Binary Cache

From NixOS Wiki

This article or section needs expansion.

Reason: What is the format of a binary cache? How does it differ from a local/remote Nix store? See `NixOS/nix` PR #6870 (https://github.com/NixOS/nix/pull/6870). (Maybe even splitting it into a guide and a reference?) (Discuss in Talk:Binary Cache# (https://nixos.wiki/wiki/Talk:Binary_Cache))

Please consult the pedia article metapage (/wiki/Category:Pedias) for guidelines on contributing.

A binary cache builds Nix packages and caches the result for other machines. Any machine with Nix installed can be a binary cache for another one, no matter the operating system.

If you are facing problems with derivations not being in a cache, try switching to a release version. Most caches will have many derivations for a specific release.

Setting up a binary cache

This tutorial explains how to setup a machine as a binary cache for other machines, serving the nix store on TCP port 80 with signing turned on. It assumes that an `nginx (/wiki/Nginx)` service is already running, that port 80 is open,[cf. 1] and that the hostname `binarycache.example.com` resolves to the server.[cf. 2]

1. Generating a private/public keypair

A keypair is necessary to sign Nix packages. Replace `binarycache.example.com` with your domain.

```
cd /var
nix-store --generate-binary-cache-key binarycache.example.com cache-priv-key.pem cache-
chown nix-serve cache-priv-key.pem
chmod 600 cache-priv-key.pem
cat cache-pub-key.pem
```

The packages can be signed before adding them to the binary cache, or on the fly as they are served. In this tutorial we'll set up `nix-serve` to sign packages on the fly when it serves them. In this case it is important that only `nix-serve` can access the private key. The location `/var/cache-priv-key.pem` is just an example.

2. Activating `nix-serve`

`nix-serve` is the service that speaks the binary cache protocol via HTTP.

To start it on NixOS:

```
services.nix-serve = {
  enable = true;
  secretKeyFile = "/var/cache-priv-key.pem";
};
```

To start it on a non-NixOS machine at boot, add to `/etc/crontab` :

```
NIX_SECRET_KEY_FILE=/var/cache-priv-key.pem
@reboot /home/USER/.nix-profile/bin/nix-serve --listen :5000 --error-log /var/log/nix-s
```

`nix-serve` will by default serve on port 5000. We are not going to open a firewall port for it, because we will let `nginx` redirect to it.

3. Creating a virtual hostname in `nginx`

We redirect the HTTP(s) traffic from port 80 to `nix-serve` . As `nix-serve` is capable of serving only on IPv4, redirecting is also useful to make the binary cache available on IPv6.

```
services.nginx = {
  enable = true;
  recommendedProxySettings = true;
  virtualHosts = {
    # ... existing hosts config etc. ...
    "binarycache.example.com" = {
      locations."/".proxyPass = "http://${config.services.nix-serve.bindAddress}:${toSt
    };
  };
};
```

Add HTTPS settings to this config if possible.[cf. 3] This tutorial will simply continue with insecure HTTP.

To set up Nginx on a non-NixOS machine, create for example `/etc/nginx/sites-enabled/nix-serve.conf` :

```
server {
    listen      80 default_server;
    listen      [::]:80 default_server;

    location / {
        proxy_pass  http://127.0.0.1:5000;
    }
}
```

4. Testing

To apply the previous settings to your NixOS machine, run:

```
# nixos-rebuild switch
```

Check the general availability:

```
$ curl http://binarycache.example.com/nix-cache-info
StoreDir: /nix/store
WantMassQuery: 1
Priority: 30
```

On the binary cache server, build some package:

```
$ nix-build '<nixpkgs>' -A pkgs.hello
/nix/store/gdh8165b7rg4y53v64chjys7mbbw89f9-hello-2.10
```

To verify the signing on the fly, make sure the following request contains a `Sig:` line:

```
$ curl http://binarycache.example.com/gdh8165b7rg4y53v64chjys7mbbw89f9.narinfo
StorePath: /nix/store/gdh8165b7rg4y53v64chjys7mbbw89f9-hello-2.10
URL: nar/gdh8165b7rg4y53v64chjys7mbbw89f9.nar
Compression: none
NarHash: sha256:0mkfk4iad66xkld3b7x34n9kxri9lrpkgk8m17p97alacx54h5c7
NarSize: 205920
References: 6yaj6n8l925xxfbcd65gzqx3dz7idrnn-glibc-2.27 gdh8165b7rg4y53v64chjys7mbbw89f
Deriver: r6h5b3wy0kwx38rn6s6qmmfq0svcnf86-hello-2.10.drv
Sig: binarycache.example.com:EmAANryZ1FFHGmz5P+HXLSDbc0KckkBEAkHsht7gEIOUXZk9yhhZSBV+eS
```

Next, with the public key that was generated to `cache-pub-key.pem`, setup a client machine to use the binary cache, and see if Nix successfully fetches the cached package.

Using a binary cache

To configure Nix to use a certain binary cache, refer to the Nix manual.[cf. 4] Add the binary cache as substituter (see the options `substituters` and `extra-substituters`) and the public key to the trusted keys (see `trusted-public-keys`).

Permanent use of binary cache:

```
# /etc/nixos/configuration.nix

  nix = {
    settings = {
      substituters = [
        "http://binarycache.example.com"
        "https://nix-community.cachix.org"
        "https://cache.nixos.org/"
      ];
      trusted-public-keys = [
        "binarycache.example.com-1:dsafdafDFW123fdasfa123124FADSAD"
        "nix-community.cachix.org-1:mB9FSh9qf2dCimDSUo8Zy7bkq5CX+/rkCWyvRCYg3Fs="
      ];
    };
  };
```

Warning: Keys that are entered incorrectly or are otherwise invalid, aside from preventing you from

benefiting from the cached derivations, may also prevent you from rebuilding your system. This is most likely to occur after garbage collection (e.g., via `nix-collect-garbage -d`). Consult NixOS/nix#8271 (https://github.com/NixOS/nix/issues/8271) for additional details and a workaround.

Temporary use of binary cache:

```
$ nix-store -r /nix/store/gdh8165b7rg4y53v64chjys7mbbw89f9-hello-2.10 --option substitu
these paths will be fetched (0.00 MiB download, 24.04 MiB unpacked):
  /nix/store/7gx4kiv5m0i7d7qkixq2cwzbr10lvxwc-glibc-2.27
  /nix/store/gdh8165b7rg4y53v64chjys7mbbw89f9-hello-2.10
copying path '/nix/store/7gx4kiv5m0i7d7qkixq2cwzbr10lvxwc-glibc-2.27' from 'http://bina
copying path '/nix/store/gdh8165b7rg4y53v64chjys7mbbw89f9-hello-2.10' from 'http://bina
warning: you did not specify '--add-root'; the result might be removed by the garbage c
/nix/store/gdh8165b7rg4y53v64chjys7mbbw89f9-hello-2.10
```

Using a binary cache on non-NixOS installations

To use a binary cache with a Nix that has been installed on an operating system other than NixOS (e.g. Ubuntu or macOS) `/etc/nix/nix.conf` will need to be edited manually. This can be done by adding something similar to the following lines to `/etc/nix/nix.conf`:

```
trusted-public-keys = nix-community.cachix.org-1:mB9FSh9qf2dCimDSUo8Zy7bkq5CX+/rkCWyvRC
trusted-substituters = https://nix-community.cachix.org https://cache.nixos.org
trusted-users = root @wheel
```

Note that not all of that information is needed, see the manual for the respective options (trusted-public-keys (https://nixos.org/manual/nix/stable/command-ref/conf-file.html#conf-trusted-public-keys), trusted-substituters (https://nixos.org/manual/nix/stable/command-ref/conf-file.html#conf-trusted-substituters), trusted-users (https://nixos.org/manual/nix/stable/command-ref/conf-file.html#conf-trusted-users)).

With the `trusted-*` options set correctly, a user can benefit permanently from a substituter by add the following to their `~/.config/nix/nix.conf`

```
substituters = https://nix-community.cachix.org https://cache.nixos.org
```

or temporarily as explained above.

Binary cache hint in Flakes

You can place a hint to your binary cache in your Flake so when someone builds an output of your Flake, the nix command will ask interactively to trust the specified binary cache.

```
{
  outputs = { ... }: {
    nixConfig = {
      extra-substituters = [
        "https://colmena.cachix.org"
      ];
      extra-trusted-public-keys = [
        "colmena.cachix.org-1:7BzpDnjjH8ki2CT3f6GdOk7QAzPOl+1t3LvTLXqYcSg="
      ];
    };
  };
}
```

Populating a binary cache

As the cache is served from the nix store of the machine serving the binary cache, one option is to build the packages directly on that machine.

Another option is to build the packages on a separate machine and push them only when all the checks pass using `nix copy` :

```
$ nix copy --to ssh://binarycache.example.com PACKAGE
```

For details see the Sharing Packages Between Machines (https://nixos.org/manual/nix/stable/package-management/sharing-packages.html) in the Nix manual.

Hosted binary cache

https://cachix.org (https://cachix.org) provides hosted binary caches starting with a free plan for public caches.

How to check if content is on a binary cache

We can use **curl** to check if a binary cache contains a given derivation. *curl https://cache/store_hash.narinfo (https://cache/store_hash.narinfo)*

```
$ curl https://fzakaria.cachix.org/949dxjmz632id67hjic04x6f3ljldzxh.narinfo

StorePath: /nix/store/949dxjmz632id67hjic04x6f3ljldzxh-mvn2nix-0.1
URL: nar/4026897ef85219b5b697c1c4ef30d50275423857cb7a81e138c4b1025f550935.nar.xz
Compression: xz
FileHash: sha256:4026897ef85219b5b697c1c4ef30d50275423857cb7a81e138c4b1025f550935
FileSize: 24392
NarHash: sha256:0kk3d8rk82ynqwg8isk83hvq8vszh4354fqg4hhaz40kd49rmm9n
NarSize: 29208
References: 2hhmmj0vbb5d181nfx2mx3p7k54q44ij-apache-maven-3.6.3 6737cq9nvp4k5r70qcgf610
Deriver: 585w6p8rclbvz97fwgixvfgnh5493ia2-mvn2nix-0.1.drv
Sig: fzakaria.cachix.org-1:MkOrZCa9qdxHFdE2mtFRsbEzmLUgWGgSrqD3advKfdHLW+SKxj/V2n6+4a/q
```

Or use *nix path-info*:

```
$ nix path-info -r /nix/store/sb7nbfcc1ca6j0d0v18f7qzwlsyvi8fz-ocaml-4.10.0 --store htt
[0.0 MiB DL] querying libunistring-0.9.10 on https://cache.nixos.org/nix/store/0gc9dr71
/nix/store/2jysm3dfsgby5sw5jgj43qjrb5v79ms9-bash-4.4-p23
/nix/store/4wy9j24psf9ny4di3anjs7yk2fvfb0gq-glibc-2.31-dev
/nix/store/4z79ipgxqn80ns7mpax25zmb77i3ndfw-gawk-5.1.0
/nix/store/9df65igwjmf2wbw0gbrrgair6piqjgmi-glibc-2.31
/nix/store/czc3c1apx55s37qx4vadqhn3fhikchxi-libunistring-0.9.10
/nix/store/fgn3sih5vi7543jcw389a7qqax8nwkhz-glibc-2.31-bin
/nix/store/sb7nbfcc1ca6j0d0v18f7qzwlsyvi8fz-ocaml-4.10.0
/nix/store/xim9l8hym4iga6d4azam4m0k0p1nw2rm-libidn2-2.3.0
```

Example: Fetch metadata of `bash`

```
curl https://cache.nixos.org/$(readlink -f $(which bash) | cut -c12-43).narinfo
```

See also

1. NixOS Manual, 11.5. Firewall (https://nixos.org/nixos/manual/#sec-firewall)
2. `networking.hostName (https://search.nixos.org/options?query=networking.hostName)`
3. NixOS Manual, 26.3. Using ACME certificates in Nginx (https://nixos.org/nixos/manual/#module-security-acme-nginx)
4. Nix Manual, 21. Files (https://nixos.org/nix/manual/#ch-files)

- How to use binary cache in NixOS (https://discourse.nixos.org/t/how-to-use-binary-cache-in-nixos/5202/4)

Retrieved from "https://nixos.wiki/index.php?title=Binary_Cache&oldid=10880 (https://nixos.wiki/index.php?title=Binary_Cache&oldid=10880)"

Categories (/wiki/Special:Categories):

Pages with syntax highlighting errors (/index.php?title=Category:Pages_with_syntax_highlighting_errors&action=edit&redlink=1)

Pages or sections flagged with Template:Expansion (/wiki/Category:Pages_or_sections_flagged_with_Template:Expansion)

Getting Started

Fork me on GitHub

Creating the cache

With Nix and Cachix installed you can cache any Nix build result.

After logging into Cachix you'll be able to create a new binary cache.

It's recommended to setup new binary caches **per trust model**, depending on who are the users that will have write access and the same for read access.

Most teams will have one private and one public cache.

Public caches have read access open to everyone.

If you'd like to keep your **binaries protected from public access**, make sure to create a private cache.

Authenticating

There are two kinds of auth tokens:

a. Personal

   These allow full access to your account and can be generated here.

b. Per-cache

   These allow write and/or read access to only a specific binary cache. On dashboard you can click on your newly generated binary cache *Settings* and generate new term:*access token*.

   You can set auth token either with:

   a. `$ cachix authtoken XXX`
   b. `$ export CACHIX_AUTH_TOKEN=XXX`

Signing key (advanced mode)

> **Note:**
>
> In case you didn't opt-in to self-generated signing key, you can skip this step.

Read a blog post on upsides and downsides of self-generated signing key.

To generate a signing key:

```
$ cachix authtoken <my auth token>
$ cachix generate-keypair <mycache>
```

Signing key is saved locally on your computer (the only copy!) and printed to stdout, make sure to make a backup.

Cachix will automatically pick up the recently written signing key (or if you export it via environment variable `$CACHIX_SIGNING_KEY`).

Pushing binaries with Cachix

Assuming you have a project with `default.nix` you can build it and push:

```
$ nix-build | cachix push mycache
```

It's recommended to set up Continuous Integration to push for every branch of every project.

See all differents ways of pushing.

Using binaries with Nix

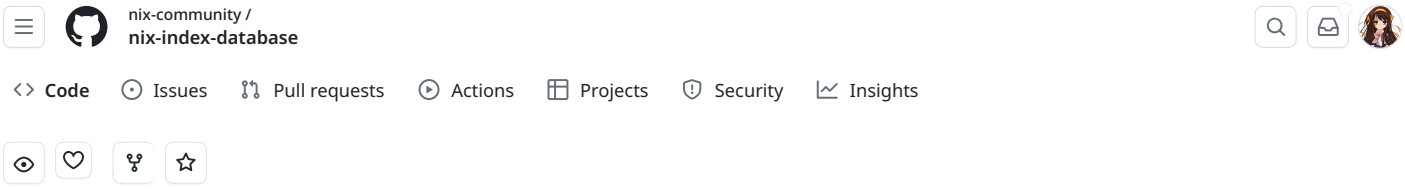
> ## Note:
>
> For read access to private caches you'll also need to run `cachix authtoken XXX` or export *$CACHIX_AUTH_TOKEN* before invoking `cachix use` in order to configure access token, to be used for authenticating using netrc file.

With Nix and Cachix installed invoke:

```
$ cachix use mycache
```

to configure Nix to use your binary cache.

There are different ways to configure Nix so Cachix will pick the most appropriate one for your setup.



Weekly updated nix-index database [maintainer=**@Mic92**]

MIT license
Code of conduct
Security policy
186 stars **13** forks **5** watching **5** Branches **159** Tags Activity Custom properties
Public repository

main 5 Branches 159 Tags Go to file t Go to file Add file Code

github-actions[bot] flake.lock: Update 9 minutes ago

.github	Bump cachix/install-nix-action from 24 to 25	3 weeks ago
.mergify.yml	mergify: also merge dependabot	2 months ago
LICENSE	Initial commit	4 years ago
README.md	also integrate comma	10 months ago
comma-wrapper.nix	Comma: use the NIX_INDEX_DATABASE env...	8 months ago
darwin-module.nix	Avoid calling builtins.import on modules.	2 months ago
flake.lock	flake.lock: Update	9 minutes ago
flake.nix	Avoid calling builtins.import on modules.	2 months ago
garnix.yaml	Exclude aarch64 VM test on Garnix.	8 months ago
home-manager-module.nix	Avoid calling builtins.import on modules.	2 months ago
nix-index-wrapper.nix	Use the NIX_INDEX_DATABASE env var if po...	8 months ago
nixos-module.nix	Avoid calling builtins.import on modules.	2 months ago
packages.nix	update packages.nix to release 2024-01-28-...	last week
tests.nix	Fix issue with checks during scheduled wor...	8 months ago

README Code of conduct MIT license Security

nix-index-database

Weekly updated nix-index database

This repository also provides nixos modules and home-manager modules that add a `nix-index` wrapper to use the database from this repository.

The home-manager module also allows integration with the existing `command-not-found` functionality.

Demo

```
$ nix run github:mic92/nix-index-database bin/cntr
cntr.out                                  978,736 x /nix/store/09p2hys5bxcnzcaad3bknlnwsgdkznl1-cntr-1.5.1/bin/
```

Usage in NixOS

Include the nixos module in your configuration (requires 23.05 or nixos unstable)

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

    nix-index-database.url = "github:Mic92/nix-index-database";
    nix-index-database.inputs.nixpkgs.follows = "nixpkgs";
  };

  outputs = { self, nixpkgs, nix-index-database, ... }: {
    nixosConfigurations = {
```

```
      my-nixos = nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        modules = [
          ./configuration.nix
          nix-index-database.nixosModules.nix-index
          # optional to also wrap and install comma
          # { programs.nix-index-database.comma.enable = true; }
        ];
      };
    };
  };
}
```

Usage in Home-manager

1. Follow the manual to set up home-manager with flakes.
2. Include the home-manager module in your configuration:

```
{
  inputs = {
    # Specify the source of Home Manager and Nixpkgs.
    nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
    home-manager = {
      url = "github:nix-community/home-manager";
      inputs.nixpkgs.follows = "nixpkgs";
    };

    nix-index-database.url = "github:Mic92/nix-index-database";
    nix-index-database.inputs.nixpkgs.follows = "nixpkgs";
  };
  outputs = { nixpkgs, home-manager, nix-index-database, ... }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
    in {
      homeConfigurations.jdoe = home-manager.lib.homeManagerConfiguration {
        inherit pkgs;

        modules = [
          nix-index-database.hmModules.nix-index
          # optional to also wrap and install comma
          # { programs.nix-index-database.comma.enable = true; }
        ];
      };
    };

}
```

Additionally, if your shell is managed by home-manager, you can have `nix-index` integrate with your shell's `command-not-found` functionality by setting `programs.nix-index.enable = true`.

Ad-hoc download

```
download_nixpkgs_cache_index () {
  filename="index-$(uname -m | sed 's/^arm64$/aarch64/')-$(uname | tr A-Z a-z)"
  mkdir -p ~/.cache/nix-index && cd ~/.cache/nix-index
  # -N will only download a new version if there is an update.
  wget -q -N https://github.com/Mic92/nix-index-database/releases/latest/download/$filename
  ln -f $filename files
}

download_nixpkgs_cache_index
```

Releases 159
Sponsor this project

Release 2024-01-28-030920 Latest
1 week

opencollective.com/**nix-community**

+ 158 releases

Contributors 15

Languages

- **Nix** 100.0%

Automated, pre-built packages for Wayland (sway/wlroots) tools for NixOS. [maintainers=**@colemickens**, **@Artturin**]

Code of conduct

Security policy

438 stars **45** forks **10** watching **1** Branch **0** Tags Activity Custom properties

Public repository

master 1 Branch 0 Tags Go to file t Go to file Add file Code

colebot auto-update: sway-unwrapped: ba427a469a8394cb2fafa1a0602d6fd78c5c4d68... 5 hours ago

.github	fix conf key	2 weeks ago
demo	statix fix	3 days ago
pkgs	auto-update: sway-unwrapped: ba427a469...	5 hours ago
templates	statix fix	3 days ago
.envrc	.envrc: move to nix-direnv and flake support	4 years ago
.gitignore	flakes: initial support	4 years ago
README.md	auto-update: updated readme	2 weeks ago
default.nix	fix compat files	10 months ago
flake.lock	flake.lock: Update	yesterday
flake.nix	statix fix	3 days ago
main.nu	fix main.nu	6 months ago
overlay.nix	fix compat files	10 months ago
packages.nix	init wbg	4 months ago
shell.nix	flake: switch to nix-community/flake-compat	last year

README Code of conduct Security

nixpkgs-wayland

overview

Automated, pre-built, (potentially) pre-release packages for Wayland (sway/wlroots) tools for NixOS (**nixos-unstable** channel).

These packages are auto-updated to the latest version available from their upstream source control. This means this overlay and package set will often contain **unreleased** versions.

Community chat is on Matrix: #nixpkgs-wayland:matrix.org. We are not on Libera.

Started by: @colemickens and co-maintained by @Artturin (🙏).

- Development Guide

Usage

Binary Cache

The Cachix landing page for `nixpkgs-wayland` shows how to utilize the binary cache.

Packages from this overlay are regularly built against `nixos-unstable` and pushed to this cache.

Continuous Integration

We have multiple CI jobs:

1. Update - this tries to advance nixpkgs and upgrade all packages. If it can successfully build them, the updates are push to master
2. Advance - this tries to advance nixpkgs without touching the packages. This can help show when nixpkgs upgrades via `nixos-unstable` has advanced into a state where we are broken building against it.
3. Build - this just proves that `master` was working against `nixos-unstable` at the point in time captured by whatever is in `flake.lock` on `master`.

We don't have CI on Pull Requests, but I keep an eye on it after merging external contributions.

Flake Usage

- Build and run the Wayland-fixed up version of OBS-Studio:

```
nix shell "github:nix-community/nixpkgs-wayland#obs-studio" --command obs
```

- Build and run `waybar`:

```
nix run "github:nix-community/nixpkgs-wayland#waybar"
```

- Use as an overlay or package set via flakes:

```
{
  inputs = {
    nixpkgs-wayland.url = "github:nix-community/nixpkgs-wayland";

    # only needed if you use as a package set:
    nixpkgs-wayland.inputs.nixpkgs.follows = "nixpkgs";
  };

  outputs = inputs: {
    nixosConfigurations."my-laptop-hostname" =
    let system = "x86_64-linux";
    in nixpkgs.lib.nixosSystem {
      inherit system;
      modules = [({pkgs, config, ... }: {
        config = {
          nix.settings = {
            # add binary caches
            trusted-public-keys = [
              "cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY="
              "nixpkgs-wayland.cachix.org-1:3lwxaILxMRkVhehr5StQprHdEo4IrE8sRho9R9HOLYA="
            ];
            substituters = [
              "https://cache.nixos.org"
              "https://nixpkgs-wayland.cachix.org"
            ];
          };

          # use it as an overlay
          nixpkgs.overlays = [ inputs.nixpkgs-wayland.overlay ];

          # or, pull specific packages (built against inputs.nixpkgs, usually `nixos-unstable`)
          environment.systemPackages = [
            inputs.nixpkgs-wayland.packages.${system}.waybar
          ];
```

```
            ];
          };
        })];
      };
    };
  }
```

Install for NixOS (non-flakes, manual import)

If you are not using Flakes, then consult the NixOS Wiki page on Overlays. Also, you can expand this section for a more literal, direct example. If you do pin, use a tool like `niv` to do the pinning so that you don't forget and wind up stuck on an old version.

▸ Details

Install for non-NixOS users

Non-NixOS users have many options, but here are two explicitly:

1. Activate flakes mode, then just run them outright like the first example in this README.
2. See the following details block for an example of how to add `nixpkgs-wayland` as a user-level overlay and then install a package with `nix-env`.

▸ Details

Packages

Package	Description
aml	Another main loop
cage	A Wayland kiosk that runs a single, maximized application
drm_info	Small utility to dump info about DRM devices
dunst	Lightweight and customizable notification daemon
eww	ElKowars wacky widgets
eww-wayland	ElKowars wacky widgets
foot	A fast, lightweight and minimalistic Wayland terminal emulator
freerdp3	A Remote Desktop Protocol Client
gebaar-libinput	Gebaar, A Super Simple WM Independent Touchpad Gesture Daemon for libinput
glpaper	Wallpaper program for wlroots based Wayland compositors such as sway that allows you to render glsl shaders as your wallpaper
grim	Grab images from a Wayland compositor
gtk-layer-shell	A library to create panels and other desktop components for Wayland using the Layer Shell protocol
i3status-rust	Very resource-friendly and feature-rich replacement for i3status
imv	A command line image viewer for tiling window managers
kanshi	Dynamic display configuration tool
lavalauncher	A simple launcher panel for Wayland desktops
libvncserver_master	VNC server library
mako	A lightweight Wayland notification daemon
neatvnc	A VNC server library
new-wayland-protocols	Wayland protocol extensions
obs-wlrobs	An obs-studio plugin that allows you to screen capture on wlroots based wayland compositors
rootbar	A bar for Wayland WMs

~~toolbar~~	~~A bar for Wayland WMs~~
salut	A sleek notification daemon
shotman	The uncompromising screenshot GUI for Wayland compositors
sirula	Simple app launcher for wayland written in rust
slurp	Select a region in a Wayland compositor
sway-unwrapped	An i3-compatible tiling Wayland compositor
swaybg	Wallpaper tool for Wayland compositors
swayidle	Idle management daemon for Wayland
swaylock	Screen locker for Wayland
swaylock-effects	Screen locker for Wayland
swww	Efficient animated wallpaper daemon for wayland, controlled at runtime
waybar	Highly customizable Wayland bar for Sway and Wlroots based compositors
waypipe	A network proxy for Wayland clients (applications)
wayprompt	multi-purpose prompt tool for Wayland
wayvnc	A VNC server for wlroots based Wayland compositors
wbg	Wallpaper application for Wayland compositors
wdisplays	A graphical application for configuring displays in Wayland compositors
wev	Wayland event viewer
wf-recorder	Utility program for screen recording of wlroots-based compositors
wl-clipboard	Command-line copy/paste utilities for Wayland
wl-gammarelay-rs	A simple program that provides DBus interface to control display temperature and brightness under wayland without flickering
wl-screenrec	High performance wlroots screen recording, featuring hardware encoding
wlay	Graphical output management for Wayland
wldash	Wayland launcher/dashboard
wlogout	A wayland based logout menu
wlr-randr	An xrandr clone for wlroots compositors
wlroots	A modular Wayland compositor library
wlsunset	Day/night gamma adjustments for Wayland
wlvncc	A Wayland Native VNC Client
wob	A lightweight overlay bar for Wayland
wofi	A launcher/menu program for wlroots based wayland compositors such as sway
wshowkeys	Displays keys being pressed on a Wayland session
wtype	xdotool type for wayland
xdg-desktop-portal-wlr	xdg-desktop-portal backend for wlroots

Tips

General

- I recommend using `home-manager` !
- It has modules and options for:
  - `swav`

- `waybar`
- `obs` and plugins!
- more!

sway

- You will likely want a default config file to place at `$HOME/.config/sway/config`. You can use the upstream default as a starting point: https://github.com/swaywm/sway/blob/master/config.in
- If you start `sway` from a raw TTY, make sure you use `exec sway` so that if sway crashes, an unlocked TTY is not exposed.

Nvidia Users

- Everything should just work now (aka, wlroots/sway don't need patching).
- This is a known-good working config, at least at one point in time: mixins/nvidia.nix@ccd992

Development Guide

- Use `nix develop`
- `./main.nu`:
  - `./main.nu build` - builds and caches derivations that don't exist in the cache, use `nix-eval-jobs`
  - `./main.nu advance` - advances the flake inputs, runs `main build`
  - `./main.nu update` - advances the flake inputs, updates pkg revs, runs `main build`
- `build` pushes to the `nixpkgs-wayland` cachix

If for some reason the overlay isn't progressing and you want to help, just clone the repo, run `nix develop -c ./main.nu update`

Releases

No releases published

Packages

No packages published

Contributors 57

+ 43 contributors

Languages

Nix 87.6% Nushell 12.4%

Nix User Repository: User contributed nix packages [maintainer=**@Mic92**]

nur.nix-community.org/
MIT license
Code of conduct
Security policy
1.1k stars **292** forks **18** watching **8** Branches **0** Tags Activity Custom properties
Public repository

master · 8 Branches · 0 Tags · Go to file t · Go to file · Add file · Code

Nur a bot automatic update		2 hours ago
.github	Bump cachix/install-nix-action from 24 to 25	3 weeks ago
bin	move ci to flake	3 years ago
ci	nur/index: fix meta no longer beeing availa...	10 months ago
lib	lib/repoSource: don't copy local repositorie...	4 years ago
.editorconfig	add .editorconfig	6 years ago
.gitignore	move scripts to nur subcommands	6 years ago
.mergify.yml	mergify: fix merge condition	7 months ago
LICENSE	LICENSE: correct authors	6 years ago
README.md	Update README.md	2 months ago
bors.toml	bors: another try	2 years ago
default.nix	Fixed bug where some nur-combined pack...	4 years ago
flake.lock	Remove unused nixpkgs entry in flake.lock	2 years ago
flake.nix	flake.nix: export hmModules.nur	2 years ago
repos.json	add avrahambenaram repository	last week
repos.json.lock	automatic update	2 hours ago

README · Code of conduct · MIT license · Security

NUR

The Nix User Repository (NUR) is a community-driven meta repository for Nix packages. It provides access to user repositories that contain package descriptions (Nix expressions) and allows you to install packages by referencing them via attributes. In contrast to Nixpkgs, packages are built from source and **are not reviewed by any Nixpkgs member**.

The NUR was created to share new packages from the community in a faster and more decentralized way.

NUR automatically checks its list of repositories and performs evaluation checks before it propagates the updates.

Installation

First include NUR in your `packageOverrides`:

To make NUR accessible for your login user, add the following to `~/.config/nixpkgs/config.nix`:

```
{
  packageOverrides = pkgs: {
    nur = import (builtins.fetchTarball "https://github.com/nix-community/NUR/archive/master.tar.gz") {
      inherit pkgs;
    };
  };
}
```

For NixOS add the following to your `/etc/nixos/configuration.nix` Notice: If you want to use NUR in nix-env, home-manager or in nix-shell you also need NUR in `~/.config/nixpkgs/config.nix` as shown above!

```
{
  nixpkgs.config.packageOverrides = pkgs: {
    nur = import (builtins.fetchTarball "https://github.com/nix-community/NUR/archive/master.tar.gz") {
      inherit pkgs;
    };
  };
}
```

Pinning

Using `builtins.fetchTarball` without a sha256 will only cache the download for 1 hour by default, so you need internet access almost every time you build something. You can pin the version if you don't want that:

```
builtins.fetchTarball {
  # Get the revision by choosing a version from https://github.com/nix-community/NUR/commits/master
  url = "https://github.com/nix-community/NUR/archive/3a6a6f4da737da41e27922ce2cfacf68a109ebce.tar.gz";
  # Get the hash by running `nix-prefetch-url --unpack <url>` on the above url
  sha256 = "04387gzgl8y555b3lkz9aiw9xsldfg4zmzp930m62qw8zbrvrshd";
}
```

How to use

Then packages can be used or installed from the NUR namespace.

```
$ nix-shell -p nur.repos.mic92.hello-nur
nix-shell> hello
Hello, NUR!
```

or

```
$ nix-env -f '<nixpkgs>' -iA nur.repos.mic92.hello-nur
```

or

```
# configuration.nix
environment.systemPackages = with pkgs; [
  nur.repos.mic92.hello-nur
];
```

Each contributor can register their repository under a name and is responsible for its content.

NUR does not check the repository for malicious content on a regular basis and it is recommended to check the expressions before installing them.

Using the flake in NixOS

Using overlays and modules from NUR in your configuration is fairly straight forward.

In your flake.nix add `nur.nixosModules.nur` to your module list:

```
{
  inputs.nur.url = github:nix-community/NUR;

  outputs = { self, nixpkgs, nur }: {
    nixosConfigurations.myConfig = nixpkgs.lib.nixosSystem {
      # ...
      modules = [
        nur.nixosModules.nur
        # This adds a nur configuration option.
        # Use `config.nur` for packages like this:
        # ({ config, ... }: {
        #   environment.systemPackages = [ config.nur.repos.mic92.hello-nur ];
        # })
```

```
        ];
      };
    };
  }
```

You cannot use `config.nur` for importing NixOS modules from NUR as this will lead to infinite recursion errors.

Instead use:

```
{
  inputs.nur.url = "github:nix-community/NUR";
  outputs = { self, nixpkgs, nur }: rec {
    nixosConfigurations.laptop = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        { nixpkgs.overlays = [ nur.overlay ]; }
        ({ pkgs, ... }:
          let
            nur-no-pkgs = import nur {
              nurpkgs = import nixpkgs { system = "x86_64-linux"; };
            };
          in {
            imports = [ nur-no-pkgs.repos.iopq.modules.xraya  ];
            services.xraya.enable = true;
          })
        #./configuration.nix or other imports here
      ];
    };
  };
}
```

Using modules overlays or library functions in NixOS

If you intend to use modules, overlays or library functions in your NixOS configuration.nix, you need to take care not to introduce infinite recursion. Specifically, you need to import NUR like this in the modules:

```
{ pkgs, config, lib, ... }:
let
  nur-no-pkgs = import (builtins.fetchTarball "https://github.com/nix-community/NUR/archive/master.tar.gz") {};
in {

  imports = [
    nur-no-pkgs.repos.paul.modules.foo
  ];

  nixpkgs.overlays = [
    nur-no-pkgs.repos.ben.overlays.bar
  ];

}
```

Integrating with Home Manager

Integrating with Home Manager can be done by adding your modules to the `imports` attribute. You can then configure your services like usual.

```
let
  nur-no-pkgs = import (builtins.fetchTarball "https://github.com/nix-community/NUR/archive/master.tar.gz") {};
in
{
  imports = lib.attrValues nur-no-pkgs.repos.moredhel.hmModules.rawModules;

  services.unison = {
    enable = true;
    profiles = {
      org = {
        src = "/home/moredhel/org";
        dest = "/home/moredhel/org.backup";
```

```
        extraArgs = "-batch -watch -ui text -repeat 60 -fat";
      };
    };
  };
}
```

Finding packages

You can find all packages using Packages search for NUR or search our nur-combined repository, which contains all nix expressions from all users, via github.

How to add your own repository.

First, create a repository that contains a `default.nix` in its top-level directory. We also provide a repository template that contains a prepared directory structure.

DO NOT import packages for example `with import <nixpkgs> {};`. Instead take all dependency you want to import from Nixpkgs from the given `pkgs` argument. Each repository should return a set of Nix derivations:

```
{ pkgs }:
{
  hello-nur = pkgs.callPackage ./hello-nur {};
}
```

In this example `hello-nur` would be a directory containing a `default.nix`:

```
{ stdenv, fetchurl, lib }:

stdenv.mkDerivation rec {
  pname = "hello";
  version = "2.10";

  src = fetchurl {
    url = "mirror://gnu/hello/${pname}-${version}.tar.gz";
    sha256 = "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i";
  };

  postPatch = ''
    sed -i -e 's/Hello, world!/Hello, NUR!/' src/hello.c
  '';

  # fails due to patch
  doCheck = false;

  meta = with lib; {
    description = "A program that produces a familiar, friendly greeting";
    longDescription = ''
      GNU Hello is a program that prints "Hello, world!" when you run it.
      It is fully customizable.
    '';
    homepage = https://www.gnu.org/software/hello/manual/;
    changelog = "https://git.savannah.gnu.org/cgit/hello.git/plain/NEWS?h=v${version}";
    license = licenses.gpl3Plus;
    maintainers = [ maintainers.eelco ];
    platforms = platforms.all;
  };
}
```

You can use `nix-shell` or `nix-build` to build your packages:

```
$ nix-shell --arg pkgs 'import <nixpkgs> {}' -A hello-nur
nix-shell> hello
nix-shell> find $buildInputs
```

```
$ nix-build --arg pkgs 'import <nixpkgs> {}' -A hello-nur
```

For development convenience, you can also set a default value for the pkgs argument:

```
{ pkgs ? import <nixpkgs> {} }:
{
  hello-nur = pkgs.callPackage ./hello-nur {};
}
```

```
$ nix-build -A hello-nur
```

Add your own repository to the `repos.json` of NUR:

```
$ git clone --depth 1 https://github.com/nix-community/NUR
$ cd NUR
```

edit the file `repos.json` :

```
{
    "repos": {
        "mic92": {
            "url": "https://github.com/Mic92/nur-packages"
        },
        "<fill-your-repo-name>": {
            "url": "https://github.com/<your-user>/<your-repo>"
        }
    }
}
```

At the moment, each URL must point to a git repository. By running `bin/nur update` the corresponding `repos.json.lock` is updatec and the repository is tested. This will also perform an evaluation check, which must be passed for your repository. Commit the changed `repos.json` but NOT `repos.json.lock`

```
$ ./bin/nur format-manifest # ensure repos.json is sorted alphabetically
$ git add repos.json
$ git commit -m "add <your-repo-name> repository"
$ git push
```

and open a pull request towards https://github.com/nix-community/NUR.

At the moment repositories should be buildable on Nixpkgs unstable. Later we will add options to also provide branches for other Nixpkgs channels.

Use a different nix file as root expression

To use a different file instead of `default.nix` to load packages from, set the `file` option to a path relative to the repository root:

```
{
    "repos": {
        "mic92": {
            "url": "https://github.com/Mic92/nur-packages",
            "file": "subdirectory/default.nix"
        }
    }
}
```

Update NUR's lock file after updating your repository

By default, we only check for repository updates once a day with an automatic github action to update our lock file `repos.json.lock` To update NUR faster, you can use our service at https://nur-update.nix-community.org/ after you have pushed an update to your repository, e.g.:

```
curl -XPOST https://nur-update.nix-community.org/update?repo=mic92
```

Check out the github page for further details

HELP! Why are my NUR packages not updating?

With every build triggered via the URL hook, all repositories will be evaluated.Only if the evaluation does not contain errors the repository revision for the user is updated. Typical evaluation errors are:

- Using a wrong license attribute in the metadata.
- Using a builtin fetcher because it will cause access to external URLs during evaluation. Use pkgs.fetch* instead (i.e. instead of `builtins.fetchGit` use `pkgs.fetchgit`)

You can find out if your evaluation succeeded by checking the latest build job.

Local evaluation check

In your `nur-packages/` folder, run the check evaluation task

```
nix-env -f . -qa \* --meta \
  --allowed-uris https://static.rust-lang.org \
  --option restrict-eval true \
  --option allow-import-from-derivation true \
  --drv-path --show-trace \
  -I nixpkgs=$(nix-instantiate --find-file nixpkgs) \
  -I ./ \
  --json | jq -r 'values | .[].name'
```

On success, this shows a list of your packages

Git submodules

To fetch git submodules in repositories set `submodules` :

```
{
    "repos": {
        "mic92": {
            "url": "https://github.com/Mic92/nur-packages",
            "submodules": true
        }
    }
}
```

NixOS modules, overlays and library function support

It is also possible to define more than just packages. In fact any Nix expression can be used.

To make NixOS modules, overlays and library functions more discoverable, we propose to put them in their own namespace within th repository. This allows us to make them later searchable, when the indexer is ready.

Providing NixOS modules

NixOS modules should be placed in the `modules` attribute:

```
{ pkgs }: {
  modules = import ./modules;
}
```

```
# modules/default.nix
{
  example-module = ./example-module.nix;
}
```

An example can be found here. Modules should be defined as paths, not functions, to avoid conflicts if imported from multiple locations.

Providing Overlays

For overlays use the `overlays` attribute:

```
# default.nix
{
  overlays = {
    hello-overlay = import ./hello-overlay;
  };
}
```

```
# hello-overlay/default.nix
self: super: {
  hello = super.hello.overrideAttrs (old: {
    separateDebugInfo = true;
  });
}
```

Providing library functions

Put reusable nix functions that are intend for public use in the `lib` attribute:

```
{ pkgs }:
with pkgs.lib;
{
  lib = {
    hexint = x: hexvals.${toLower x};

    hexvals = listToAttrs (imap (i: c: { name = c; value = i - 1; })
      (stringToCharacters "0123456789abcdef"));
  };
}
```

Overriding repositories

You can override repositories using `repoOverrides` argument. This allows to test changes before publishing.

```
{
  packageOverrides = pkgs: {
    nur = import (builtins.fetchTarball "https://github.com/nix-community/NUR/archive/master.tar.gz") {
      inherit pkgs;
      repoOverrides = {
        mic92 = import ../nur-packages { inherit pkgs; };
        ## remote locations are also possible:
        # mic92 = import (builtins.fetchTarball "https://github.com/your-user/nur-packages/archive/master.tar.gz") {
      };
    };
  };
}
```

The repo must be a valid package repo, i.e. its root contains a `default.nix` file.

Overriding repositories with Flake

Experimental Note that flake support is still experimental and might change in future in a backwards incompatible way.

You can override repositories in two ways:

- With packageOverrides

```
{
  inputs.nur.url = "github:nix-community/NUR";
  inputs.paul.url = "path:/some_path/nur-paul"; # example: a local nur.repos.paul for development

  outputs = {self, nixpkgs, nur, paul }: {
```

```
  system = "x86_64-linux";

  nurpkgs = import nixpkgs { inherit system; };

  ...
  modules = [
       {
         nixpkgs.config.packageOverrides = pkgs: {
            nur = import nur {
              inherit pkgs nurpkgs;
              repoOverrides = { paul = import paul { inherit pkgs; }; };
            };
          };
        }
  ];
  ...
}
```

- With overlay

```
{
  modules = [
    {
      nixpkgs.overlays = [
        (final: prev: {
          nur = import nur {
            nurpkgs = prev;
            pkgs = prev;
            repoOverrides = { paul = import paul { pkgs = prev; }; };
          };
        })
      ];
    }
    ...
  ];
}
```

The **repo must contains** a `flake.nix` file to addition of `default.nix` : flake.nix example

- If you need to use NUR defined modules and to avoid infinite recursion complete `nur-no-pkgs` (from previous Flake Support section) as:

```
{
  nur-no-pkgs = import nur {
    nurpkgs = import nixpkgs { system = "x86_64-linux"; };
    repoOverrides = { paul = import paul { }; };
  };
}
```

Contribution guideline

- When adding packages to your repository make sure they build and set `meta.broken` attribute to true otherwise.
- Supply meta attributes as described in the Nixpkgs manual, so packages can be found by users.
- Keep your repositories slim - they are downloaded by users and our evaluator and needs to be hashed.
- Reuse packages from Nixpkgs when applicable, so the binary cache can be leveraged

Examples for packages that could be in NUR:

- Packages that are only interesting for a small audience
- Pre-releases
- Old versions of packages that are no longer in Nixpkgs, but needed for legacy reason (i.e. old versions of GCC/LLVM)
- Automatic generated package sets (i.e. generate packages sets from PyPi or CPAN)
- Software with opinionated patches
- Experiments

Why package sets instead of overlays?

To make it easier to review nix expression NUR makes it obvious where the package is coming from. If NUR would be an overlay malicious repositories could override existing packages. Also without coordination multiple overlays could easily introduce dependency cycles.

Contact

We have a matrix channel on #nur:nixos.org. Apart from that we also read posts on https://discourse.nixos.org.

(Back to to

Releases

No releases published

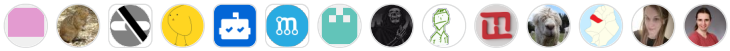
Sponsor this project

opencollective.com/**nix-community**

Packages

No packages published

Contributors 320



+ 306 contributors

Deployments 1

github-pages 6 years ago

Languages

Python 76.9% **Nix** 14.5% **Shell** 8.6%

NUR a Nix Community project

Navigation :
Documentation

Search...

DOCUMENTATION

NUR

The Nix User Repository (NUR) is a community-driven meta repository for Nix packages. It provides access to user repositories that contain package descriptions (Nix expressions) and allows you to install packages by referencing them via attributes. In contrast to Nixpkgs, packages are built from source and **are not reviewed by any Nixpkgs member**.

The NUR was created to share new packages from the community in a faster and more decentralized way.

NUR automatically checks its list of repositories and performs evaluation checks before it propagates the updates.

Installation

First include NUR in your `packageOverrides` :

To make NUR accessible for your login user, add the following to `~/.config/nixpkgs/config.nix` :

```
{
  packageOverrides = pkgs: {
    nur = import (builtins.fetchTarball "https://github.com/nix-community/NUR/
      inherit pkgs;
    };
  };
}
```

For NixOS add the following to your `/etc/nixos/configuration.nix` Notice: If you want to use NUR

NUR a Nix Community project

```
{
  nixpkgs.config.packageOverrides = pkgs: {
    nur = import (builtins.fetchTarball "https://github.com/nix-community/NUR/
      inherit pkgs;
    };
  };
}
```

Pinning

Using `builtins.fetchTarball` without a sha256 will only cache the download for 1 hour by default, so you need internet access almost every time you build something. You can pin the version if you don't want that:

```
builtins.fetchTarball {
  # Get the revision by choosing a version from https://github.com/nix-communi
  url = "https://github.com/nix-community/NUR/archive/3a6a6f4da737da41e27922ce
  # Get the hash by running `nix-prefetch-url --unpack <url>` on the above url
  sha256 = "04387gzgl8y555b3lkz9aiw9xsldfg4zmzp930m62qw8zbrvrshd";
}
```

How to use

Then packages can be used or installed from the NUR namespace.

```
$ nix-shell -p nur.repos.mic92.hello-nur
nix-shell> hello
Hello, NUR!
```

or

```
$ nix-env -f '<nixpkgs>' -iA nur.repos.mic92.hello-nur
```

or

```
# configuration.nix
environment.systemPackages = with pkgs; [
```

Each contributor can register their repository under a name and is responsible for its content.

NUR does not check the repository for malicious content on a regular basis and it is recommended to check the expressions before installing them.

Using the flake in NixOS

Using overlays and modules from NUR in your configuration is fairly straight forward.

In your flake.nix add `nur.nixosModules.nur` to your module list:

```
{
  inputs.nur.url = github:nix-community/NUR;

  outputs = { self, nixpkgs, nur }: {
    nixosConfigurations.myConfig = nixpkgs.lib.nixosSystem {
      # ...
      modules = [
        nur.nixosModules.nur
        # This adds a nur configuration option.
        # Use `config.nur` for packages like this:
        # ({ config, ... }: {
        #   environment.systemPackages = [ config.nur.repos.mic92.hello-nur ];
        # })
      ];
    };
  };
}
```

You cannot use `config.nur` for importing NixOS modules from NUR as this will lead to infinite recursion errors.

Instead use:

```
{
  inputs.nur.url = "github:nix-community/NUR";
  outputs = { self, nixpkgs, nur }: rec {
    nixosConfigurations.laptop = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
```

```
            let
              nur-no-pkgs = import nur {
                nurpkgs = import nixpkgs { system = "x86_64-linux"; };
              };
            in {
              imports = [ nur-no-pkgs.repos.iopq.modules.xraya  ];
              services.xraya.enable = true;
            })
          #./configuration.nix or other imports here
        ];
      };
    };
}
```

Using modules overlays or library functions in NixOS

If you intend to use modules, overlays or library functions in your NixOS configuration.nix, you need to take care not to introduce infinite recursion. Specifically, you need to import NUR like this in the modules:

```
{ pkgs, config, lib, ... }:
let
  nur-no-pkgs = import (builtins.fetchTarball "https://github.com/nix-communit
in {

  imports = [
    nur-no-pkgs.repos.paul.modules.foo
  ];

  nixpkgs.overlays = [
    nur-no-pkgs.repos.ben.overlays.bar
  ];

}
```

Integrating with Home Manager

Integrating with Home Manager can be done by adding your modules to the `imports` attribute. You can then configure your services like usual.

```
let
```

```
{
  imports = lib.attrValues nur-no-pkgs.repos.moredhel.hmModules.rawModules;

  services.unison = {
    enable = true;
    profiles = {
      org = {
        src = "/home/moredhel/org";
        dest = "/home/moredhel/org.backup";
        extraArgs = "-batch -watch -ui text -repeat 60 -fat";
      };
    };
  };
}
```

Finding packages

You can find all packages using Packages search for NUR or search our nur-combined repository, which contains all nix expressions from all users, via github.

How to add your own repository.

First, create a repository that contains a `default.nix` in its top-level directory. We also provide a repository template that contains a prepared directory structure.

DO NOT import packages for example `with import <nixpkgs> {};`. Instead take all dependency you want to import from Nixpkgs from the given `pkgs` argument. Each repository should return a set of Nix derivations:

```
{ pkgs }:
{
  hello-nur = pkgs.callPackage ./hello-nur {};
}
```

In this example `hello-nur` would be a directory containing a `default.nix`:

```
{ stdenv, fetchurl, lib }:

stdenv.mkDerivation rec {
```

```
  src = fetchurl {
    url = "mirror://gnu/hello/${pname}-${version}.tar.gz";
    sha256 = "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i";
  };

  postPatch = ''
    sed -i -e 's/Hello, world!/Hello, NUR!/' src/hello.c
  '';

  # fails due to patch
  doCheck = false;

  meta = with lib; {
    description = "A program that produces a familiar, friendly greeting";
    longDescription = ''
      GNU Hello is a program that prints "Hello, world!" when you run it.
      It is fully customizable.
    '';
    homepage = https://www.gnu.org/software/hello/manual/;
    changelog = "https://git.savannah.gnu.org/cgit/hello.git/plain/NEWS?h=v${v
    license = licenses.gpl3Plus;
    maintainers = [ maintainers.eelco ];
    platforms = platforms.all;
  };
}
```

You can use `nix-shell` or `nix-build` to build your packages:

```
$ nix-shell --arg pkgs 'import <nixpkgs> {}' -A hello-nur
nix-shell> hello
nix-shell> find $buildInputs
```

```
$ nix-build --arg pkgs 'import <nixpkgs> {}' -A hello-nur
```

For development convenience, you can also set a default value for the pkgs argument:

```
{
  hello-nur = pkgs.callPackage ./hello-nur {};
}
```

```
$ nix-build -A hello-nur
```

Add your own repository to the `repos.json` of NUR:

```
$ git clone --depth 1 https://github.com/nix-community/NUR
$ cd NUR
```

edit the file `repos.json`:

```
{
    "repos": {
        "mic92": {
            "url": "https://github.com/Mic92/nur-packages"
        },
        "<fill-your-repo-name>": {
            "url": "https://github.com/<your-user>/<your-repo>"
        }
    }
}
```

At the moment, each URL must point to a git repository. By running `bin/nur update` the corresponding `repos.json.lock` is updated and the repository is tested. This will also perform an evaluation check, which must be passed for your repository. Commit the changed `repos.json` but NOT `repos.json.lock`

```
$ ./bin/nur format-manifest # ensure repos.json is sorted alphabetically
$ git add repos.json
$ git commit -m "add <your-repo-name> repository"
$ git push
```

and open a pull request towards https://github.com/nix-community/NUR.

At the moment repositories should be buildable on Nixpkgs unstable. Later we will add options to also provide branches for other Nixpkgs channels.

To use a different file instead of `default.nix` to load packages from, set the `file` option to a path relative to the repository root:

```
{
    "repos": {
        "mic92": {
            "url": "https://github.com/Mic92/nur-packages",
            "file": "subdirectory/default.nix"
        }
    }
}
```

Update NUR's lock file after updating your repository

By default, we only check for repository updates once a day with an automatic github action to update our lock file `repos.json.lock` . To update NUR faster, you can use our service at https://nur-update.nix-community.org/ after you have pushed an update to your repository, e.g.:

```
curl -XPOST https://nur-update.nix-community.org/update?repo=mic92
```

Check out the github page for further details

HELP! Why are my NUR packages not updating?

With every build triggered via the URL hook, all repositories will be evaluated.Only if the evaluation does not contain errors the repository revision for the user is updated. Typical evaluation errors are:

- Using a wrong license attribute in the metadata.
- Using a builtin fetcher because it will cause access to external URLs during evaluation. Use pkgs.fetch* instead (i.e. instead of `builtins.fetchGit` use `pkgs.fetchgit`)

You can find out if your evaluation succeeded by checking the latest build job.

Local evaluation check

In your `nur-packages/` folder, run the check evaluation task

```
  --allowed-uris https://static.rust-lang.org \
  --option restrict-eval true \
  --option allow-import-from-derivation true \
  --drv-path --show-trace \
  -I nixpkgs=$(nix-instantiate --find-file nixpkgs) \
  -I ./ \
  --json | jq -r 'values | .[].name'
```

On success, this shows a list of your packages

Git submodules

To fetch git submodules in repositories set `submodules`:

```
{
    "repos": {
        "mic92": {
            "url": "https://github.com/Mic92/nur-packages",
            "submodules": true
        }
    }
}
```

NixOS modules, overlays and library function support

It is also possible to define more than just packages. In fact any Nix expression can be used.

To make NixOS modules, overlays and library functions more discoverable, we propose to put them in their own namespace within the repository. This allows us to make them later searchable, when the indexer is ready.

Providing NixOS modules

NixOS modules should be placed in the `modules` attribute:

```
{ pkgs }: {
  modules = import ./modules;
}
```

```
{
  example-module = ./example-module.nix;
}
```

An example can be found here. Modules should be defined as paths, not functions, to avoid conflicts if imported from multiple locations.

Providing Overlays

For overlays use the `overlays` attribute:

```
# default.nix
{
  overlays = {
    hello-overlay = import ./hello-overlay;
  };
}
```

```
# hello-overlay/default.nix
self: super: {
  hello = super.hello.overrideAttrs (old: {
    separateDebugInfo = true;
  });
}
```

Providing library functions

Put reusable nix functions that are intend for public use in the `lib` attribute:

```
{ pkgs }:
with pkgs.lib;
{
  lib = {
    hexint = x: hexvals.${toLower x};

    hexvals = listToAttrs (imap (i: c: { name = c; value = i - 1; })
      (stringToCharacters "0123456789abcdef"));
  };
}
```

NUR a Nix Community project

Overriding repositories

You can override repositories using `repoOverrides` argument. This allows to test changes before publishing.

```
{
  packageOverrides = pkgs: {
    nur = import (builtins.fetchTarball "https://github.com/nix-community/NUR/
      inherit pkgs;
      repoOverrides = {
        mic92 = import ../nur-packages { inherit pkgs; };
        ## remote locations are also possible:
        # mic92 = import (builtins.fetchTarball "https://github.com/your-user/
      };
    };
  };
}
```

The repo must be a valid package repo, i.e. its root contains a `default.nix` file.

Overriding repositories with Flake

Experimental Note that flake support is still experimental and might change in future in a backwards incompatible way.

You can override repositories in two ways:

- With packageOverrides

```
{
  inputs.nur.url = "github:nix-community/NUR";
  inputs.paul.url = "path:/some_path/nur-paul"; # example: a local nur.repos.p

  outputs = {self, nixpkgs, nur, paul }: {

  system = "x86_64-linux";

  nurpkgs = import nixpkgs { inherit system; };

  ...
  modules = [
```

```
            nur = import nur {
              inherit pkgs nurpkgs;
              repoOverrides = { paul = import paul { inherit pkgs; }; };
            };
          };
        }
  ];
  ...
}
```

- With overlay

```
{
  modules = [
    {
      nixpkgs.overlays = [
        (final: prev: {
          nur = import nur {
            nurpkgs = prev;
            pkgs = prev;
            repoOverrides = { paul = import paul { pkgs = prev; }; };
          };
        })
      ];
    }
    ...
  ];
}
```

The **repo must contains** a `flake.nix` file to addition of `default.nix` : flake.nix example

- If you need to use NUR defined modules and to avoid infinite recursion complete `nur-no-pkgs` (from previous Flake Support section) as:

```
{
  nur-no-pkgs = import nur {
    nurpkgs = import nixpkgs { system = "x86_64-linux"; };
    repoOverrides = { paul = import paul { }; };
  };
}
```

Contribution guideline

- When adding packages to your repository make sure they build and set `meta.broken` attribute to true otherwise.
- Supply meta attributes as described in the Nixpkgs manual, so packages can be found by users.
- Keep your repositories slim - they are downloaded by users and our evaluator and needs to be hashed.
- Reuse packages from Nixpkgs when applicable, so the binary cache can be leveraged

Examples for packages that could be in NUR:

- Packages that are only interesting for a small audience
- Pre-releases
- Old versions of packages that are no longer in Nixpkgs, but needed for legacy reason (i.e. old versions of GCC/LLVM)
- Automatic generated package sets (i.e. generate packages sets from PyPi or CPAN)
- Software with opinionated patches
- Experiments

Why package sets instead of overlays?

To make it easier to review nix expression NUR makes it obvious where the package is coming from. If NUR would be an overlay malicious repositories could override existing packages. Also without coordination multiple overlays could easily introduce dependency cycles.

Contact

Packaging existing software with Nix

Contents

One of Nix's primary use-cases is in addressing common difficulties encountered while packaging software, like managing dependencies.

In the long term, Nix helps tremendously in alleviating that stress, but when *first* packaging existing software with Nix, it's common to encounter missing dependencies preventing builds from succeeding.

In this tutorial, you'll create your first Nix derivations to package C/C++ software, taking advantage of the Nixpkgs Standard Environment (`stdenv`) which automates much of the work of building self-contained C/C++ packages.

The tutorial begins by considering `hello`, an implementation of "hello world" which only requires dependencies provided by `stdenv`.

Next, you will build more complex packages with their own dependencies, leading you to use additional derivation features.

You'll encounter and address Nix error messages, build failures, and a host of other issues, developing your iterative debugging techniques along the way.

> **Note**
>
> A *package* is an informally defined Nixpkgs concept referring to a Nix derivation representing an installation of some project. Packages have mostly standardised attributes and output layouts, allowing them to be discovered in searches and installed into environments alongside other packages.
>
> For the purposes of this tutorial, “package” means something like “result of a derivation”; this is the artifact you or others will use, as a consequence of having “packaged existing software with Nix”.

A simple project

To start, consider this skeleton derivation:

```
1 { stdenv }:
2
3 stdenv.mkDerivation { };
```

This is a function which takes an attribute set containing `stdenv`, and produces a derivation (which currently does nothing).

As you progress through this tutorial, you will update this several times, adding more details while following the general pattern.

Hello, World!

GNU Hello is an implementation of the “hello world” program, with source code accessible from the GNU Project’s FTP server.

To begin, you should add a `name` attribute to the set passed to `mkDerivation`; every derivation needs a name, and Nix will throw `error: derivation name missing` without one.

```
...
stdenv.mkDerivation {
```

```
+ name = "hello";
...
```

Next, you will download the latest version of `hello` using `fetchzip`, which takes the URI path to the download file and a SHA256 hash of its contents.

> **Note**
>
> `fetchzip` can fetch more archives than just zip files!

The hash cannot be known until after the tarball has been downloaded and unpacked, but Nix will complain if the hash supplied to `fetchzip` was incorrect, so it is common practice to supply a fake one with `lib.fakeSha256` and change the derivation definition after Nix reports the correct hash:

```nix
# hello.nix
{ lib
, stdenv
, fetchzip
}:

stdenv.mkDerivation {
  name = "hello";

  src = fetchzip {
    url = "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz";
    sha256 = lib.fakeSha256;
  };
}
```

Save this file to `hello.nix` and try to build it with `nix-build`, observing your first build failure:

```
$ nix-build hello.nix
error: cannot evaluate a function that has an argument without a value ('
       Nix attempted to evaluate a function as a top level expression; in
       this case it must have its arguments supplied either by default
       values, or passed explicitly with '--arg' or '--argstr'. See
       https://nix.dev/manual/nix/2.18/language/constructs.html#functions.

       at /home/nix-user/hello.nix:2:3:

            1| # hello.nix
            2| { lib
             |   ^
```

```
            3| , stdenv
```

Problem: the expression in `hello.nix` is a *function*, which only produces its intended output if it is passed the correct *arguments*.

A new command

`lib` is available from `nixpkgs`, which must be imported with another Nix expression in order to pass it as an argument to this derivation.

The recommended way to do this is to create a `default.nix` in the same directory as `hello.nix`, with the following contents:

```nix
# default.nix
let
  pkgs = import <nixpkgs> { };
in
{
  hello = pkgs.callPackage ./hello.nix { };
}
```

This allows you to use `nix-build -A hello` to realize the derivation in `hello.nix`, similar to the current convention used in `nixpkgs`.

> **Note**
>
> `callPackage` automatically passes attributes from `pkgs` to the given function, if they match attributes required by that function's argument attrset.
>
> In this case, `callPackage` will supply `lib`, `stdenv`, and `fetchzip` to the function defined in `hello.nix`.

Now run the `nix-build` command with the new argument:

```
$ nix-build -A hello
error:
...
       … while evaluating attribute 'src' of derivation 'hello'

         at /home/nix-user/hello.nix:9:3:
```

```
           8|
           9|   src = fetchzip {
             |   ^
          10|     url = "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz"

       error: hash mismatch in file downloaded from 'https://ftp.gnu.org/gnu
         specified: sha256:0000000000000000000000000000000000000000000000000000
         got:       sha256:0xw6cr5jgi1ir13q6apvrivwmmpr5j8vbymp0x6ll0kcv6366
```

Finding the file hash

As expected, the incorrect file hash caused an error, and Nix helpfully provided the correct one, which you can now substitute into `hello.nix` to replace `lib.fakeSha256`:

```nix
# hello.nix
{ lib
, stdenv
, fetchzip
}:

stdenv.mkDerivation {
  name = "hello";

  src = fetchzip {
    url = "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz";
    sha256 = "0xw6cr5jgi1ir13q6apvrivwmmpr5j8vbymp0x6ll0kcv6366hnn";
  };
}
```

Now run the previous command again:

```
$ nix-build -A hello
this derivation will be built:
  /nix/store/rbq37s3r76rr77c7d8x8px7z04kw2mk7-hello.drv
building '/nix/store/rbq37s3r76rr77c7d8x8px7z04kw2mk7-hello.drv'...
...
configuring
...
configure: creating ./config.status
config.status: creating Makefile
...
building
... <many more lines omitted>
```

Great news: the derivation built successfully!

The console output shows that `configure` was called, which produced a `Makefile` that was then used to build the project. It wasn't necessary to write any build instructions in this case because the `stdenv` build system is based on `autoconf`, which automatically detected the structure of the project directory.

Build result

Check your working directory for the result:

```
$ ls
default.nix hello.nix  result
```

This `result` is a symbolic link to a Nix store location containing the built binary; you can call `./result/bin/hello` to execute this program:

```
$ ./result/bin/hello
Hello, world!
```

Congratulations, you have successfully packaged your first program with Nix!

Next, you'll package another piece of software with external-to-`stdenv` dependencies that present new challenges, requiring you to make use of more `mkDerivation` features.

Something bigger

Now you will package a somewhat more complicated program, `icat`, which allows you to render images in your terminal.

To start, modify the `default.nix` from the previous section by adding a new attribute for `icat`:

```
1 # default.nix
2 let
3   pkgs = import <nixpkgs> { };
4 in
```

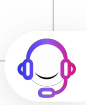
```
{
  hello = pkgs.callPackage ./hello.nix { };
  icat = pkgs.callPackage ./icat.nix { };
}
```

Now copy `hello.nix` to a new file, `icat.nix`, and update the `name` attribute in that file:

```
# icat.nix
{ lib
, stdenv
, fetchzip
}:

stdenv.mkDerivation {
  name = "icat";

  src = fetchzip {
    ...
  };
}
```

Now to download the source code. `icat`'s upstream repository is hosted on GitHub, so you should modify the previous source fetcher, this time using `fetchFromGitHub` instead of `fetchzip`, updating the argument attribute set to the function accordingly:

```
# icat.nix
{ lib
, stdenv
, fetchFromGitHub
}:

stdenv.mkDerivation {
  name = "icat";

  src = fetchFromGitHub {
    ...
  };
}
```



Fetching source from GitHub

While `fetchzip` required `url` and `sha256` arguments, more are needed for `fetchFromGitHub`.

The source is hosted on GitHub at `https://github.com/atextor/icat`, which already gives the first two arguments:

- `owner`: the name of the account controlling the repository; `owner = "atextor";`
- `repo`: the name of the repository to fetch; `repo = "icat";`

You can navigate to the project's Tags page to find a suitable `rev`, such as the git commit hash or tag (e.g. `v1.0`) corresponding to the release you want to fetch.

In this case, the latest release tag is `v0.5`.

As in the `hello` example, a hash must also be supplied.

This time, instead of using `lib.fakeSha256` and letting `nix-build` report the correct one in an error, you can fetch the correct hash in the first place with the `nix-prefetch-url` command.

You need the SHA256 hash of the *contents* of the tarball (as opposed to the hash of the tarball file itself), so you will need to pass the `--unpack` and `--type sha256` arguments too:

```
$ nix-prefetch-url --unpack https://github.com/atextor/icat/archive/refs/tag
path is '/nix/store/p8jl1jlqxcsc7ryiazbpm7c1mqb6848b-v0.5.tar.gz'
0wyy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pzr99ljs9awy8ka
```

Now you can supply the correct hash to `fetchFromGitHub`:

```nix
# icat.nix
{ lib
, stdenv
, fetchFromGitHub
}:

stdenv.mkDerivation {
  name = "icat";

  src = fetchFromGitHub {
    owner = "atextor";
    repo = "icat";
    rev = "v0.5";
    sha256 = "0wyy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pzr99ljs9awy8ka";
  };
}
```

Missing dependencies

Running `nix-build` with the new `icat` attribute, an entirely new issue is reported:

```
$ nix-build -A icat
these 2 derivations will be built:
  /nix/store/86q9x927hsyyzfr4lcqirmsbimysi6mb-source.drv
  /nix/store/l5wz9inkvkf0qhl8kpl39vpg2xfm2qpy-icat.drv
...
error: builder for '/nix/store/l5wz9inkvkf0qhl8kpl39vpg2xfm2qpy-icat.drv' fa
       last 10 log lines:
       >                  from /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp4l-g
       >                  from icat.c:31:
       > /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp4l-glibc-2.37-8-dev/includ
       >   195 | # warning "_BSD_SOURCE and _SVID_SOURCE are deprecated, use
       >       |   ^~~~~~~
       > icat.c:39:10: fatal error: Imlib2.h: No such file or directory
       >    39 | #include <Imlib2.h>
       >       |          ^~~~~~~~~~
       > compilation terminated.
       > make: *** [Makefile:16: icat.o] Error 1
       For full logs, run 'nix log /nix/store/l5wz9inkvkf0qhl8kpl39vpg2xfm2q
```

A compiler error! The `icat` source was pulled from GitHub, and Nix tried to build what it found, but compilation failed due to a missing dependency: the `imlib2` header.

If you search for `imlib2` on search.nixos.org, you'll find that `imlib2` is already in `nixpkgs`.

You can add this package to your build environment by adding `imlib2` to the set of inputs to the expression in `icat.nix`, and then adding `imlib2` to the list of `buildInputs` in `stdenv.mkDerivation`:

```
 1 # icat.nix
 2 { lib
 3 , stdenv
 4 , fetchFromGitHub
 5 , imlib2
 6 }:
 7
 8 stdenv.mkDerivation {
 9   name = "icat";
10
11   src = fetchFromGitHub {
```

```
    owner = "atextor";
    repo = "icat";
    rev = "v0.5";
    sha256 = "0wyy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pzr99ljs9awy8ka";
  };

  buildInputs = [ imlib2 ];
}
```

Run `nix-build -A icat` again and you'll encounter another error, but compilation proceeds further this time:

```
$ nix-build -A icat
this derivation will be built:
  /nix/store/bw2d4rp2k1l5rg49hds199ma2mz36x47-icat.drv
...
error: builder for '/nix/store/bw2d4rp2k1l5rg49hds199ma2mz36x47-icat.drv' fa
       last 10 log lines:
       >                  from icat.c:31:
       > /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp4l-glibc-2.37-8-dev/includ
       >   195 | # warning "_BSD_SOURCE and _SVID_SOURCE are deprecated, use
       >       |   ^~~~~~~
       > In file included from icat.c:39:
       > /nix/store/4fvrh0sjc8sbkbqda7dfsh7q0gxmnh9p-imlib2-1.11.1-dev/inclu
       >    45 | #include <X11/Xlib.h>
       >       |          ^~~~~~~~~~~~
       > compilation terminated.
       > make: *** [Makefile:16: icat.o] Error 1
       For full logs, run 'nix log /nix/store/bw2d4rp2k1l5rg49hds199ma2mz36x
```

You can see a few warnings which should be corrected in the upstream code, but the important bit for this tutorial is `fatal error: X11/Xlib.h: No such file or directory`: another dependency is missing.

> **Note**
>
> Determining from where to source a dependency is currently a somewhat-involved process: it helps to become familiar with searching the `nixpkgs` source for keywords.
>
> Consider using `nix-locate` from the `nix-index` tool to find derivations that provide what you need.

You will need the `Xlib.h` headers from the `X11` C package, the Nixpkgs derivation for

which is `libX11`, available in the `xorg` package set.

Add this to your derivation's input attribute set and to `buildInputs`:

```nix
# icat.nix
{ lib
, stdenv
, fetchFromGitHub
, imlib2
, xorg
}:

stdenv.mkDerivation {
  name = "icat";

  src = fetchFromGitHub {
    owner = "atextor";
    repo = "icat";
    rev = "v0.5";
    sha256 = "0wyy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pzr99ljs9awy8ka";
  };

  buildInputs = [ imlib2 xorg.libX11 ];
}
```

Note

Only add the top-level `xorg` derivation to the input attrset, rather than the full `xorg.libX11`, as the latter would cause a syntax error.

Because Nix is lazily-evaluated, using `xorg.libX11` means that we only include the `libX11` attribute and the derivation doesn't actually include all of `xorg` into the build context.

Run the last command again:

```
$ nix-build -A icat
this derivation will be built:
  /nix/store/x1d79ld8jxqdla5zw2b47d2sl87mf56k-icat.drv
...
error: builder for '/nix/store/x1d79ld8jxqdla5zw2b47d2sl87mf56k-icat.drv' fa
       last 10 log lines:
       >   195 | # warning "_BSD_SOURCE and _SVID_SOURCE are deprecated, use
       >       |   ^~~~~~~
       > icat.c: In function 'main':
```

```
> icat.c:319:33: warning: ignoring return value of 'write' declared w
>   319 |                                 write(tempfile, &buf, 1);
>       |                                 ^~~~~~~~~~~~~~~~~~~~~~~~
> gcc -o icat icat.o -lImlib2
> installing
> install flags: SHELL=/nix/store/8fv91097mbh5049i9rglc73dx6kjg3qk-ba
> make: *** No rule to make target 'install'.  Stop.
For full logs, run 'nix log /nix/store/x1d79ld8jxqdla5zw2b47d2sl87mf5
```

The missing dependency error is solved, but there is now another problem: `make: *** No rule to make target 'install'.  Stop.`

`installPhase`

The `stdenv` is automatically working with the `Makefile` that comes with `icat`: you can see in the console output that `configure` and `make` are executed without issue, so the `icat` binary is compiling successfully.

The failure occurs when the `stdenv` attempts to run `make install`: the `Makefile` included in the project happens to lack an `install` target, and the `README` in the `icat` repository only mentions using `make` to build the tool, leaving the installation step up to users.

To add this step to your derivation, use the **`installPhase`** attribute, which contains a list of command strings to execute to perform the installation.

Because the `make` step completes successfully, the `icat` executable is available in the build directory, and you only need to copy it from there to the output directory.

In Nix, the output directory is stored in the `$out` variable, accessible in the derivation's component scripts. Create a `bin` directory within the `$out` directory and copy the `icat` binary there:

```nix
# icat.nix
{ lib
, stdenv
, fetchFromGitHub
, imlib2
, xorg
}:

stdenv.mkDerivation {
```

```
10   name = "icat";
11
12   src = fetchFromGitHub {
13     owner = "atextor";
14     repo = "icat";
15     rev = "v0.5";
16     sha256 = "0wyy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pzr99ljs9awy8ka";
17   };
18
19   buildInputs = [ imlib2 xorg.libX11.dev ];
20
21   installPhase = ''
22     mkdir -p $out/bin
23     cp icat $out/bin
24   '';
25 }
```

Phases and hooks

Nixpkgs `stdenv.mkDerivation` derivations are separated into phases, each of which is intended to control some aspect of the build process.

You saw earlier how `stdenv.mkDerivation` expected the project's `Makefile` to have an `install` target, and failed when it didn't.

To fix this, you defined a custom `installPhase` containing instructions for copying the `icat` binary to the correct output location, in effect installing it.

Up to that point, the `stdenv.mkDerivation` automatically determined the `buildPhase` information for the `icat` package.

During derivation realisation, there are a number of shell functions ("hooks", in `nixpkgs`) which may execute in each derivation phase, which do things like set variables, source files, create directories, and so on.

These are specific to each phase, and run both before and after that phase's execution, controlling the build environment and helping to prevent environment-modifying behavior defined within packages from creating sources of nondeterminism within and between Nix derivations.

It's good practice when packaging software with Nix to include calls to these hooks in the derivation phases you define, even when you don't make direct use of them; this facilitates

easy overriding of specific parts of the derivation later, in addition to the previously-mentioned reproducibility benefits.

You should now adjust your `installPhase` to call the appropriate hooks:

```
# icat.nix
...
  installPhase = ''
    runHook preInstall
    mkdir -p $out/bin
    cp icat $out/bin
    runHook postInstall
  '';
...
```

A successful build

Running the `nix-build` command once more will finally do what you want, and more safely than before; you can `ls` in the local directory to find a `result` symlink to a location in the Nix store:

```
$ ls
default.nix hello.nix icat.nix result
```

`result/bin/icat` is the executable built previously. Success!

References

- Nixpkgs Manual - Standard Environment
- Nix Pills - `callPackage` Design Pattern

Next steps

- Dependencies in the development shell
- Automatic environment activation with direnv
- Setting up a Python development environment