Rapid Introduction to Nix

The goal of this mini-tutorial is to introduce you to Nix the language, including flakes, as quickly as possible while also preparing the motivated learner to dive deeper into the whole Nix ecosystem. At the end of this introduction, you will be able to create a flake.nix that builds a package and provides a developer environment shell.

Purely functional

If you are already experienced in purely functional programming, it is highly recommended to read Nix - taming Unix with functional programming to gain a foundational perspective into Nix being purely functional but in the context of *file system* (as opposed to values stored in memory).

> *[..] we can treat the file system in an operating system like memory in a running program, and equate package management to memory management*

Pre-requisites

- **Install Nix**: Nix can be installed on Linux and macOS. If you are using NixOS, it already comes with Nix pre-installed.
- **Play with Nix**: Before writing Nix expressions, it is useful to get a feel for working with the `nix` command. See First steps with Nix

Attrset

To learn more

- Official manual
- nix.dev on attrsets

The Nix programming language provides a lot of general constructs. But at its most basic use, it makes heavy use of *nested hash maps* otherwise called an "attrset". They are equivalent to `Map Text a` in Haskell. The following is a simple example of an attrset:

```
{
  foo = {
    bar = 1;
  };
}
```

We have an outer attrset with a single key `foo`, whose value is another attrset with a single key `bar` and a value of `1`.

repl

Nix expressions can be readily evaluated in the Nix repl. To start the repl, run `nix repl`.

```
$ nix repl
Welcome to Nix 2.12.0. Type :? for help.

nix-repl>
```

You can then evaluate expressions:

```
nix-repl> 2+3
5

nix-repl> x = { foo = { bar = 1; }; }
```

```
nix-repl> x
{ foo = { ... }; }

nix-repl> x.foo
{ bar = 1; }

nix-repl> x.foo.bar
1

nix-repl>
```

Flakes

To learn more

- Serokell Blog: Basic flake structure

A Nix flake is defined in the `flake.nix` file, which denotes an attrset containing two keys `inputs` and `outputs`. *Outputs* can reference *inputs*. Thus, changing an *input* can change the *outputs*. The following is a simple example of a flake:

```
{
  inputs = { };

  outputs = inputs: {
    foo = 42;
  };
}
```

This flake has zero `inputs`. `outputs` is a function that takes the (realised) inputs as an argument and returns the final output attrset. This output attrset, in our example, has a single key `foo` with a value of `42`.

We can use the `nix flake show` command to see the output structure of a flake:

```
$ nix flake show
path:/Users/srid/code/nixplay?lastModified=1675373998&narHash=sha256-ifNiFGU1\
└───foo: unknown
$
```

We can use `nix eval` to evaluate any output. For example,

```
$ nix eval .#foo
42
```

Graph

A flake can refer to other flakes in its inputs. Phrased differently, a flake's outputs can be used as inputs in other flakes. The most common example is the nixpkgs flake which gets used as an input in most flakes. Intuitively, you may visualize a flake to be a node in a larger graph, with inputs being the incoming arrows and outputs being the outgoing arrows.

Inputs

To learn more

- URL-like syntax used by the `url` attribute

Let's do something more interesting with our `flake.nix` by adding the nixpkgs input:

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

  outputs = inputs: {
    # Note: If you are macOS, substitute `x86_64-linux` with `aarch64-darwin`
    foo = inputs.nixpkgs.legacyPackages.x86_64-linux.cowsay;
  };
}
```

About `nixpkgs-unstable`

The `nixpkgs-unstable` branch is frequently updated, hence its name, but this doesn't imply instability or unsuitability for use.

The nixpkgs flake has an output called `legacyPackages`, which is indexed by the platform (called "system" in Nix-speak), further containing all the packages for that system. We assign that package to our flake output key `foo`.

You can use `nix repl` to explore the outputs of any flake, using TAB completion:

```
$ nix repl --extra-experimental-features 'flakes repl-flake' github:nixos
Welcome to Nix 2.12.0. Type :? for help.

Loading installable 'github:nixos/nixpkgs/nixpkgs-unstable#'...
Added 5 variables.
nix-repl> legacyPackages.aarch64-darwin.cowsay
«derivation /nix/store/0s2vdpkpdiljmh8y06xgdw5vg2cqfs0m-cowsay-3.7.0.drv»

nix-repl>
```

Predefined outputs

Nix commands treat certain outputs as special. These are:

Output	Nix command	Description
`packages`	`nix build`	Derivation output
`devShells`	`nix develop`	Development shells
`apps`	`nix run`	Runnable applications
`checks`	`nix flake check`	Tests and checks

All of these predefined outputs are further indexed by the “system” value.

Packages

To learn more

- `pkgs.stdenv.mkDerivation` can be used to build a custom package from scratch

`packages` is the most often used output. Let us extend our previous `flake.nix` to use it:

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

```
  outputs = inputs: {
    foo = 42;
    packages.x86_64-linux = {
      cowsay = inputs.nixpkgs.legacyPackages.x86_64-linux.cowsay;
    };
  };
}
```

Here, we are producing an output named `packages` that is an attrset of systems (currently, only `x86_64-linux`) to attrsets of packages. We are definining exactly one package, `cowsay`, for the `x86_64-linux` system.

```
$ nix flake show
path:/Users/srid/code/nixplay?lastModified=1675374260&narHash=sha256-FRven09f
├───foo: unknown
└───packages
    └───aarch64-darwin
        └───cowsay: package 'cowsay-3.7.0'
```

Notice that `nix flake show` recognizes the *type* of `packages`. With `foo`, it couldn't (hence type is `unknown`) but with `packages`, it can (hence type is "package").

The `packages` output is recognized by `nix build`.

```
$ nix build .#cowsay
```

The `nix build` command takes as argument a value of the form `<flake-url>#<package-name>`. By default, `.` (which is a flake URL) refers to the current flake. Thus, `nix build .#cowsay` will build the `cowsay` package from the current flake under the current system. `nix build` produces a `./result` symlink that points to the Nix store path containing the package:

```
$ ./result/bin/cowsay hello
 _______
< hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

If you run `nix build` without arguments, it will default to `.#default`.

Apps

A flake app is similar to a flake package except it always refers to a runnable program. You can expose the cowsay executable from the cowsay package as the default flake app:

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

  outputs = inputs: {
    apps.x86_64-linux = {
      default = {
        type= "app";
        program = "${inputs.nixpkgs.legacyPackages.x86_64-linux.cowsay}/bin/c
      } ;
    };
  };
}
```

Now, you can run `nix run` to run the cowsay app, which is equivalent to doing `nix build .#cowsay && ./result/bin/cowsay` in the previous flake.

Interlude: demo

```
❯ touch flake.nix

todo-app (076185e) [?] via λ
❯ # Add the file to git, for nix flakes to detect it

todo-app (076185e) [?] via λ
❯ git add flake.nix

todo-app (076185e) [+] via λ
❯ hx flake.nix
```

DevShells

To learn more

- Official Nix manual
- NixOS Wiki

Like `packages`, another predefined flake output is `devShells` - which is used to provide a development shell aka. a nix shell or devshell. A devshell is a sandboxed environment containing the packages and other shell environment

you specify. nixpkgs provides a function called `mkShell` that can be used to create devshells.

As an example, we will update our `flake.nix` to provide a devshell that contains the jq tool.

```
{
  inputs = {
    nixpkgs = {
      url = "github:NixOS/nixpkgs/nixos-unstable";
    };
  };
  outputs = inputs: {
    foo = 42;
    devShells = {  # nix develop
      aarch64-darwin = {
        default =
          let
            pkgs = inputs.nixpkgs.legacyPackages.aarch64-darwin;
          in pkgs.mkShell {
            packages = [
              pkgs.jq
            ];
          };
      };
    };
  };
}
```

`nix flake show` will recognize this output as a “development environmenet”:

```
$ nix flake show
path:/Users/srid/code/nixplay?lastModified=1675448105&narHash=sha256-dikTfYD1
├───devShells
│   └───aarch64-darwin
│       └───default: development environment 'nix-shell'
└───foo: unknown
```

Just as `packages` can be built using `nix build`, you can enter the devshell using `nix develop`:

```
$ nix develop
❯ which jq
/nix/store/33n0kx526i5dnv2gf39qv1p3a046p9yd-jq-1.6-bin/bin/jq
❯ echo '{"foo": 42}' | jq .foo
42
❯
```

Typing `Ctrl+D` or `exit` will exit the devshell.

Conclusion

This mini tutorial provided a rapid introduction to Nix flakes, enabling you to get started with writing simple flake for your projects. Consult the links above for more information. There is a lot more to Nix than the concepts presented here! You can also read Zero to Nix for a highlevel introduction to all things Nix and flakes.

See also

- A (more or less) one page introduction to Nix, the language
- Nix - taming Unix with functional programming

Links to this page

Nixifying a Haskell project using nixpkgs

> A basic understanding of the Nix and Flakes is assumed. See Rapid Introduction to Nix

Nix Tutorial Series

> ☑ Rapid Introduction to Nix

First steps with Nix

> See Rapid Introduction to Nix where we will go over writing simple Nix expressions and flakes.

> You have installed Nix. Now let's play with the `nix` command but without bothering to write any Nix expressions yet (we reserve that for the next tutorial). In particular, we will learn how to use packages from the nixpkgs repository and elsewhere.

Convert `configuration.nix` to be a flake

> [!info] More on Flakes See Rapid Introduction to Nix for more information on flakes.