Best practices

Contents

URLs

The Nix language syntax supports bare URLs, so one could write `https://example.com` instead of `"https://example.com"`

RFC 45 was accepted to deprecate unquoted URLs and provides a number of arguments how this feature does more harm than good.

> **Tip**
>
> Always quote URLs.

Recursive attribute set `rec { ... }`

`rec` allows you to reference names within the same attribute set.

Example:

Expression

```
rec {
  a = 1;
```

Skip to main content

```
  b = a + 2;
}
```

Value

```
{ a = 1; b = 3; }
```

A common pitfall is to introduce a hard to debug error `infinite recursion` when shadowing a name. The simplest example for this is:

```
let a = 1; in rec { a = a; }
```

Tip

Avoid `rec`. Use `let ... in`.

Example:

Expression

```
let
  a = 1;
in {
  a = a;
  b = a + 2;
}
```

Tip

Self-reference can be achieved by explicitly naming the attribute set:

Expression

```
let
  argset = {
    a = 1;
    b = argset.a + 2;
  };
in
  argset
```

`with` scopes

It's still common to see the following expression in the wild:

Skip to main content

```
Expression
with (import <nixpkgs> {});

# ... lots of code
```

This brings all attributes of the imported expression into scope of the current expression.

There are a number of problems with that approach:

- Static analysis can't reason about the code, because it would have to actually evaluate this file to see which names are in scope.
- When more than one `with` is used, it's not clear anymore where the names are coming from.
- Scoping rules for `with` are not intuitive, see this Nix issue for details.

Tip

Do not use `with` at the top of a Nix file. Explicitly assign names in a `let` expression.

Example:

```
Expression
let
  pkgs = import <nixpkgs> {};
  inherit (pkgs) curl jq;
in

# ...
```

Smaller scopes are usually less problematic, but can still lead to surprises due to scoping rules.

Skip to main content

Tip

If you want to avoid `with` altogether, try replacing expressions of this form

```
Expression
buildInputs = with pkgs; [ curl jq ];
```

with the following:

```
Expression
buildInputs = builtins.attrValues {
  inherit (pkgs) curl jq;
};
```

`<...>` lookup paths

You will often encounter Nix language code samples that refer to `<nixpkgs>`.

`<...>` is special syntax that was introduced in 2011 to conveniently access values from the environment variable `$NIX_PATH`.

This means, the value of a lookup path depends on external system state. When using lookup paths, the same Nix expression can produce different results.

In most cases, `$NIX_PATH` is set to the latest channel when Nix is installed, and is therefore likely to differ from machine to machine.

Note

Channels are a mechanism for referencing remote Nix expressions and retrieving their latest version.

The state of a subscribed channel is external to the Nix expressions relying on it. It is not easily portable across machines. This may limit reproducibility.

For example, two developers on different machines are likely to have `<nixpkgs>` point to different revisions of the Nixpkgs repository. Builds may work for one and fail for the other, causing confusion.

Skip to main content

> **Tip**
>
> Declare dependencies explicitly using the techniques shown in Towards reproducibility: pinning Nixpkgs.
>
> Do not use lookup paths, except in minimal examples.

Some tools expect the lookup path to be set. In that case:

> **Tip**
>
> Set `$NIX_PATH` to a known value in a central location under version control.
>
> > **NixOS**
> >
> > On NixOS, `$NIX_PATH` can be set permanently with the `nix.nixPath` option.

Reproducible Nixpkgs configuration

To quickly obtain packages for demonstration, we use the following concise pattern:

```
1 import <nixpkgs> {}
```

However, even when `<nixpkgs>` is replaced as shown in Towards reproducibility: pinning Nixpkgs, the result may still not be fully reproducible. This is because, for historical reasons, the Nixpkgs top-level expression by default impurely reads from the file system to obtain configuration parameters. Systems that have the appropriate files populated may end up with different results.

It is a well-known problem that can't be resolved without breaking existing setups.

> **Tip**
>
> Explicitly set `config` and `overlays` when importing Nixpkgs:
>
> ```
> 1 import <nixpkgs> { config = {}; overlays = []; }
> ```

Skip to main content

This is what we do in our tutorials to ensure that the examples will behave exactly as expected. We skip it in minimal examples to reduce distractions.

Updating nested attribute sets

The attribute set update operator merges two attribute sets.

Example:

Expression

```
{ a = 1; b = 2; } // { b = 3; c = 4; }
```

Value

```
{ a = 1; b = 3; c = 4; }
```

However, names on the right take precedence, and updates are shallow.

Example:

Expression

```
{ a = { b = 1; }; } // { a = { c = 3; }; }
```

Value

```
{ a = { c = 3; }; }
```

Here, key `b` was completely removed, because the whole `a` value was replaced.

Tip

Use the `pkgs.lib.recursiveUpdate` Nixpkgs function:

Expression

```
let pkgs = import <nixpkgs> {}; in
pkgs.lib.recursiveUpdate { a = { b = 1; }; } { a = { c = 3;}; }
```

Value

```
{ a = { b = 1; c = 3; }; }
```

Reproducible source paths

Expression

Skip to main content

```
pkgs.stdenv.mkDerivation {
  name = "foo";
  src = ./.;
}
```

If the Nix file containing this expression is in `/home/myuser/myproject`, then the store path of `src` will be `/nix/store/<hash>-myproject`.

The problem is that now your build is no longer reproducible, as it depends on the parent directory name. That cannot be declared in the source code, and results in an impurity.

If someone builds the project in a directory with a different name, they will get a different store path for `src` and everything that depends on it. This can be the cause of needless rebuilds.

Tip

Use `builtins.path` with the `name` attribute set to something fixed.

This will derive the symbolic name of the store path from `name` instead of the working directory:

Expression

```
let pkgs = import <nixpkgs> {}; in

pkgs.stdenv.mkDerivation {
  name = "foo";
  src = builtins.path { path = ./.; name = "myproject"; };
}
```