# **Best practices**

#### **Contents**

- URLs
- Recursive attribute set **rec** { ... }
- with scopes
- <...> lookup paths
- Reproducible Nixpkgs configuration
- Updating nested attribute sets
- Reproducible source paths

#### **URLs**

The Nix language syntax supports bare URLs, so one could write <a href="https://example.com">https://example.com</a> instead of <a href="https://example.com">"https://example.com"</a>

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:

```
rec {
   a = 1:
```

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

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

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

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

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

Example:

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

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

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

### with scopes

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

```
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.

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

buildInputs = with pkgs; [ curl jq ];

with the following:

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

### lookup paths

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

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, SNIX\_PATH is set to the latest channel when Nix is installed, and is therefore likely to differ from machine to machine.



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.

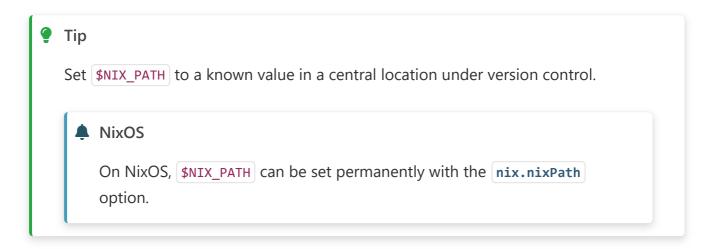


Tip

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:



## Reproducible Nixpkgs configuration

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

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

However, even when <a href="https://www.nixpkgs">nixpkgs</a> 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 = []; }
```

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
```

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

Example:

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

Value
```

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

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

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

Value

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

#### Reproducible source paths

Expression

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

If the Nix file containing this expression is in <a href="https://home/myuser/myproject">home/myuser/myproject</a>, then the store path of <a href="mailto:src">src</a> will be <a href="mailto:nix/store/<hash>-myproject">nix/store/<hash>-myproject</a>.

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.

