usage, these do similar things to `home-manager`, but scoped system-wide.

Use this for installing packages system-wide and configuring services.

- Declarative
- Persistent
- Effects

Conclusion

I hope to have successfully covered why flakes aren't everything, and perhaps even why they aren't real. Although nixpkgs isn't always a shining example of fabulous Nix project architecture, it is a large project, and there is a lot to be learned from how they organize things, which arguably was more than was internalized while flakes were designed.

Even assuming that flakes are good at macro-level composition, they often are accompanied by poor use of micro-level composition, which still is best done by using the old primitives.

With better architecture, we can work around the limitations of flakes to create pleasant-to-work-with and extensible Nix code. We can clarify the meaning of all the "just write a flake" blog posts to see the Nix tools within and avoid spurious, unnecessary, dependencies on flakes that make code harder to understand.

- Newer post

  ←
- Older post

  The postmodern build system →