Packaging existing software with Nix

Contents

One of Nix's primary use-cases is in addressing common difficulties encountered with packaging software, such as specifying and obtaining dependencies.

In the long term, Nix helps tremendously with alleviating such problems. But when *first* packaging existing software with Nix, it's common to encounter errors that seem inscrutable.

Introduction

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 involved.

What will you learn?

The tutorial begins with `hello`, an implementation of "hello world" which only requires

Skip to main content

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.

What do you need?

- Familiarity with the Unix shell and plain text editors
- You should be confident with reading the Nix language. Feel free to go back and work through the tutorial first.

How long does it take?

Going through all the steps carefully will take around 60 minutes.

Your first package

> **Note**
>
> A *package* is a loosely defined concept that refers to either a collection of files and other data, or a Nix expression representing such a collection before it comes into being. Packages in Nixpkgs have a conventional structure, allowing them to be discovered in searches and composed in environments alongside other packages.
>
> For the purposes of this tutorial, a "package" is a Nix language function that will evaluate to a derivation. It will enable you or others to produce an artifact for practical use, as a consequence of having "packaged existing software with Nix".

To start, consider this skeleton derivation:

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

Skip to main content

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

A package function

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

To begin, add a `pname` attribute to the set passed to `mkDerivation`. Every package needs a name and a version, and Nix will throw `error: derivation name missing` without.

```
stdenv.mkDerivation {
+ pname = "hello";
+ version = "2.12.1";
```

Next, you will declare a dependency on the latest version of `hello`, and instruct Nix to use `fetchzip` to download the source code archive.

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

The hash cannot be known until after the archive has been downloaded and unpacked. Nix will complain if the hash supplied to `fetchzip` is incorrect. Set the `hash` attribute to an empty string and then use the resulting error message to determine the correct hash:

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

stdenv.mkDerivation {
  pname = "hello";
  version = "2.12.1";

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

Skip to main content

Save this file to `hello.nix` and run `nix-build` to observe your first build failure:

```
$ nix-build hello.nix
error: cannot evaluate a function that has an argument without a value ('stdenv')
       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/stable/language/constructs.html#functions.

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

            2| {
            3|   stdenv,
             |   ^
            4|   fetchzip,
```

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

Building with `nix-build`

`stdenv` 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` file in the same directory as `hello.nix`, with the following contents:

```nix
# default.nix
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
{
  hello = pkgs.callPackage ./hello.nix { };
}
```

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

Skip to main content

> **Note**
>
> `callPackage` automatically passes attributes from `pkgs` to the given function, if they match attributes required by that function's argument attribute set.
>
> In this case, `callPackage` will supply `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/hello/hel
         specified: sha256:0000000000000000000000000000000000000000000000000000
         got:       sha256:0xw6cr5jgi1ir13q6apvrivwmmpr5j8vbymp0x6ll0kcv6366hnn
       error: 1 dependencies of derivation '/nix/store/8l961ay0q0ydfsgby0ngz6nmkchjqd5
```

Finding the file hash

As expected, the incorrect file hash caused an error, and Nix helpfully provided the correct one. In `hello.nix`, replace the empty string with the correct hash:

```
1 # hello.nix
2 {
3   stdenv,
4   fetchzip,
5 }:
6
7 stdenv.mkDerivation {
8   pname = "hello";
9   version = "2.12.1";
10
```

Skip to main content

```
    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 GNU 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!

Skip to main content

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.

A package with dependencies

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

Change the `default.nix` from the previous section by adding a new attribute for `icat`:

```nix
# default.nix
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
{
  hello = pkgs.callPackage ./hello.nix { };
  icat = pkgs.callPackage ./icat.nix { };
}
```

Copy `hello.nix` to a new file `icat.nix`, and update the `pname` and `version` attributes in that file:

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

stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

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

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

Skip to main content

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

stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

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

Fetching source from GitHub

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

The source URL is `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";
  ```

Navigate to the project's Tags page to find a suitable Git revision (`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 the empty string 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

Skip to main content

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

Set the correct hash for `fetchFromGitHub`:

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

stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

  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' failed with
       last 10 log lines:
       >                  from /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp4l-glibc-2.37-
       >                  from icat.c:31:
       > /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp4l-glibc-2.37-8-dev/include/features
       >   195 | # warning "_BSD_SOURCE and _SVID_SOURCE are deprecated, use _DEFAULT_
       >       |   ^~~~~~~
       > icat.c:39:10: fatal error: Imlib2.h: No such file or directory
       >    39 | #include <Imlib2.h>
       >       |          ^~~~~~~~~~
       > compilation terminated.
```

Skip to main content

```
       > make: *** [Makefile:16: icat.o] Error 1
       For full logs, run 'nix log /nix/store/l5wz9inkvkf0qhl8kpl39vpg2xfm2qpy-icat.dr
```

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.

Add this package to your build environment by adding `imlib2` to the arguments of the function in `icat.nix`. Then add the argument's value `imlib2` to the list of `buildInputs` in `stdenv.mkDerivation`:

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

stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

  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' failed with
       last 10 log lines:
       >                  from icat.c:31:
       > /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp4l-glibc-2.37-8-dev/include/features
       >   195 | # warning "_BSD_SOURCE and _SVID_SOURCE are deprecated, use _DEFAULT_
```

Skip to main content

```
> /nix/store/4fvrh0sjc8sbkbqda7dfsh7q0gxmnh9p-imlib2-1.11.1-dev/include/Imlib2.
>    45 | #include <X11/Xlib.h>
>       |          ^~~~~~~~~~~
> compilation terminated.
> make: *** [Makefile:16: icat.o] Error 1
For full logs, run 'nix log /nix/store/bw2d4rp2k1l5rg49hds199ma2mz36x47-icat.dr
```

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.

Finding packages

Determining from where to source a dependency is currently somewhat involved, because package names don't always correspond to library or program names.

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. There are multiple ways to figure this out:

`search.nixos.org`

> **Tip**
>
> The easiest way to find what you need is on search.nixos.org/packages.

Unfortunately in this case, searching for `x11` produces too many irrelevant results because X11 is ubiquitous. On the left side bar there is a list package sets, and selecting `xorg` shows something promising.

In case all else fails, it helps to become familiar with searching the Nixpkgs source code for keywords.

Local code search

To find name assignments in the source, search for `"<keyword> ="`. For example, these are the

Skip to main content

Or fetch a clone of the repository and search the code locally.

Start a shell that makes the required tools available – `git` for version control, and `rg` for code search (provided by the `ripgrep` package):

```
$ nix-shell -p git ripgrep
[nix-shell:~]$
```

The Nixpkgs repository is huge. Only clone the latest revision to avoid waiting a long time for a full clone:

```
[nix-shell:~]$ git clone https://github.com/NixOS/nixpkgs --depth 1
...
[nix-shell:~]$ cd nixpkgs/
```

To narrow down results, only search the `pkgs` subdirectory, which holds all the package recipes:

```
[nix-shell:~]$ rg "x11 =" pkgs
pkgs/tools/X11/primus/default.nix
21:  primus = if useNvidia then primusLib_ else primusLib_.override { nvidia_x11 = nul
22:  primus_i686 = if useNvidia then primusLib_i686_ else primusLib_i686_.override { n

pkgs/applications/graphics/imv/default.nix
38:    x11 = [ libGLU xorg.libxcb xorg.libX11 ];

pkgs/tools/X11/primus/lib.nix
14:    if nvidia_x11 == null then libGL

pkgs/top-level/linux-kernels.nix
573:    ati_drivers_x11 = throw "ati drivers are no longer supported by any kernel >=4
... <a lot more results>
```

Since `rg` is case sensitive by default, Add `-i` to make sure you don't miss anything:

```
[nix-shell:~]$ rg -i "libx11 =" pkgs
pkgs/applications/version-management/monotone-viz/graphviz-2.0.nix
55:    ++ lib.optional (libX11 == null) "--without-x";

pkgs/top-level/all-packages.nix
14191:    libX11 = xorg.libX11;

pkgs/servers/x11/xorg/default.nix
1119:  libX11 = callPackage ({ stdenv, pkg-config, fetchurl, xorgproto, libpthreadstub
```

Skip to main content

```
pkgs/servers/x11/xorg/overrides.nix
147:  libX11 = super.libX11.overrideAttrs (attrs: {
```

Local derivation search

To search derivations on the command line, use `nix-locate` from the `nix-index`.

Adding package sets as dependencies

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

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

stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

  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.

Skip to main content

Fixing build failures

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' failed with
       last 10 log lines:
       >   195 | # warning "_BSD_SOURCE and _SVID_SOURCE are deprecated, use _DEFAULT_
       >       |   ^~~~~~
       > icat.c: In function 'main':
       > icat.c:319:33: warning: ignoring return value of 'write' declared with attrib
       >   319 |                                 write(tempfile, &buf, 1);
       >       |                                 ^~~~~~~~~~~~~~~~~~~~~~~~
       > gcc -o icat icat.o -lImlib2
       > installing
       > install flags: SHELL=/nix/store/8fv91097mbh5049i9rglc73dx6kjg3qk-bash-5.2-p15
       > make: *** No rule to make target 'install'.  Stop.
       For full logs, run 'nix log /nix/store/x1d79ld8jxqdla5zw2b47d2sl87mf56k-icat.dr
```

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

`installPhase`

`stdenv` is automatically working with the `Makefile` that comes with `icat`. The console output shows 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. 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. It contains a list of command strings that are executed to perform the installation.

Because `make` finishes successfully, the `icat` executable is available in the build directory. You only need to copy it from there to the output directory.

Skip to main content

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

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

stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

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

  buildInputs = [ imlib2 xorg.libX11 ];

  installPhase = ''
    mkdir -p $out/bin
    cp icat $out/bin
  '';
}
```

Phases and hooks

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

Earlier you observed 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

Skip to main content

These are specific to each phase, and run both before and after that phase's execution. They modify the build environment for common operations during the build.

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. And it keeps the code tidy and makes it easier to read.

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, repeatably. Call `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

Skip to main content

Next steps

- Package parameters and overrides with callPackage
- Dependencies in the development shell
- Automatic environment activation with direnv
- Setting up a Python development environment
- Add your own new packages to Nixpkgs
  - How to contribute
  - How to get help