A Beginner-friendly Flake Tutorial

Amy de Buitléir

Using a flake

Before learning to write Nix flakes, let's learn how to use them.

I've created a simple example of a flake in this repo. To run this flake, you don't need to install anything. Simply run the command below.

```
$ nix run "git+https://codeberg.org/mhwombat/hello-flake"
Hello from your flake!
```

That's a lot to type every time we want to use this package. Instead, we can enter a shell with the package available to us, using the `nix shell` command.

```
$ nix shell "git+https://codeberg.org/mhwombat/hello-flake"
```

In this shell, the command is our `$PATH`, so we can execute the command by name.

```
$ hello-flake
Hello from your flake!
```

Once we exit that shell, the `hello-flake` command is no longer available.

```
$ exit
$ hello-flake
sh: line 3: hello-flake: command not found
```

A brief examination of the flake

Let's clone the repo and see how the flake is defined.

```
$ cd ~/tutorial-practice
$ git clone https://codeberg.org/mhwombat/hello-flake
Cloning into 'hello-flake'...
$ cd hello-flake
$ ls
flake.lock
flake.nix
hello-flake
LICENSE
README.md
```

This is a simple repo with just a few files. Like most git repos, it includes `LICENSE`, which contains the software license, and `README.md` which provides information about the repo.

The `hello-flake` file is the command we were executing earlier. This particular executable is just a shell script, so we can view it. It's an extremly simple script with just two lines.

```
$ cat hello-flake
#!/usr/bin/env sh

echo "Hello from your flake!"
```

Now that we have a copy of the repo, we can execute this script directly.

```
$ ./hello-flake
Hello from your flake!
```

Not terribly exciting, I know. But starting with such a simple package makes it easier to focus on the flake system without getting bogged down in the details. We'll make this script a little more interesting later.

Let's look at another file. The file that defines how to package a flake is always called `flake.nix`.

```
$ cat flake.nix
{
  description = "a very simple and friendly flake";

  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 = import nixpkgs { inherit system; };
      in
      {
        packages = rec {
          hello = pkgs.stdenv.mkDerivation rec {
            name = "hello-flake";

            src = ./.;

            unpackPhase = "true";

            buildPhase = ":";

            installPhase =
              ''
                mkdir -p $out/bin
                cp $src/hello-flake $out/bin/hello-flake
                chmod +x $out/bin/hello-flake
              '';
          };
          default = hello;
        };

        apps = rec {
          hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };
          default = hello;
        };
      }
    );
}
```

If this is your first time seeing a flake definition, it probably looks intimidating. We'll look at this in more detail shortly. For now, I'd like to focus on the inputs section.

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

```
    flake-utils.url = "github:numtide/flake-utils";
  };
```

There are just two entries, one for `nixpkgs` and one for `flake-utils`. The first one, `nixpkgs` refers to the collection of standard software packages that can be installed with the Nix package manager. The second, `flake-utils`, is a collection of utilities that simplify writing flakes. The important thing to note is that the `hello-flake` package *depends* on `nixpkgs` and `flake-utils`.

Finally, let's look at `flake.lock`.

```
$ cat flake.lock
{
  "nodes": {
    "flake-utils": {
      "inputs": {
        "systems": "systems"
      },
      "locked": {
        "lastModified": 1681202837,
        "narHash": "sha256-H+Rh19JDwRtpVPAWp64F+rlEtxUWBAQW28eAi3SRSzg=",
        "owner": "numtide",
        "repo": "flake-utils",
        "rev": "cfacdce06f30d2b68473a46042957675eebb3401",
        "type": "github"
      },
      "original": {
        "owner": "numtide",
        "repo": "flake-utils",
        "type": "github"
      }
    },
    "nixpkgs": {
      "locked": {
        "lastModified": 1681665000,
        "narHash": "sha256-hDGTR59wC3qrQZFxVi2U3vTY+r02+Okbq080hO1C4Nk=",
        "owner": "NixOS",
        "repo": "nixpkgs",
        "rev": "3a6205d9f79fe526be03d8c465403b118ca4cf37",
        "type": "github"
      },
      "original": {
        "owner": "NixOS",
        "repo": "nixpkgs",
        "type": "github"
      }
    },
    "root": {
      "inputs": {
        "flake-utils": "flake-utils",
        "nixpkgs": "nixpkgs"
      }
    },
    "systems": {
      "locked": {
        "lastModified": 1681028828,
        "narHash": "sha256-Vy1rq5AaRuLzOxct8nz4T6wlgyUR7zLU309k9mBC768=",
        "owner": "nix-systems",
        "repo": "default",
```

```
        "rev": "da67096a3b9bf56a91d16901293e51ba5b49a27e",
        "type": "github"
      },
      "original": {
        "owner": "nix-systems",
        "repo": "default",
        "type": "github"
      }
    }
  },
  "root": "root",
  "version": 7
}
```

If `flake.nix` seemed intimidating, then this file looks like an invocation for Cthulhu. The good news is that this file is automatically generated; you never need to write it. It contains information about all of the dependencies for the flake, including where they came from, the exact version/revision, and hash. This lockfile *uniquely* specifies all flake dependencies, (e.g., version number, branch, revision, hash), so that *anyone, anywhere, any time, can re-create the exact same environment that the original developer used.*

No more complaints of "but it works on my machine!". That is the benefit of using flakes.

Flake structure

The basic structure of a flake is shown below.

```
{
  description = ... # package description
  inputs = ... # dependencies
  outputs = ... # what the flake produces
  nixConfig = ... # advanced configuration options
}
```

The `description` part is self-explanatory; it's just a string. You probably won't need `nixConfig` unless you're doing something fancy. I'm going to focus on what goes into the `inputs` and `outputs` sections, and highlight some of the things I found confusing.

Inputs

This section specifies the dependencies of a flake. It's an *attribute set*; it maps keys to values.

To ensure that a build is reproducible, the build step runs in a *pure* environment, with no access to anything except files that are part of its repo. Therefore, any external dependencies must be specified in the "inputs" section so they can be fetched in advance (before we enter the pure environment).

Each entry in this section maps an input name to a *flake reference*. This commonly takes the following form.

```
NAME.url = URL-LIKE-EXPRESSION
```

As a first example, all (almost all?) flakes depend on "nixpkgs", which is a large Git repository of programs and libraries that are pre-packaged for Nix. We can write that as

```
nixpkgs.url = "github:NixOS/nixpkgs/nixos-VERSION";
```

where `NN.MM` is replaced with the version number that you used to build the package, e.g. `22.11`. Information about the latest nixpkgs releases is available at https://status.nixos.org/. You can also write the entry without the version number

```
nixpkgs.url = "github:NixOS/nixpkgs/nixos-VERSION";
```

or more simply,

```
nixpkgs.url = "nixpkgs";
```

You might be concerned that omitting the version number would make the build non-reproducible. If someone else builds the flake, could they end up with a different version of nixpkgs? No! remember that the lockfile (`flake.lock`) *uniquely* specifies all flake inputs.

Git and Mercurial repositories are the most common type of flake reference, as in the examples below.

A Git repository `git+https://github.com/NixOS/patchelf` :
A specific branch of a Git repository `git+https://github.com/NixOS/patchelf?ref=master`
A specific branch and revision of a Git repository `git+https://github.com/NixOS/patchelf?ref=master&rev=`
A tarball `https://github.com/NixOS/patchelf/archive/master.tar.gz`

You can find more examples of flake references in the Nix Reference Manual.

Although you probably won't need to use it, there is another syntax for flake references that you might encounter. This example

```
inputs.import-cargo = {
  type = "github";
  owner = "edolstra";
  repo = "import-cargo";
};
```

is equivalent to

```
inputs.import-cargo.url = "github:edolstra/import-cargo";
```

Each of the `inputs` is fetched, evaluated and passed to the `outputs` function as a set of attributes with the same name as the corresponding input.

Outputs

This section is a function that essentially returns the recipe for building the flake.

We said above that `inputs` are passed to the `outputs`, so we need to list them as parameters. This example references the `import-cargo` dependency defined in the previous example.

```
outputs = { self, nixpkgs, import-cargo }: {
  ... outputs ...
};
```

So what actually goes in this section (where I wrote `...outputs...`)? That depends on the programming languages your software is written in, the build system you use, and more. There are Nix functions and tools that can simplify much of this, and new, easier-to-use ones are released regularly. We'll look at some of these in the next section. # A generic flake

The previous section presented a very high-level view of flakes, focusing on the basic structure. In this section, we will add a bit more detail.

Flakes are written in the Nix programming language, which is a functional language. As with most programming languages, there are many ways to achieve the same result. Below is an example you can follow when writing your own flakes. I'll explain the example in some detail.

```
{
  description = "BRIEF PACKAGE DESCRIPTION";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
    ...OTHER DEPENDENCIES... ❶
  };

  outputs = { self, nixpkgs, flake-utils, ...OTHER DEPENDENCIES... ❷ }:
    flake-utils.lib.eachDefaultSystem (system: ❸
      let
        pkgs = import nixpkgs { inherit system; };
        python = pkgs.python3;
      in
      {
        devShells = rec {
          default = pkgs.mkShell {
            packages = [ PACKAGES NEEDED FOR DEVELOPMENT SHELL; ❹ ]))
            ];
          };

        packages = rec {
          myPackageName = PACKAGE DEFINITION; ❺
          default = myPackageName;
        };

        apps = rec {
          myPackageName = flake-utils.lib.mkApp { drv = self.packages.${system}.myPackageName; };
          default = myPackageName;
        };
      }
    );
}
```

We discussed how to specify flake inputs ❶ in the previous section, so this part of the flake should be familiar. Remember also that any dependencies in the input section should also be listed at the beginning of the outputs section ❷.

Now it's time to look at the content of the output section. If we want the package to be available for multiple systems (e.g., "x86_64-linux", "aarch64-linux", "x86_64-darwin", and "aarch64-darwin"), we need to define the output for each of those systems. Often the definitions are identical, apart from the name of the system. The eachDefaultSystem function ❸ provided by flake-utils allows us to write a single definition using a variable for the system name. The function then iterates over all default systems to generate the outputs for each one.

The `devShells` variable specifies the environment that should be available when doing development on the package. If you don't need a special development environment, you can omit this section. At ❹ you would list any tools (e.g., compilers and language-specific build tools) you want to have available in a development shell. If the compiler needs access to language-specific packages, there are Nix functions to assist with that. These functions are very language-specific, and not always well-documented. We will see examples for some languages later in the tutorial. In general, I recommend that you do a web search for "nix language", and try to find resources that were written or updated recently.

The `packages` variable defines the packages that this flake provides. The package definition ❺ depends on the programming languages your software is written in, the build system you use, and more. There are Nix functions and tools that can simplify much of this, and new, easier-to-use ones are released regularly. Again, I recommend that you do a web search for "nix language", and try to find resources that were written or updated recently.

The list below contains are a few functions that are commonly used in this section.

General-purpose The standard environment provides `mkDerivation`, which is especially useful for the typical ./configure; make; make install scenario. It's customisable.

Python `buildPythonApplication`, `buildPythonPackage`.

Haskell `mkDerivation` (Haskell version, which is a wrapper around the standard environment version), `developPackage`, `callCabal2Nix`.

Modifying the flake

Let's make a simple modification to the script. This will give you an opportunity to check your understanding of flakes.

The first step is to enter a development shell.

```
$ cd ~/tutorial-practice/hello-flake
$ nix develop
```

The `flake.nix` file specifies all of the tools that are needed during development of the package. The `nix develop` command puts us in a shell with those tools. As it turns out, we didn't need any extra tools (beyond the standard environment) for development yet, but that's usually not the case. Also, we will soon need another tool.

A development environment only allows you to *develop* the package. Don't expect the package *outputs* (e.g. executables) to be available until you build them. However, our script doesn't need to be compiled, so can't we just run it?

```
$ hello-flake
bash: line 1: hello-flake: command not found
```

That worked before, why isn't it working now? Earlier we used `nix shell` to enter an environment where `hello-flake` was available and on the `$PATH`. In a *development* environment, the executable won't be on the `$PATH`; the flake hasn't been built yet! We can, however, run it by specifying the path to the script.

```
$ ./hello-flake
Hello from your flake!
```