Nix flakes

Eelco Dolstra

Introduction

Flakes are **a way to package your Nix files**.

Goals:

- Support multi-repo Nix-based projects
- Standard structure and better discoverability for Nix-based projects
- Better reproducibility
- Replace `nix-channel`
- Finish the `nix` CLI

Goal: Support multi-repo projects

Current: ad hoc composition using `builtins.fetchGit { rev = …; }`, `NIX_PATH`, relative paths, …

Flakes: Make it easy to build a project that depends on other repos. No need to fiddle with `NIX_PATH`, manually fetch other repos, ...

Goal: Structure and discoverability

Current: projects have ad-hoc structure:

- File naming
- Attribute naming / semantics
- Inter-repository dependencies

Flakes: provide a standard way to organize, build and query a project.

Goal: Reproducibility

Current: reproducible builds but not reproducible / hermetic evaluation. Nix expressions can depend on

- Arbitrary files
- Git repositories (without specifying a revision)
- Environment variables
- ...

Flakes: If two users evaluate the same attribute from the same revision of the same flake, they will get the same result. Allows persistent evaluation caching.

Goal: nix-channel replacement

Current: `nix-channel` is pretty annoying:

- Hard to set up a channel (they're not Git repos).
- Can't easily pin a channel to a specific version.
- Channels don't auto-update.
- No inter-repo dependencies.

Flakes: Replace channels and provide a much nicer UX.

Goal: Finish the `nix` CLI

Current: The `nix` command:

- Lacks replacements for `nix-env`, `nix-shell` and `nix-channel`
- Has an unfinished notion of "installables" (what does "`nix build nixpkgs.hello`" mean?)

Flakes: Provides replacements for those commands, with a consistent user interface.

Using flakes

```
# nix run nixpkgs#rustc -c rustc --version
rustc 1.38.0

# nix run nixpkgs/release-19.09#rustc -c rustc --version
rustc 1.37.0

# nix run patchelf -c patchelf --version
patchelf 0.10.20191023.2ba6481

# nix run github:edolstra/dwarffs/50023d28e814… -c …
```

Overriding the registry

```
# nix flake pin nixpkgs

# nix flake add nixpkgs nixpkgs/release-19.09
# nix run nixpkgs#rustc -c rustc --version
rustc 1.37.0
```

Installing packages

```
# nix profile install nixpkgs#hello

# nix profile upgrade
```

Hacking on a flake

```
# nix flake clone nixos-homepage
# cd nixos-homepage
# emacs index.tt

# nix build
# firefox ./result/index.html
# nix build .#packagesExplorer
# nix dev-shell
# nix flake check

# nixos-container create homepage --flake nixos-homepage
# nixos-container start homepage
# firefox http://$(nixos-container show-ip homepage)
```

What is a flake?

A flake is a Git repository containing a file named `flake.nix`, which defines:

- Metadata: description, edition, ...
- Dependencies on other flakes / repositories
- Outputs: the Nix values provided by the flake

Example of a flake.nix

```
{
  description = "A filesystem that fetches DWARF debug info on demand";

  edition = 201909;

  outputs = { self, nixpkgs }: {

    packages.x86_64-linux.dwarffs =
      with import nixpkgs { system = "x86_64-linux"; }
      stdenv.mkDerivation { …; src = ./.; };

    defaultPackage.x86_64-linux = self.packages.x86_64-linux.dwarffs;

    nixosModules.dwarffs = { config, … }: { … };

  };
}
```

What are the outputs of a flake?

- Packages
- Nixpkgs overlays
- NixOS modules
- NixOS system configurations
- CI jobs
- Development environments
- ...

Lock files

- Flakes are evaluates in pure mode.
- Pure evaluation requires that all dependencies are exact (i.e. contain a Git revision / content hash).
- Therefore, flakes have a lock file (`flake.lock`) that maps the inputs to exact (“immutable”) locations like `github:NixOS/nixpkgs/6987e52d407`….
- Lock files are generated automatically if they don’t exist yet.

Continuous integration

Editing jobset flakes:nixos-homepage

State: Enabled | One-shot | Disabled

☑ Visible

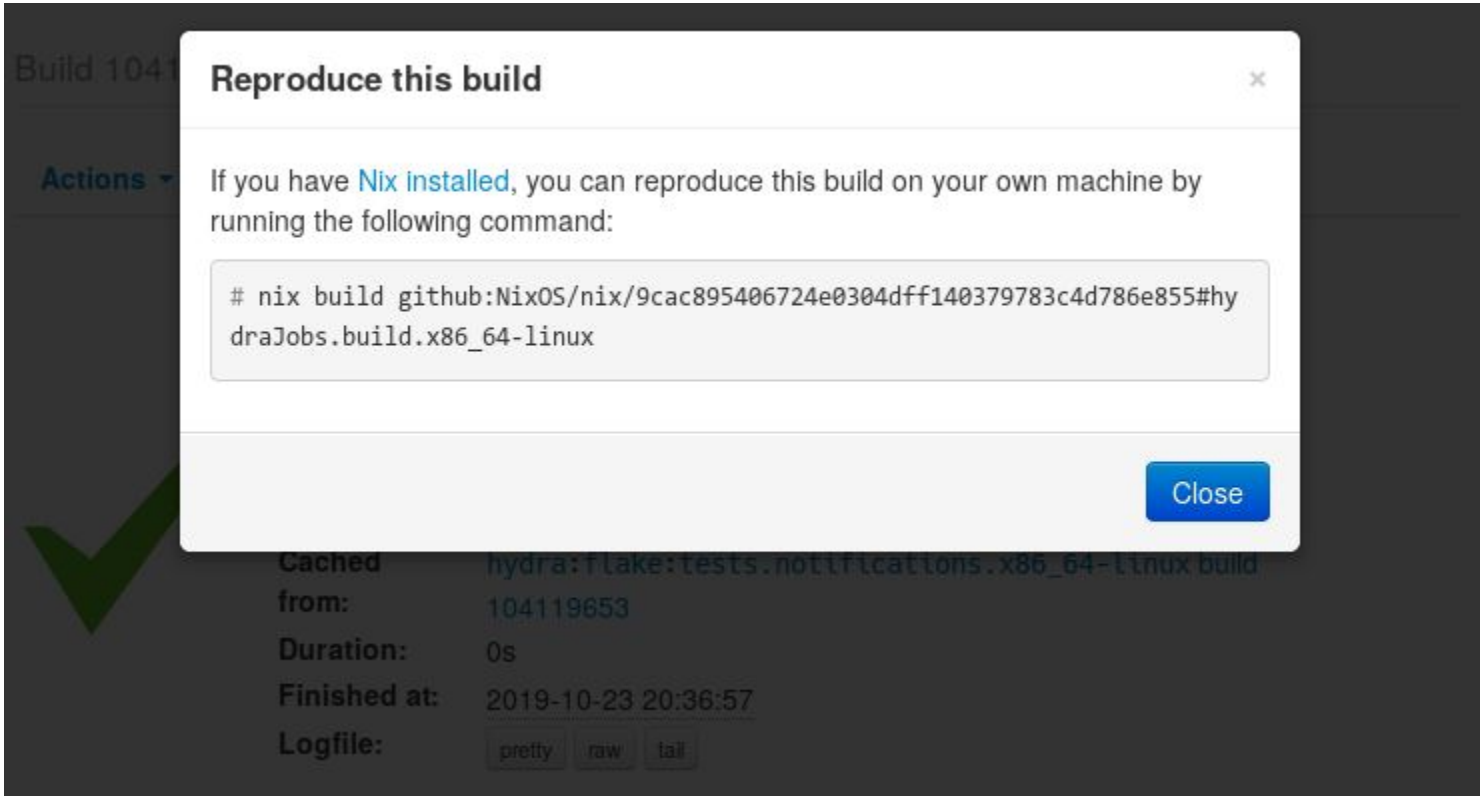
Identifier: nixos-homepage

Description:

Type: Flake | Legacy

Flake URI: github:NixOS/nixos-homepage/flake

Continuous integration



Flake support in NixOS

```
# nixos-rebuild switch --flake /path/to/flake

# nixos-version --json
{"nixosVersion": "19.09.20191023.bcceb88"
,"nixpkgsRevision": "bcceb882ccdf…"
,"configurationRevision": "fcf6d38a7f8a…"
}
```

NixOS configuration

```
{ edition = 201909;
  inputs.nixpkgs.url = "nixpkgs/release-19.09";
  outputs = { self, nixpkgs, dwarffs, hydra, nix }: {
    nixosConfigurations.machine = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules =
        [ dwarffs.nixosModules.dwarffs
          hydra.nixosModules.hydra
          { nixpkgs.overlays = [ nix.overlay ];
            fileSystems = { ... }; ...
          }
        ];
    };
  };
}
```

Evaluation caching

```
# time nix build nixpkgs#firefox
real    0m1.497s

# time nix build nixpkgs#firefox
real    0m0.052s
```

Availability

- RFC 49
- Binaries
- Source

Misc

Flake references

A flake reference specifies the location of a flake, or specifies a flake ID to be looked up in the registry. Examples

- `/path/to/my-flake` - local Git repo
- `https://github.com/NixOS/nixpkgs.git` - a remote Git repo
- `github:NixOS/nixpkgs` - a special syntax (and semantics!) for GitHub repos
- `nixpkgs` - an indirection through the flake registry

Flake references (cont'd)

Flake references can also specify branches/tags or revisions:

- `github:NixOS/nixpkgs/18.09` - a branch
- `github:NixOS/nixpkgs/6987e52d407...` - a revision
- `nixpkgs/18.09` - a branch applied to a registry indirection

Nix apps

```
# nix app blender-bin
```

Other commands

- `nix flake clone`
- `nix flake info`
- `nix flake init`