Nix & NixOS

A new, unique way of handling packages

- Jeffrey Samuel

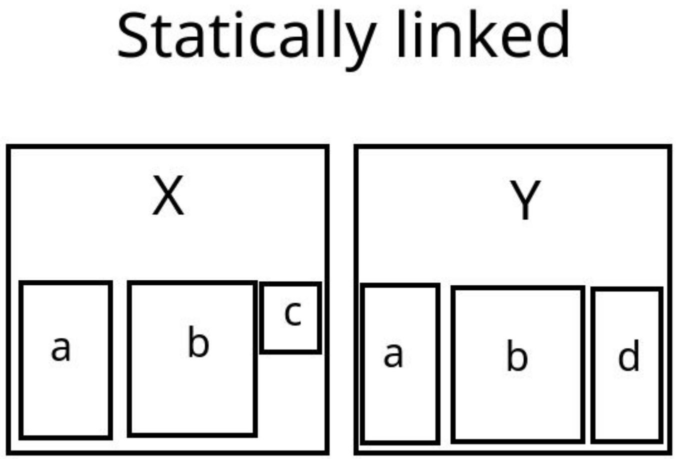
What is NixOS

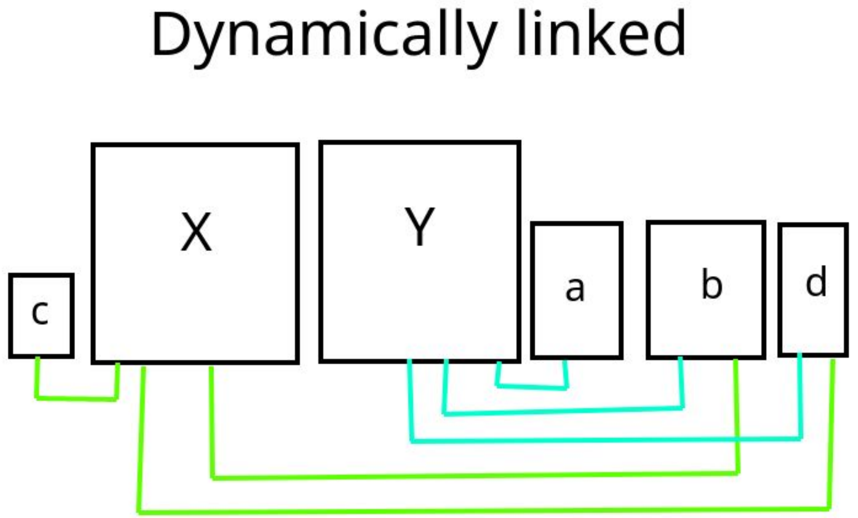
Nix is described as a package manager that uses a “purely functional” deployment model by using its own functional programming language with the same name to build packages.

It works in a vastly different way from normal package managers in an attempt to make package systems reliable, reproducible and portable.

NixOS is a linux distro built on top of the Nix package manager that extends its declarative and purely functional approach to package management to the entire operating system.

The current way of installing packages





A few pros and cons of both of these

- Dynamically linked packages provide extra security by preventing delayed security updates.
- Lesser storage space wasted.

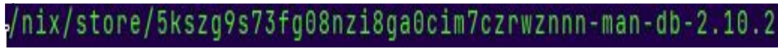
- Statically linked packages usually work the same, regardless of external packages, since their dependencies are internal.
- Dynamically linked package systems can break due to conflicting versions, AKA; Dependency Hell

How Nix tries something different (Immutability)

Nix installs packages (which are called “derivations” in nix) such a way that they never change after being built and installed - they are immutable. Instead of the usual /bin directory, it stores these in the “Nix store” under /nix/store, where each instance of a derivation gets its own unique subdirectory.

When a derivation is upgraded, the old version is not modified, rather a new unique directory is made for the newer version of the derivation. Therefore, it allows for multiple versions of a package to coexist sidestepping the entire problem with dependency hell.

An example for the directory for man-db on nix



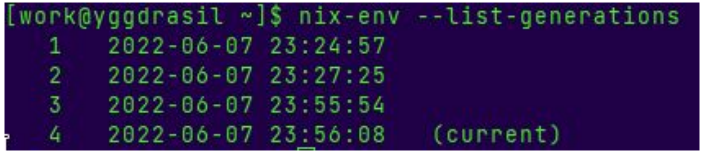
```
/nix/store/5kszg9s73fg08nzi8ga0cim7czrwznnn-man-db-2.10.2
```

How Nix tries something different (Rollbacks)

When any changes are made to the package system, Nix makes a new “generation” that keeps track of all the packages that are in scope, their dependencies and the overall state. This allows for quick and easy roll-backs, since the packages that are deleted still exist, but are just removed from the newest generation.

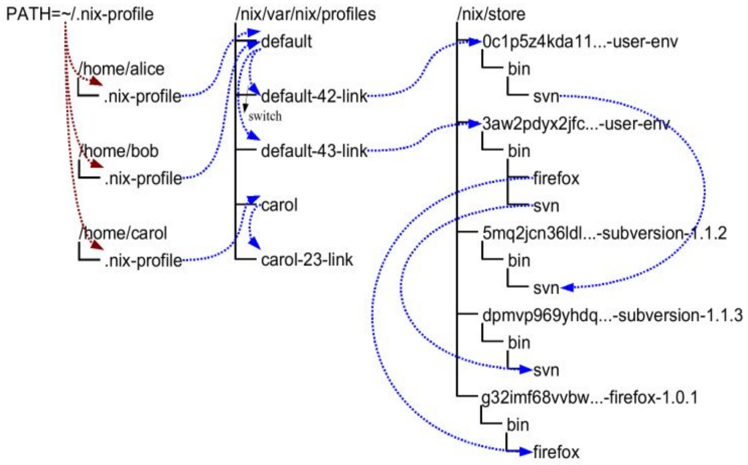
These generations actually point to derivations called “user environments” that have dependencies on all the specific derivations that make up that particular generation.

An example for the generations of a system



```
[work@yggdrasil ~]$ nix-env --list-generations
   1   2022-06-07 23:24:57
   2   2022-06-07 23:27:25
   3   2022-06-07 23:55:54
   4   2022-06-07 23:56:08   (current)
```

An example layout for generations and user-env



How Nix tries something different (Garbage collection)

Given the fact that Nix derivations are immutable, a lot of extra disk space is used. This is handled by Nix's garbage collector, which functions similarly to a garbage collector in programming languages by removing unneeded derivations from the system.

It does this by preserving all derivations that are depended on by any generation (that hasn't been deleted) or profiles present and removing any derivations that are left out. This ensures that garbage collection will never remove any package necessary either by the current generation, or one that could be rolled back to and at the same time will remove any package that won't be used by the system.

Nix command cheat sheet

Operations	APT (Ubuntu)	Nix	Pacman (Arch)
Install a package	`apt install neovim`	`nix-env -i neovim`	`pacman -S neovim`

Upgrade packages	`apt update`
	`& apt upgrade`
		`nix-channel --update`
		`& nix-env -u`
			`pacman -Syu`

Query a package	`apt search neovim`	`nix-env -qaP neovim`	`pacman -Ss neovim`
Uninstall a package	`apt autoremove neovim`	`nix-env --uninstall neovim`	`pacman -Rs neovim`
Clear cache/garbage	`apt autoclean`	`nix-collect-garbage -d`	`pacman -Qdtq | pacman -Rs -`

Nix expression language

Both the Nix package manager and NixOS heavily rely on instructions and configurations written in the Nix expression language to function.

The Nix expression language follows the purely functional programming paradigm. In the purely functional programming paradigm, the program is made up of only deterministic, mathematical functions that always return the same result and do not cause, and are unaffected by, any external factors or “side-effects”. This includes modifying any mutable state of the program such as variables.

The Nix expression language follows this paradigm as it isn’t used to execute arbitrary actions on the system to accomplish its task, but rather to define a set of deterministic rules for the state of the system to follow. This is done to ensure that on any 2 separate machines using Nix and the same Nix expression, the outcome will be the same. It is not a complete language but is rather meant solely to help the functioning of Nix and NixOS

A short look at Nix expression syntax

There are no statements, just expressions.

The data types present are integer, floating point, string, path, boolean and null.
It also includes the composite types of lists and attribute sets.
Functions are also considered a data type since they always return a value.

If-else expressions are also present.

Functions are of the lambda function format, where an input maps to a specified output

There are more keywords included to make programming for Nix easier

Attribute set syntax

```
{ <key1> = <value1>;
  <key2> = <value2>; }
```

Attribute set example

```
{ a = 27; b = “foo”; }
```

Function syntax

```
<input>: <output expression>
```

An example that squares the input

```
x: x*2
```

Making a Nix derivation

To make a Nix derivation we need 3 things:

- A Nix Expression to describe the derivation we desire
- A builder to execute the steps needed to build the source files (usually a bash script)
- The source files of the program

Here, the expression will describe everything Nix needs to know about the derivation we want to build, including dependencies, environment variables, the source, the builder and the name. From this information, Nix will build the derivation in a reproducible way so that using it will always give you the exact same output derivation.

Making a Nix derivation (The expression)

Here is a sample expression for a program that will print out some text (`hello-there.nix`):

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

derivation {
  name = "hello-there";

  builder = "${bash}/bin/bash";
  args = [ ./builder.sh ];
  inherit gcc coreutils;
  src = ./hello.c;

  system = builtins.currentSystem;
}
```

The builder script (`builder.sh`):

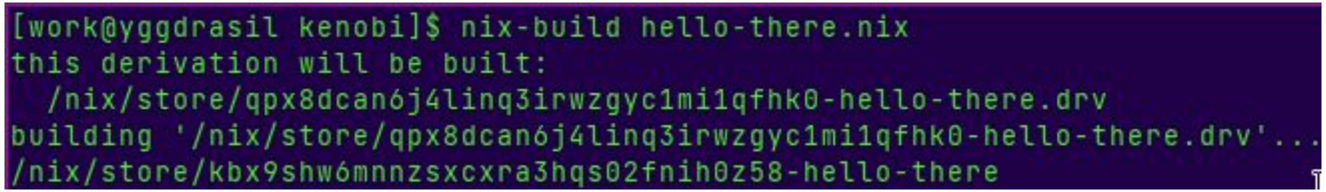
```
export PATH="$coreutils/bin:$gcc/bin"
mkdir $out
gcc -o $out/hello-there $src
```

The source code (`hello.c`):

```
#include <stdio.h>

int main(){
     printf("General Kenobi!\n");
     return 0;
}
```

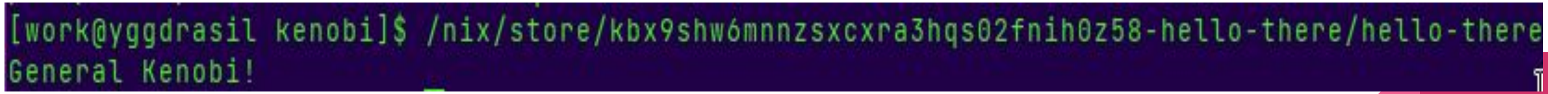
Building the expression into a derivation

Now that we have defined everything Nix needs to build the derivation, we run `nix-build` on the .nix file. Nix will convert that into an intermediate .drv file and then build the derivation into its unique directory. We have now completely made our first Nix derivation.



```
[work@yggdrasil kenobi]$ nix-build hello-there.nix
this derivation will be built:
  /nix/store/qpx8dcan6j4linq3irwzgyc1mi1qfhk0-hello-there.drv
building '/nix/store/qpx8dcan6j4linq3irwzgyc1mi1qfhk0-hello-there.drv'...
/nix/store/kbx9shw6mnnzsxcxra3hqs02fnih0z58-hello-there
```

And as we can see, we can now run our derivation! \o/



```
[work@yggdrasil kenobi]$ /nix/store/kbx9shw6mnnzsxcxra3hqs02fnih0z58-hello-there/hello-there
General Kenobi!
```

This was an extremely simple case, but the same basic principles apply to larger, real world examples

A look behind the scenes

The Intermediate .drv file generated

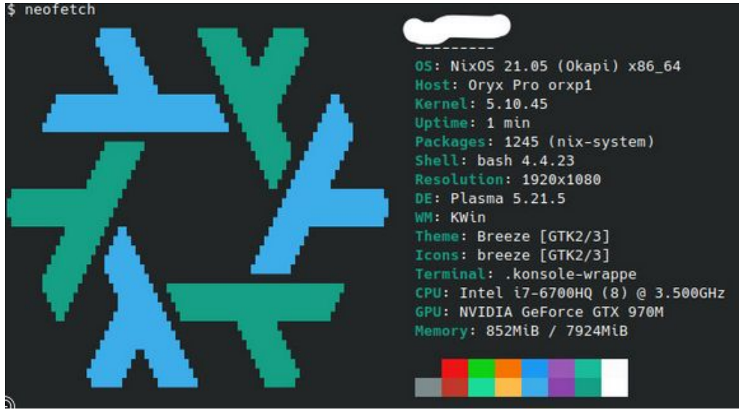
```
Derive([("out","/nix/store/kbx9shw6mnnzsxcxra3hqs02fnih0z58-hello-there","","")],
      [("/nix/store/c9fcx1y9w0pw9gj5kbm4ajxvanq82wwd-gcc-wrapper-11.3.0.drv",["out"]),
      ("/nix/store/d4rxzr8pxdhr8sa22170ribmz22d2hdj-bash-5.1-p16.drv",["out"]),
      ("/nix/store/iy3knhnlsmyfm9k3zgm9v6k8kdggj066-coreutils-9.1.drv",["out"])],

      ["/nix/store/1yk2gxg5fw97134kan7h9n62a5jh2yp0-hello.c","/nix/store/y7ihp6f51v2p08g
      7xjj180pdvbgrgdfn-builder.sh"],
      "x86_64-linux",
      "/nix/store/zwjm0gln1vk7x1akpyz0yxjsd1yc46gi-bash-5.1-p16/bin/bash",
      ["/nix/store/y7ihp6f51v2p08g7xjj180pdvbgrgdfn-builder.sh"],

      [("builder","/nix/store/zwjm0gln1vk7x1akpyz0yxjsd1yc46gi-bash-5.1-p16/bin/bash"),
      ("coreutils","/nix/store/y72i4llqf5zxvdp6b3j7ysixpdrid7qr-coreutils-9.1"),
      ("gcc","/nix/store/61zfi5pmhb0d91422f186x26v7b52y5k-gcc-wrapper-11.3.0"),
      ("name","hello-there"),
      ("out","/nix/store/kbx9shw6mnnzsxcxra3hqs02fnih0z58-hello-there"),
      ("src","/nix/store/1yk2gxg5fw97134kan7h9n62a5jh2yp0-hello.c"),
      ("system","x86_64-linux")
      ]
  )
```

NixOS

NixOS is linux distro built upon the Nix package manager and applies the guiding principles of the package manager to the entire system.

It uses a “declarative configuration” which means that the entire state of the operating system, from the packages installed to the configuration of various services, is declared beforehand in a configuration file.

This configuration file gives a complete and precise definition of the state of the operating system, meaning that it is the only factor determining the final state of the OS and will therefore produce the exact same resulting OS when deployed on any other machine.



```
$ neofetch
[redacted]
----------
OS: NixOS 21.05 (Okapi) x86_64
Host: Oryx Pro orxp1
Kernel: 5.10.45
Uptime: 1 min
Packages: 1245 (nix-system)
Shell: bash 4.4.23
Resolution: 1920x1080
DE: Plasma 5.21.5
WM: KWin
Theme: Breeze [GTK2/3]
Icons: breeze [GTK2/3]
Terminal: .konsole-wrappe
CPU: Intel i7-6700HQ (8) @ 3.500GHz
GPU: NVIDIA GeForce GTX 970M
Memory: 852MiB / 7924MiB
```

NixOS

NixOS is built from the ground up to take advantage of the Nix package manager and to complement it, due to which it also includes:

- Rolling Back to previous state of the system. Everything is preserved (excluding user data), including configuration files, packages and dependencies.
- It allows multiple users to install packages without root access and without affecting other users' packages. It is smart enough to allow the sharing of identical packages, but will still prevent one user from modifying another user's packages (for example, to inject a trojan)
- Common UNIX directories like /bin and /lib don't exist, but rather everything is located in the /nix/store. This ensures that any breakage will not permanently affect the system
- It has Atomic upgrades, which means that the system is never in a state where it is partially upgraded. Therefore, upgrades become totally safe even in the case of something like a power cut

A (very) minimal example NixOS configuration file

This is a very minimal example of a nixOS config file that will contain an ssh server that will boot on start, with the operating system installed on the device /dev/sda

```
{ config, pkgs, ... }:

{
  imports = [
    # Include the results of the hardware scan.
    ./hardware-configuration.nix
  ];

  boot.loader.systemd-boot.enable = true;

  fileSystems."/".device = "/dev/sda/nixos";

  services.sshd.enable = true;
}
```

Who is this for?

The discussed traits of Nix and NixOS makes it unlike any other operating system that exists. But it isn’t just a proof of concept OS, but rather has very real applications:

- Web servers and infrastructure that require minimal downtime, reliability and reproducibility
- Developers working with multiple different environments or with tools that require such complex environments.
- Individuals who want reliability first and foremost, and are willing to put some effort into configuring such a system.

Go on!
Try out Nix or NixOS!

https://nixos.org/download.html

Read The Manual

https://nixos.org/manual/nix/stable/introduction.html

Additional sources used

The Nix Pills series: https://nixos.org/guides/nix-pills/index.html

The Learn page: https://nixos.org/learn.html#learn-guides

The NixOS manual: https://nixos.org/manual/nixos/stable/