Introduction to Nix

NixOS Compose tutorial, ComPAS'25, Bordeaux

Millian Poquet, Quentin Guilloteau, Olivier Richard

2025-06-24

Nix's main idea

Definition

An operation is said to have a **side effect** if it has any observable effect other than its primary effect of reading the value of its arguments and returning a value to the invoker of the operation.

How to manage software **without** side effects ?

- **Immutable** file system to store packages
- **Pure** functions to build packages
- **Cryptographic hashes** for external data (source code...)

What is Nix?

Nix: package manager

- Download, store and *install* packages
- Get into tmp well-defined environments (shells)
  - `virtualenv`, but for any language
  - `docker run`, but without isolation

Nix: programming language

- λ DSL
- Define how to build packages
- Define environments (set of packages)

NixOS: Linux distribution

- Declarative system config via Nix language
- Sources: https://github.com/NixOS/nixpkgs

How to store packages?

Filesystem Hierarchy Standard

- All packages merged together
- Multiversion is tedious
- Always in the *default* environment
  - ELFs with vague deps – `require libmylib.so`
  - Libs from default paths (`/lib`, `/usr/lib`, or `ldconfig`)
  - PATH to default paths or hacked

How to store packages?

Nix Store

- Each package in its own directory + links
- Naming: hash of inputs + package name + package “version”
- Precise dependencies
  - ELFs have set `DT_RUNPATH`
  - Wrapper scripts to set `PATH`, `PYTHONPATH`…

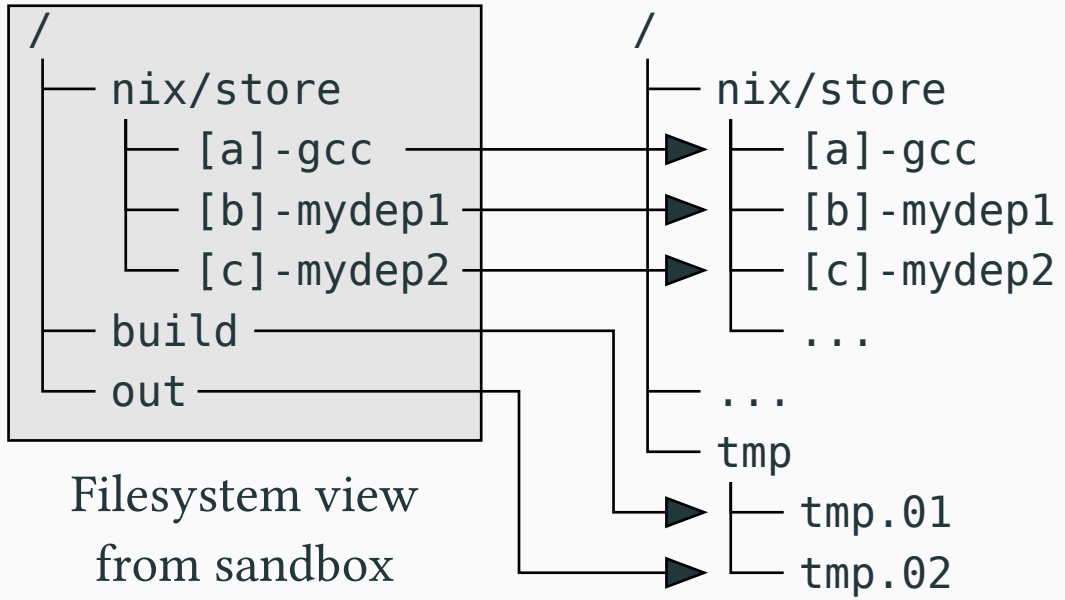
How does Nix achieve purity?

Main ideas

- Content hash used for external data (source code / bootstrap)
- Packages are built in a sandbox (Linux namespaces)
  - Controlled builder args and env vars
  - No filesystem access outside of build script / sources / deps
  - No network/ipc/... access

Workflow to build a package

1. Build all deps
2. Run build in sandbox
   - Input: build script (read-only)
   - Inputs: all deps paths (read-only)
   - Outputs: temp dirs
3. temp dirs → Store (DB transaction)



Filesystem view from sandbox

Nix λ example – explicit hello world package

```
{ stdenv }:

stdenv.mkDerivation {
  name = "hello";
  src = ./.;

  phases = [ "unpackPhase" "buildPhase" "installPhase" ];
  # default unpackPhase is used
  buildPhase = ''
    gcc -o ./hello ./hello.c
  '';
  installPhase = ''
    mkdir -p $out/bin
    mv ./hello $out/bin/
  '';
}
```

Entry points

1. `prePhases`
2. `unpackPhase`
3. `patchPhase`
4. `(pre)configurePhase`
5. `(pre)buildPhase`
6. `checkPhase`
7. `(pre)installPhase`
8. ...
9. `postPhases`

Customized by

- setting `phases`
- using another builder

Nix λ example – real package, using a build system

```
{ stdenv, fetchgit, meson, ninja, pkg-config, boost, gtest }:

stdenv.mkDerivation rec {
  pname = "intervalset";
  version = "1.2.0";
  src = fetchgit {
    url = "https://framagit.org/batsim/intervalset.git";
    rev = "v${version}";
    hash = "sha256-+mG5cPgB+wAxao/8epXWrWcyvYmzmc8Un6At+6U00qs=";
  };
  buildInputs = [ meson ninja pkg-config boost gtest ];
  # configurePhase = "meson build";
  # buildPhase = "meson compile -C build";
  # checkPhase = "meson test -C build";
  # installPhase = "meson install -C build";
}
```

Usual build layers

- Package manager: `nix`, `apt`...
- Build system: `meson`, `cmake`...
- DAG builder: `ninja`, `make`...
- Compiler: `clang`, `gcc`...

Package customization – tune λ args via `override`

```
packages = rec {
  intervalset = pkgs.callPackage ./intervalset.nix { };
  intervalset-as-debian10 = intervalset.override {
    boost = boost-167;
    meson = meson-049;
  };

  boost-176 = ...;
  boost-167 = ...;
  boost = boost-176;

  meson-058 = ...;
  meson-049 = ...;
  meson = meson-058;
};
```

Package customization – tune λ definition via `overrideAttrs`

```nix
packages = rec {
  intervalset = pkgs.callPackage ./intervalset.nix { };
  intervalset-110 = intervalset.overrideAttrs (old: rec {
    version = "1.1.0";
    src = pkgs.fetchgit {
      url = "https://framagit.org/batsim/intervalset.git";
      rev = "v${version}";
      hash = "sha256-auMx9J8h3nqNVB4qLcnxVRua4E3jy5bNc2e/REPOek4=";
    };
  });
  intervalset-local = intervalset.overrideAttrs (old: rec {
    version = "local";
    src = "/home/user/projects/intervalset";
    mesonBuildType = "debug";
  });
};
```

Nix code example – shell

```nix
{ pkgs }:

pkgs.mkShell {
  buildInputs = [
    pkgs.sysbench

    pkgs.curl

    pkgs.python3
    pkgs.python3Packages.numpy
  ];
}
```

```
$ sysbench --version
sh: sysbench: command not found
$ python --version
sh: python: command not found

$ nix-shell ...
(nix-shell) $ sysbench --version
sysbench 1.0.20
(nix-shell) $ python --version
Python 3.12.7
(nix-shell) $ python
Python 3.12.7 (main, Oct  1 2024, 02:05:46) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> numpy.__version__
'1.26.4'
>>> exit()
```

Nix code example – container

Big image

```
packages = rec {
  hello = pkgs.callPackage ./hello.nix {};
  hello-cont = pkgs.dockerTools.buildImage {
    name = "hello";
    tag = "latest";
    created = "now";
    copyToRoot = pkgs.buildEnv {
      name = "image-root";
      paths = [ hello ];
      pathsToLink = [ "/bin" ];
    };
    config.Cmd = [ "/bin/hello" ];
  };
};
```

2 layers

```
{ dockerTools, batsim, bash }:
let self = rec {
  layer-dependencies = dockerTools.buildImage {
    name = "oarteam/batsim-deps";
    tag = batsim.version;
    copyToRoot = batsim.runtimeDeps ++ [ bash ];
  };
  layer-batsim = dockerTools.buildImage {
    fromImage = layer-dependencies;
    fromImageName = layer-dependencies.name;
    fromImageTag = layer-dependencies.tag;
    tag = layer-dependencies.tag;
    name = "oarteam/batsim";
    config = {
      # ...
    };
  };
};
in
  self.layer-batsim
```

Where to write your Nix expressions?

Evaluating remote (git, some https server…) Nix expressions is as easy as local ones (files)
→ You are free to decide where to write your Nix files

Common locations

- Git repo of your software
- Git repo of your experiment
- Git repo of a set of tools you (or your team/lab/…) manage
- Nixpkgs

Nix limits

Reproducible builds only if

- Compiler is deterministic
- Build chain (build system…) is deterministic

Quote from the Meson's doc

Meson aims to support reproducible builds out of the box with zero additional work (assuming the rest of the build environment is set up for reproducibility). If you ever find a case where this is not happening, it is a bug. Please file an issue with as much information as possible and we'll get it fixed.

Contaminant approach – cannot reuse non-Nix packages

- Nixpkgs has much more packages than any other Linux distro[1]
- Writing your own package is straightforward

[1] https://repology.org/repositories/statistics
As of 2025-06-23, nixpkgs=106475, AUR=78079, debian12=34451