Nix & NixOS 101

nixcademy.com

Section: Nix & NixOS 101 Training

About All Y'all 💁

About the Lecturer 👩‍🏫

The Nix & NixOS 101 Class 🏫

Trainer notes

About All Y'all 🙋

Trainer notes

Hand-raising time ✋

Have you ever?

Trainer notes

Hand-raising time ✋

Have you ever?

- written a computer program

Trainer notes

Hand-raising time ✋

NIXCADEMY EXPERT EDUCATION

Have you ever?

- written a computer program
- written in a functional programming language (Common Lisp, Elixir, Haskell, OCaml, Scheme)

Trainer notes

Hand-raising time ✋

NIXCADEMY EXPERT EDUCATION

Have you ever?

- written a computer program
- written in a functional programming language (Common Lisp, Elixir, Haskell, OCaml, Scheme)
- used Linux

Trainer notes

Hand-raising time ✋

Have you ever?

- written a computer program
- written in a functional programming language (Common Lisp, Elixir, Haskell, OCaml, Scheme)
- used Linux
- installed Linux

Trainer notes

Hand-raising time ✋

NIXCADEMY EXPERT EDUCATION

Have you ever?

- written a computer program
- written in a functional programming language (Common Lisp, Elixir, Haskell, OCaml, Scheme)
- used Linux
- installed Linux
- used Nix

Trainer notes

Hand-raising time ✋

NIXCADEMY EXPERT EDUCATION

Have you ever?

- written a computer program
- written in a functional programming language (Common Lisp, Elixir, Haskell, OCaml, Scheme)
- used Linux
- installed Linux
- used Nix
- used NixOS

Trainer notes

Your Starting Point and Expectations

NIXCADEMY EXPERT EDUCATION

What technology stacks and languages are you **experienced** with already?

What are your **feelings** about Nix(OS)?

What are your **expectations** from this training?

Trainer notes

Don't just read these texts for the participants loudly.

- Let everyone introduce themselves in round robin fashion
- make notes somewhere and drop them as a comment in the gitlab issue of this week's training.
- paint a seat plan to memorize the names of the participants faster

The main goals of this introduction are:

- Nix newbies realize that they are not the only ones
- You find out how much Nix as a technology needs to be sold in this class.
- You sit down after monday class, go through the notes and plan which topics and examples to emphasize and which ones to reduce in exposure to make the week as relevant as possible for the participants.

About the Lecturer 👨‍🏫

Trainer notes

🎨 Art Degree, Software Engineer

Nix/NixOS user & Nixpkgs contributer since 2020

🍵 Interests/Experience

Free software, functional programming, front-end dev, blogging, □□□ + ลาว, cooking, gridiron football

Professional software development since 2010 (internship → corporation → ad firm → startups → remote → contracting)

Advertising • Cryptocurrencies • Data processing • Healthcare • Real estate • User media management

Trainer notes

Nixpkgs: H2O + Movim NixOS modules, other XMPP

Language/linguistics in general

(Programming language logos are current interests more than expertise which is how they will be presented)

Art
↓
Interactive media
↓
Web dev / front-end → ECMAScript
↓
FP: Elm / PureScript / etc.
↓
Nix / NixOS
↑
Arch
↑
Debian

Trainer notes

Interactive media: Flash + ActionScript → HTML5 + jQuery

FP: Ramda, general learning

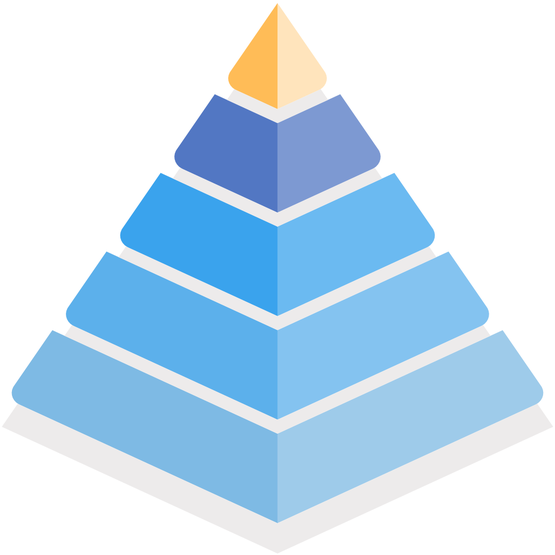
Debian: introduced to in college

The Nix & NixOS 101 Class 🏫

Trainer notes

Class Goals

NIXCADEMY EXPERT EDUCATION



05 **Full Control over Nix**
You master deadlines by working with and not against nix.

04 **Package Variation**
Overrides, Overlays, Package variants

03 **Sandbox & Packaging**
Understanding how Nix makes builds reproducible

02 **Nix Language**
Language Syntax, brief functional programming intro

01 **Nix Basics**
Basic Commands & Package Management

Trainer notes

This slide exists to explain that and why we are going to talk about many abstract things this week. Some participants might be like “but i only want to package python, why do functional programming exercises or overlays??”. Our mission is to create self-dependent proficient Nix users that have no problems creating big projects and debug complex Nix problems.

We have multiple “meta” slides that explain the Nix journey that we take this week, which might feel redundant for first-time teachers. Find a rhythm.

Class Goals: Nix

NIXCADEMY EXPERT EDUCATION

User Perspective:

- Install, Modify, & Use Packages
- Profit from the Biggest and Freshest Package Repository
- … Without Tainting the System's State

Developer Perspective:

- Define Declarative Development Environments
- Create Reproducibly Buildable Packages
- Manage, Manipulate & Pin Dependencies
- Separate and Manage Patch Sets
- Extend Nixpkgs with Complex Overlays
- Prepare Projects and Nix Expressions for Cross-Compilation

Trainer notes

- "Modify" from a user perspective is the ability to confidentally patch, upgrade/downgrade… this also is in reference to 'free' software
- Nix developers generally create user experiences.
- One project is the consumer of another project.
- We look at Nix and NixOS from upstream and downstream perspectives.
- The user perspective is created from the developer perspective, so that's why we learn these things.

Class Goals: NixOS

NIXCADEMY EXPERT EDUCATION

User Perspective:

- Install & Configure NixOS on Laptops, Servers, Cloud
- Define Containers, VMs, Custom Installer Images
- Seamlessly Update & Roll Back Systems
- Playful Evolution of Complex System Configs

Developer Perspective:

- Create NixOS Modules for Custom Services
- Simplify System Image Generation & Deployment
- Define Declarative Integration Tests

Trainer notes

What's Not Included in *This* Class

NIXCADEMY EXPERT EDUCATION

- Dependency Management for *All* the Languages
- FHS UserEnvironments
- macOS Specialties
- AppImage, SNAP, systemd Portable Services, etc.
- nix ops, krops, etc.
- Setting up Binary Caches and Remote Builders

Trainer notes

FHS is Filesystem Hierarchy Standard (/usr /bin /dev /etc /proc)

As this class is heavily foundation based, we can't look at **everything** in detail. Participants will find these topics easy to understand when they revisit the docs after the training.

We can however talk about such topics if we went through the class content in time and end up having time left on friday.

Exercise Style: Maximum Effectivity

Don't rush. 🏃
We have time. ⏳
It's not a race. 🏎️

It's also an exercise in how to read the manuals. 📚

You'll get help. 🚸
We're not in school. 🏫
You can work together. 🤝

How to draw an owl

Fig 1. Draw two circles

Fig 2. Draw the rest of the damn owl

Trainer notes

- There is no prize for finishing first
- There are no grades
- There isn't value in keeping information from your comrades; help each other
- There is nothing wrong with asking questions; feel free to question **me**

Many folks fall back into "school mode":

- Don't say anything wrong
- You have to work alone
- Don't question anything

Trainings work best when there is no hierarchy in the room and everyone feels psychologically safe to ask anything. Remind folks that the class is **for them** and to **enable them**.

The owl thingy is about the exercises: Real life projects are super complex and unwieldy, so they are not good for exercising. Exercises are limited in scope, quick to oversee and solve. There might be quite a gap between the complexity of an exercise and a real project. Still, the exercises are designed to model real world challenges. If folks don't see that, encourage them to ask what aspect of real life an exercise tries to model. If folks don't see that - why would they engage in the exercise?

Group Learning - Attention spans

NIXCADEMY EXPERT EDUCATION

THEORY

PRACTICE

Trainer notes

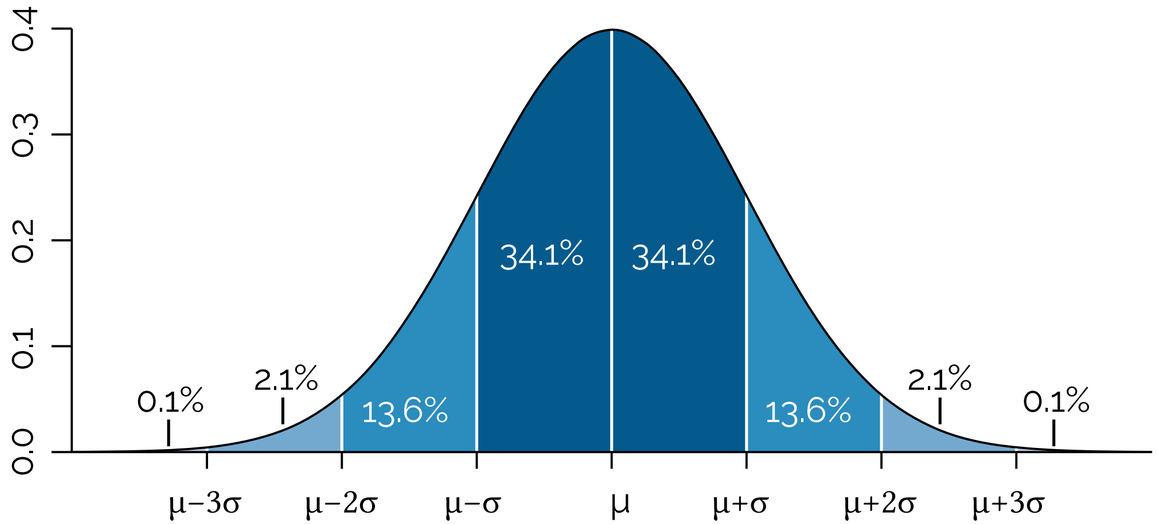
This is basically a training from the trainer's POV. Folks randomly start and stop paying attention or are distracted from training-related thoughts or the team's slack chat.

Regarding external distractions: Ask if management gave them the allowance to really focus on the training. If not, we should talk back with management to have someone with authority to tell the participants that they shall pay attention to the training and nothing else.

It's also perfectly normal that the trainer has to explain things multiple times due to attention-span-randomness.

However, the trainer must also moderate folks who distract the whole class with unrelated or too advanced discussions. Friday overflow time is for the advanced discussions.

Group Learning - Inclusivity



Trainer notes

In general, participants should be grouped by similar experience and technology stacks. This way the trainer can maximize the relevance for the team.

However, in reality there is always a partition of folks who feel bored and a partition of folks who feel overwhelmed. The best the teacher can do is make sure to have some balance here and also ask for patience as this is quite the external factor that can't be fixed during the training.

Times and Breaks ⏰

Let's plan this week's days like this:

🕘	9:00	9 AM	Begin of training
🕛	12:00	12 PM	Lunch 🍝
🕐	13:00	1 PM	Resumption of training
🕔	17:00	5 PM	End of the day 🌮 🍕 & 🧋 🍺

☕ & 🚽 Pause: 10 Minutes every ~1.5h

Trainer notes

Which time is best for lunch break to avoid long queues?

Ask folks to go to the restroom or get drinks during the breaks to preserve the atmosphere. Prefer to have more breaks if folks can't concentrate too long.

Technical Prerequisites 💻

Nix Part:

- Any Linux/macOS Laptop 💻 or VM with `sudo` rights and internet
- Windows WSL might work
- Please *don't* use Docker containers

NixOS Part:

- In this part, we also need KVM
- Nested Virtualization might not work satisfactorily

Trainer notes

Monday till Wednesday practically any Nix setup will do. From thursday on we need linux Machines with KVM.

When you restart Docker, the cache is gone.

It is very important that folks learn how a "good" Nix setup feels like and how fast it is. For that reason it is very important that *everyone* has access to the training machine.

Section: Nix

About Nix

Using Nix 🛠️

Nix Language 🈵

Packaging Basics 📦

Pinning 📌

Project Patterns 🎨

Advanced Packaging Topics 🚀

Cross compilation 🔀

Overlays 🔁

Flakes ❄️

Trainer notes

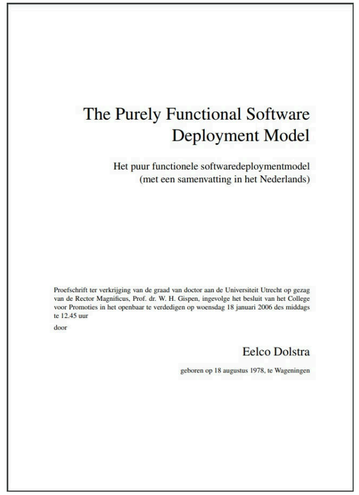
About Nix

Trainer notes

Nix: A PhD Thesis that Escalated 👩‍🎓

Nix started in 2003 as an academic project at Universiteit Utrecht (EU, Netherlands):

"This thesis is about getting computer programs from one machine to another—and having them still work when they get there."



The Purely Functional Software Deployment Model

Het puur functionele softwaredeploymentmodel
(met een samenvatting in het Nederlands)

Proefschrift ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de Rector Magnificus, Prof. dr. W. H. Gispen, ingevolge het besluit van het College voor Promoties in het openbaar te verdedigen op woensdag 18 januari 2006 des middags te 12.45 uur

door

Eelco Dolstra

geboren op 18 augustus 1978, te Wageningen

Eelco Dolstra's original PhD thesis: https://edolstra.github.io/pubs/phd-thesis.pdf

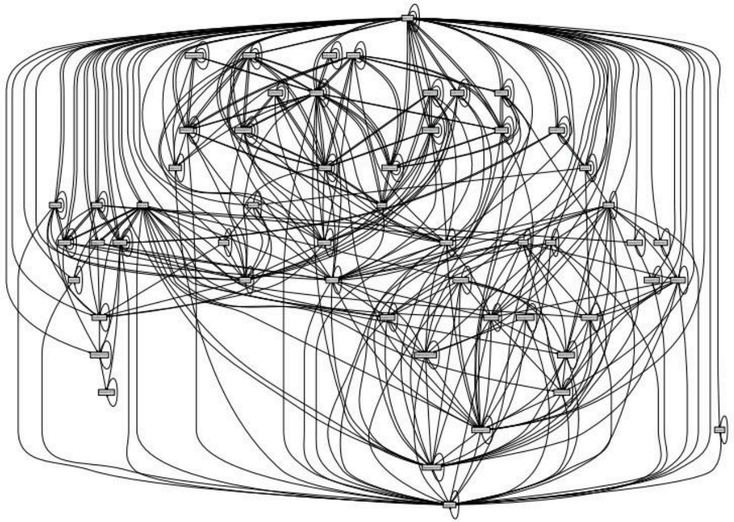
Trainer notes

Suggest to read the introductory section of the original thesis. Teacher should have done this anyway and tell participants what the actual idea behind Nix was. There are also some interesting noteworthy things that are discussed/mentioned in the intro section.

The project is also more than 20 years old. Emphasize this for everyone who might not be convinced that this is quite the mature technology.

Also mention that "Nix" is Dutch and informal German for "nothing" and that Nix is about building very complex software reproducibly starting at "nothing".

An Epic Quest: Controlling the Dependency Hell



Source:
Nix PhD Thesis

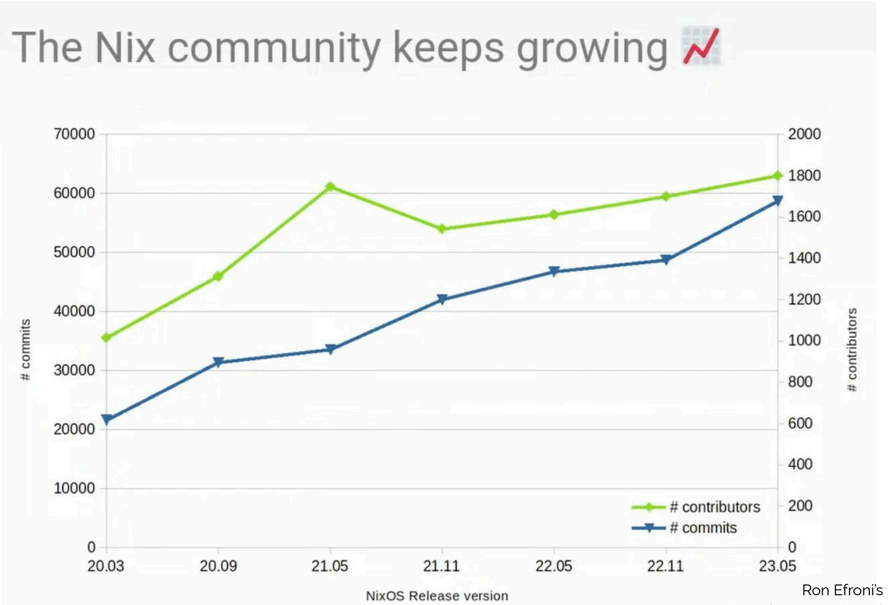
Figure 1.5.: The *dependency hell*: the runtime dependency graph of Mozilla Firefox

Trainer notes

This is the dep graph of Firefox from the PhD thesis. Discuss how messy it would be to build this with e.g. Docker containers. One big container? Many small ones? Which container to run in which order? Will this really be reproducible? Will it run in parallel? Will it cache? How easy to update?

Participants should feel this is either too difficult or too much work after your questions/discussions about this.

Nixpkgs Commits and Contributors Growth



Ron Efroni's NixCon Berlin 2024
Opening Keynote - Nix State of the Union
https://www.youtube.com/watch?v=bAZkHpSWhal

Trainer notes

Highlight that the number of commits is growing faster than the number of contributors. Mention that 10% of contributors do 90% of the work (this is not acurate but you get the idea).

Nixpkgs reaches an enormous "packages per maintainer" ratio compared to other distros. And this is one huge reason why it is so interesting for companies who need to maintain complex software under a lot of market pressure.

cache.nixos.org - Traffic

Eelco Dolstra's NixCon Paris 2022 Opening Keynote - Nix State of the Union
https://www.youtube.com/watch?v=s2fkgkNS5vk

Trainer notes

There was no graph description in the original talk. This slide is just to show growth over the years.

cache.nixos.org - Cache Size

- **468.5 TiB** Infrequent Access, **101.3 TiB** Standard
  (Sept 2023: **346.5 TiB** Infrequent Access, **114.1 TiB** Standard)
- **892M** objects
  (Sept 2023: **717M** objects)
- **2.27 PB** traffic served by Fastly in the last month

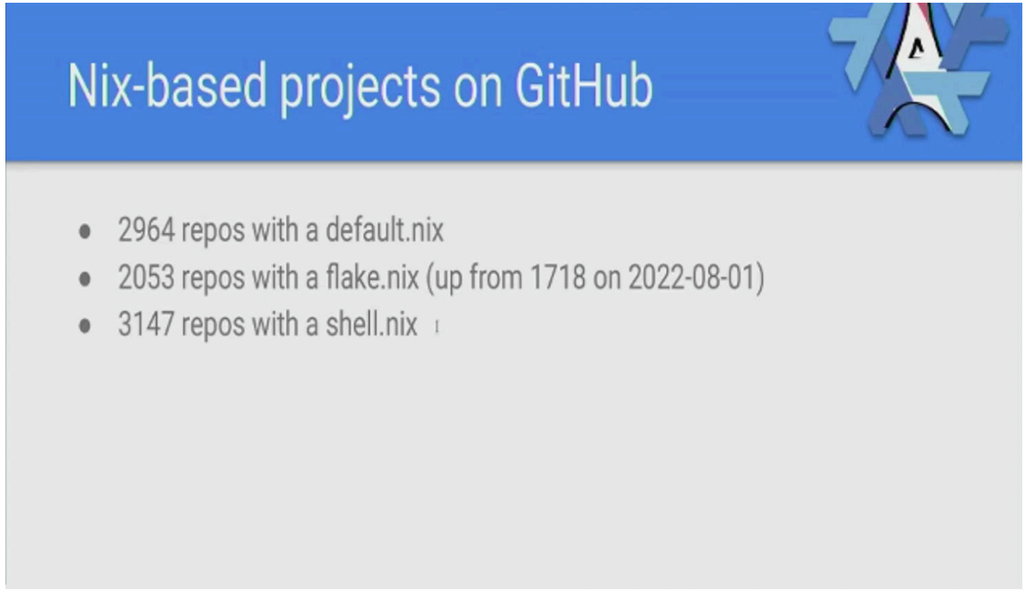
Ron Efroni's NixCon Berlin 2024 Opening Keynote - Nix State of the Union
https://www.youtube.com/watch?v=bAZkHpSWnal

Trainer notes

The cache is huge.

Note that it is not really *needed* to build all packages.

However, it also caches many Git commits and downloads that don't exist any longer, so it preserves a lot of old software so that we can still run it!

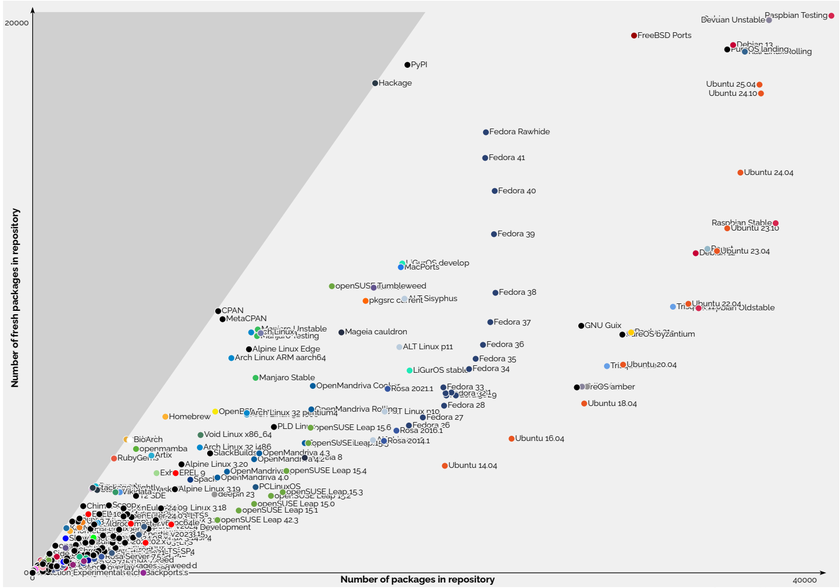


Eelco Dolstra's NixCon Paris 2022 Opening Keynote - Nix State of the Union
https://www.youtube.com/watch?v=s2fkgkN55vk

Trainer notes

Takeaway here is that for every problem you might ever encounter, the probability that you find someone's solution to it on GitHub is pretty high. This is in practice very helpful.

Nixpkgs: The Biggest Package Repository (1/2)



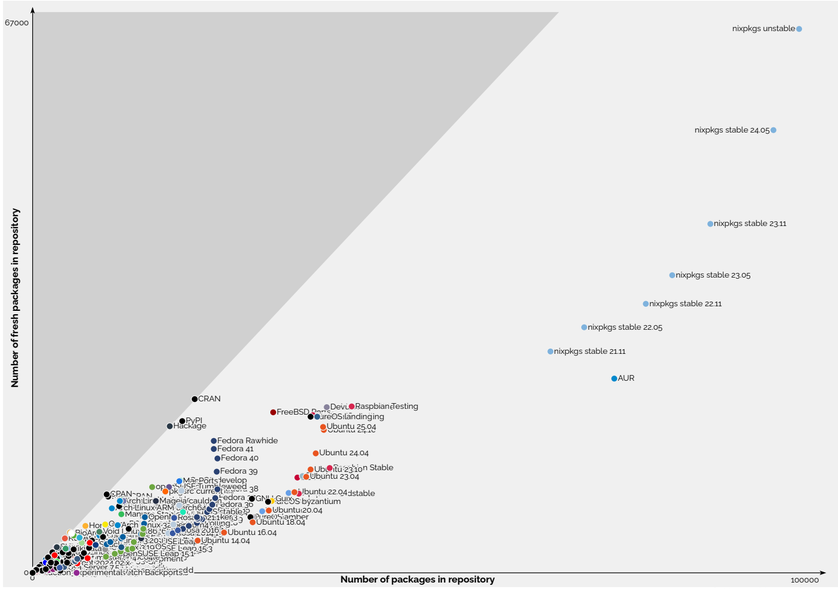
https://repology.org/repositories/graphs
as of 2024-11

Trainer notes

This is an old trick for everyone who knows this already.

Explain the axes and then ask everyone where they would expect Nixpkgs. And then, switch to the next slides which shows…

Nixpkgs: The Biggest Package Repository (2/2)



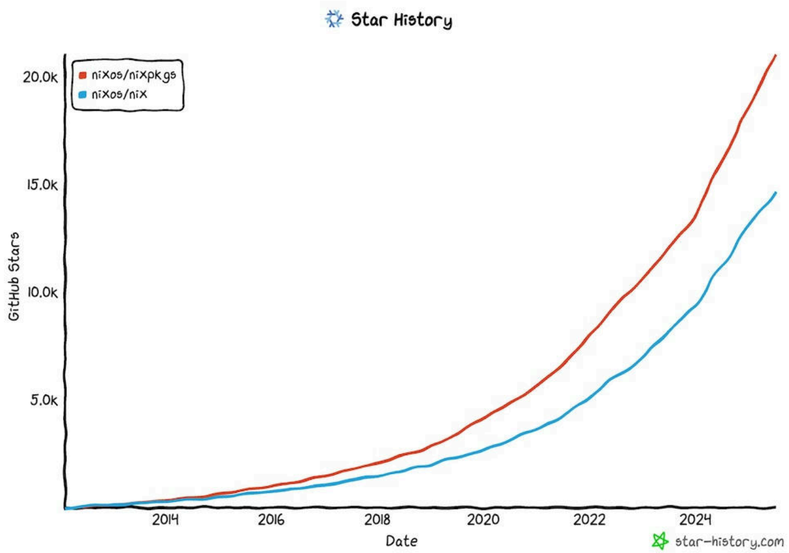
https://repology.org/repositories/graphs
as of 2024-11

Trainer notes

…that Nixpkgs is actually the biggest open source software repository in the world.

For everyone who feared that switching to Nix at work means rewriting many packages: No. How did you manage to live without it?

Nixpkgs: GitHub Star History



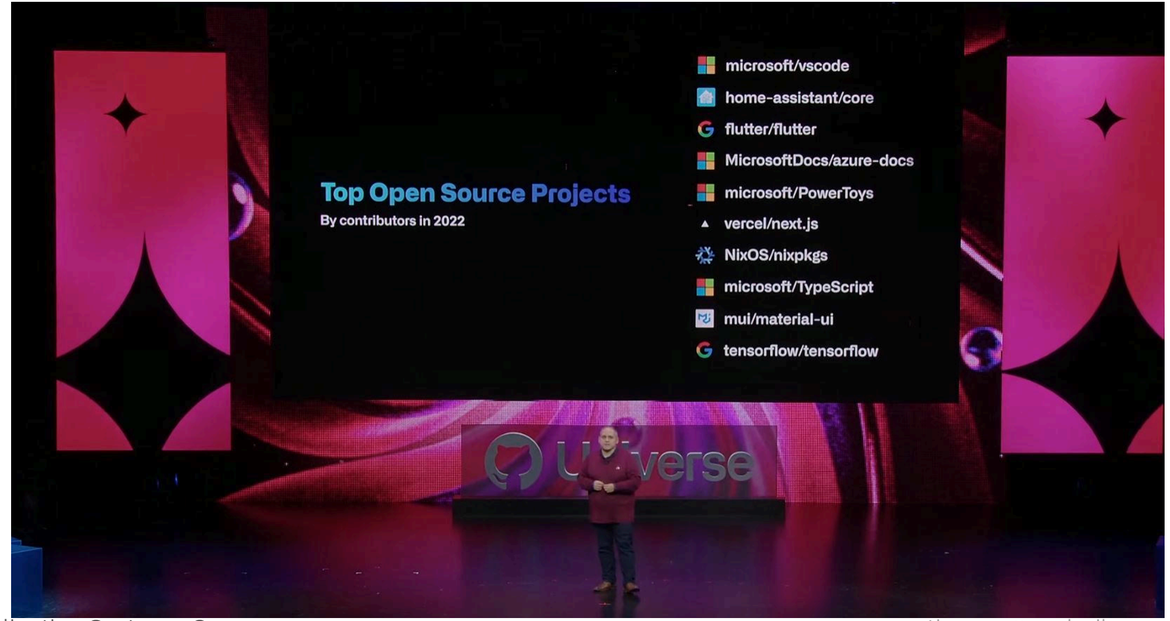
https://star-history.com
as of 2025-07

Trainer notes

It's just GitHub stars, but this growth curve will at some point flatten because it to some extent describes market growth and will result in a logistical curve shape: https://en.wikipedia.org/wiki/Logistic_function

We're not at the flattening part yet. The years where you can still be an early adopter are over very soon.

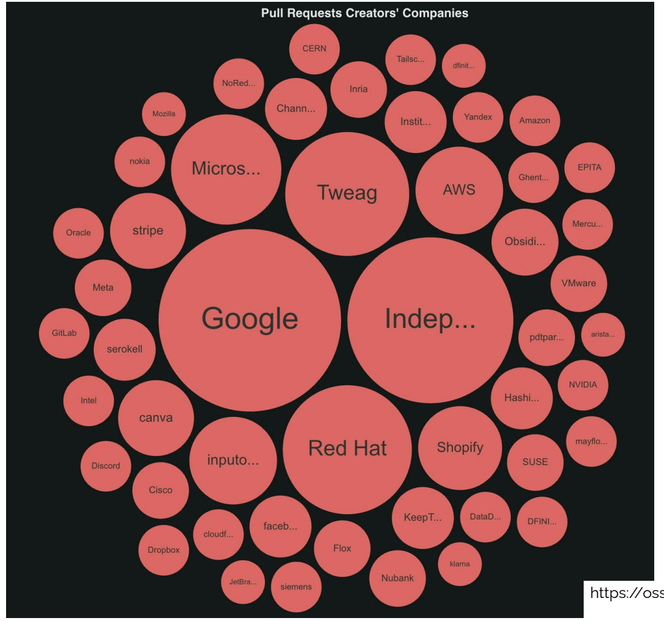
Nixpkgs: GitHub Top 10



Trainer notes

This was huge. Nixpkgs popping up between all the well-known and famous projects. 2022 was really a year in which things changed. Earlier, folks used to ask “What is this Nix stuff? Never heard about it” and slowly transitioned to “I really need to try this Nix stuff. Keep hearing about it.”

Nixpkgs: Pull Requests by Company



https://ossinsight.io/analyze/NixOS/nixpkgs
as of 2025-07

Trainer notes

Folks often underestimate the technology because they don't know how many big corporations are already using it. Read the big names loudly.

Our Steps to Master Nix

NIXCADEMY EXPERT EDUCATION

1. Nix Commands
   - User perspective
2. Nix Language
   - Functional Programming
3. The `builtins.derivation` primitive
   - Sandboxing
4. `stdenv.mkDerivation`: Generic package builds
   - Advanced composable sandbox construction
5. Patterns: `callPackage`, `override/overrideAttrs`, overlays
   - Composable build products
6. Flakes
   - The new interface for distribution to users

Trainer notes

Another meta slide that explains why we are tackling what.

This exists out of the experience that sometimes folks ask things like "Why are we learning functional programming now? I need dev shells."

Each step builds on the comprehension of the foregoing one. We know by experience that learning things in this order works really well.

Using Nix 🛠️

Trainer notes

Installation

One-liner installation command from https://nixos.org/download:

```
sh <(curl --proto '=https' --tlsv1.2 -L https://nixos.org/nix/install)
```

If asked for multi-user vs single-user installation: Select multi-user!

Trainer notes

The installer says “please close this terminal and open a new one before using the nix commands” - prepare for folks ignoring this if anyone tries installing right now.

Installation: Avoid distro packages

NIXCADEMY EXPERT EDUCATION

Prefer the official installer from https://nixos.org/download!

Only Debian/Ubuntu and Arch Linux packages are *tested*
(Link to automated tests)

If you used some other distro's packages or in doubt
👉 Uninstall and reinstall with official installer.

Trainer notes

Nix & the Nix Community has the resources to test their install on multiple platforms. You can trust it.

Installation: Alternative Determinate Systems Installer

Determinate Systems provides an alternative installer with improved UX, auto-enabled flakes, and improved macOS compatibility:

https://github.com/DeterminateSystems/nix-installer

```
curl --proto '=https' --tlsv1.2 -sSf -L https://install.determinate.systems/nix | \
  sh -s -- install
```

Please note that it nowadays installs `determinate-nix` by default.

Trainer notes

Folks might be unsure if they miss out anything if they used the "wrong" installer. Tell them they don't miss out anything. It's mostly just activating flakes, which we will see in a minute how to do.

Installation: Alternative Determinate Systems Installer

```
tfc — -zsh — 120×33
Last login: Tue Jun 20 08:21:24 on console
tfc@MacBook-Pro ~ % curl --proto '=https' --tlsv1.2 -sSf -L https://install.determinate.systems/nix | sh -s -- install

info: downloading installer https://install.determinate.systems/nix/tag/v0.9.1/nix-installer-x86_64-darwin
 `nix-installer` needs to run as `root`, attempting to escalate now via `sudo` ...
Password:
Nix install plan (v0.9.1)
Planner: macos (with default settings)

Planned actions:
* Create an APFS volume `Nix Store` for Nix on `disk1` and add it to `/etc/fstab` mounting on `/nix`
* Fetch `https://releases.nixos.org/nix/nix-2.15.0/nix-2.15.0-x86_64-darwin.tar.xz` to `/nix/temp-install-dir`
* Create group `nixbld` (GID 30000)
* Create a directory tree in `/nix`
* Move the downloaded Nix into `/nix`
* Setup the default Nix profile
* Place the Nix configuration in `/etc/nix/nix.conf`
* Configure the shell profiles
* Configure Nix daemon related settings with launchctl
* Remove directory `/nix/temp-install-dir`


Proceed? ([Y]es/[n]o/[e]xplain): yes
 INFO Step: Create an APFS volume `Nix Store` for Nix on `disk1` and add it to `/etc/fstab` mounting on `/nix`
 INFO Step: Provision Nix
 INFO Step: Configure Nix
 INFO Step: Configure Nix daemon related settings with launchctl
 INFO Step: Remove directory `/nix/temp-install-dir`
Nix was installed successfully!
To get started using Nix, open a new shell or run `. /nix/var/nix/profiles/default/etc/profile.d/nix-daemon.sh`

tfc@MacBook-Pro ~ %
```


Trainer notes

This is how the installation looks like for everyone who has never seen it.

It might be interesting to explain the bullet points to give everyone a feeling what installing Nix even means.

Enabling the New Nix Commands

NIXCADEMY EXPERT EDUCATION

Trainer notes

Non-NixOS

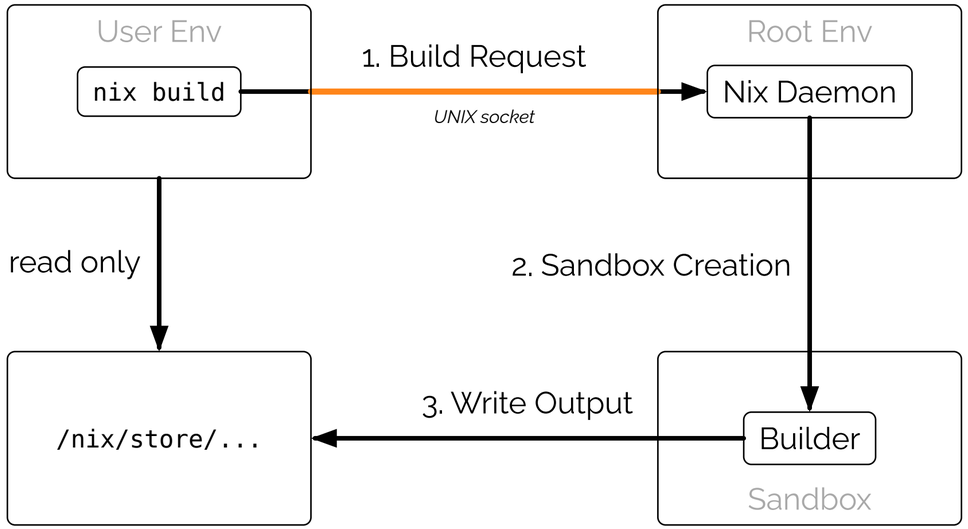
Add this line to `~/.config/nix/nix.conf` or `/etc/nix/nix.conf`:

```
experimental-features = nix-command flakes
```

NixOS

Add this to your NixOS configuration:

```
{
  nix.settings.experimental-features = [
    "nix-command" "flakes"
  ];
}
```



Trainer notes

It is important that folks know:

- Only nix daemon can write to /nix/store.
- Each sandboxed builder creates one /nix/store (* split output pkgs exists but that's not interesting right now)

1. Nix language evaluation happens in the frontend.
2. Frontend sends backend *derivations* (which we learn about in detail later)
3. Nix daemon is the backend that creates sandboxes, let's builders run there, create store paths, and then returns back to the user.

- This architecture also makes it easy to distribute builds over network.

Basic Nix Commands

Command	Effect
`nix profile`	Manipulate Nix User Environment 📦
`nix shell` / `nix develop` v `nix-shell`	Create a shell with a specific environment 🐚
`nix-collect-garbage`	Delete unreferenced store paths ♻️
`nix repl --file '<nixpkgs>'`	open a REPL with `pkgs` in scope 🔄
`nix build` v `nix-build`	One-step evaluation and realization 🏭
`nix-instantiate`	Create derivation from nix expression
`nix derivation show`	Show a derivation 🔍
`nix-store -r`	Realize a derivation

Trainer notes

- The red commands are the one that we will be using most of the time.
  - There is a consumer/user workflow (build) and a developer workflow (dev shells)
- We will also learn some commands that help with development and debugging!
- Old commands will stay useful
  - New command interfaces regularly change, which can be annoying
  - Old commands are better suited for developers (e.g. don't have to provile -L to get log output like with the new commands)

NIXCADEMY EXPERT EDUCATION

Nix / NixOS Cheatsheet

Most Important Documentation Links

Nix Docs (Tool, Language)	https://nixos.org/manual/nix
Nixpkgs (Packaging, prog langs, library)	https://nixos.org/manual/nixpkgs
NixOS	https://nixos.org/manual/nixos
Nixpkgs package and NixOS module search engine	https://search.nixos.org

Imperative Package Management

Update package list	`apt update`	*happens automatically*
Search	`apt search <pkgname>`	`nix search nixpkgs <pkgname>`
Install	`apt install <pkgname>`	`nix profile install nixpkgs#<pkgname>`
Upgrade installed	`apt upgrade`	`nix profile upgrade '.*'`
List installed	`dpkg -l`	`nix profile list`
Remove	`apt remove <pkgname>`	`nix profile remove <list-number>`
Rollback	-	`nix profile rollback`

Per-Project Shells

Ad-hoc shell with packages	`nix shell nixpkgs#pkg1` or `nix shell nixpkgs#{pkg1,pkg2}`
Project-shell with flake	`nix develop`
Project-shell with `shell.nix` or `default.nix` file	`nix-shell`

Building Packages

Build `default.nix` or default pkg from flake	`nix build`
Build specific attributes (flakes)	`nix build .#pkg1 .#pkg2`

Input Management

Flakes	
Init flake project	`nix flake init`
Init flake-parts project	`nix flake init -t github:hercules-ci/flake-parts`
Update flake inputs	`nix flake update`
Update and commit lock file	`nix flake update --commit-lock-file`
Update specific input	`nix flake lock --update-input <name>`
Niv (Pre Flakes)	
Install Niv Files	`niv init`
Add GitHub Repository	`niv add <github-owner>/<reponame>`
Update inputs	`niv update`
Update specific input	`niv update <name>`
Switch branch/tag of input	`niv update <name> -b <gitref>`

Flake References

Flake in current Directory	`.`
Local path	`[path:]/path/to/repo`
HTTPS URL to flake tarball	`https://host/flake.tar.gz`
Git Repo via HTTPS	`git+https://host/repo`
Git Repo via SSH	`git+ssh://git@host/repo`
GitHub Repo	`github:owner/repo`
Specific Branch/Tag	`git+ssh://git@host/repo?ref=abc123`

Nix REPL

Start Nix REPL	`nix repl`
Load local Flake	`:lf .`
Build derivation	`:b attribute.with.derivation`
Build & Install derivation	`:i attribute.with.derivation`
Fully print expression	`:p some.expression`
Show documentation of builtin function	`:doc builtins.listToAttrs`
Show REPL help	`:?`

NixOS System Rebuild

Rebuild system and activate	`nixos-rebuild switch`
Rebuild system and activate for now without updating bootloader	`nixos-rebuild test`
Rebuild w/o activating, but update bootloader	`nixos-rebuild boot`
Rollback	`nixos-rebuild switch --rollback`
Build on host a, deploy to host b, authorize with sudo	`nixos-rebuild switch --build-host a --target-host b --use-remote-sudo`

Note: nixos-rebuild expects a flake `/etc/nixos/flake.nix` to exist, and within that flake, a `nixosConfigurations` attribute with the hostname of the current system.

Garbage Collection

Collect unreferenced nix store paths	`nix-collect-garbage`
Also collect old profile/system generations	`nix-collect-garbage -d`
Only delete up to 50GB	`nix-collect-garbage --max-freed 50G`
Find and link identical files	`nix-store --optimise`
Print all GC roots	`nix-store --gc --print-roots`

NIXCADEMY EXPERT EDUCATION

Professional Nix & NixOS Trainings
https://nixcademy.com/

https://nixcademy.com/cheatsheet

Trainer notes

Trainer should ideally have paper cheat sheets and hand them out.

The 3 Documentations

Nix	https://nixos.org/nix/manual
Nixpkgs	https://nixos.org/nixpkgs/manual
NixOS	https://nixos.org/nixos/manual

Nix commands, flakes, Nix language → Nix manual
Anything within `pkgs` → Nixpkgs manual
NixOS installation, configuration, modules, integration testing → NixOS manual

The Nixpkgs and NixOS manuals are not indexed by search engines because they are too large single-page documents. → Keep them handy with bookmarks! 📑

Trainer notes

The not-indexed-by-google problem is maybe *the* reason why folks think Nixpkgs/NixOS has no good documentation - Google just doesn't lead them to it.

All the very proficient Nix folks have read most of it. Everyone should longterm.

Installing & Removing Packages

NIXCADEMY EXPERT EDUCATION

Nix Profile

```
$ nix profile install nixpkgs#hello
installing 'hello'

$ hello
Hello World

$ nix profile remove hello
uninstalling 'hello'
```

If a derivation's output contains binaries in its `$out/bin` folder, it's an installable package.

Trainer notes

Nix is often described as a "package manager" so we learn how to install and uninstall packages.

At this point, briefly describe the `<repo>#<attribute>` syntax as folks might see it for the first time now. Mention that this can be *any* Git repo!

Mention that Nix users rarely use this because it's so easy to have the right tools installed per project with nix shells.

`nix profile` is now deprecated.

How Nix Profiles Work

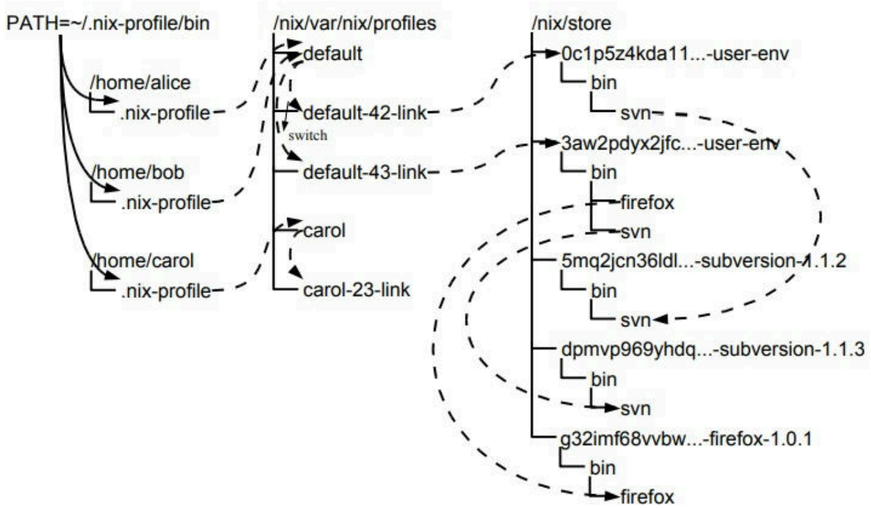


Figure 2.11.: User environments

Source: Nix PhD thesis

Trainer notes

Explain what “installing” really means by describing the different pointer indirections and “generations” here.

This is important for NixOS later too, because NixOS follows the same scheme for system generations.

What’s most interesting about this is that different folks can still rely on old generations of profiles when new ones are created.

Updating Packages

NIXCADEMY EXPERT EDUCATION

Trainer notes

Listing & Upgrading

```
$ nix profile list
...

$ nix profile upgrade --all
```

Garbage Collection ♻️

Nix has its own garbage collection for the nix store.

```
$ nix-collect-garbage -d
...
removing old generations of profile /nix/var/nix/profiles/per-user/tfc/profile
removing generation 94
...
deleting '/nix/store/jbchpg8...ac8m56vxs6j67-rpn_calc'
deleting '/nix/store/qpavjqr...9hgcpv1j1g2zi-rpn_calc'
deleting '/nix/store/trash'
deleting unused links...
note: currently hard linking saves 3683.55 MiB
2642 store paths deleted, 3129.22 MiB freed
```

This deletes all derivation outputs in the nix store that are not directly or indirectly referenced on the system.

Trainer notes

Tell folks to play with this maybe now if they haven't used Nix much yet, because later with huge nix stores this takes a long time.

Nix Language 🈴

Trainer notes

NIXCADEMY EXPERT EDUCATION

Why is Nix not just YAML?

Nix is like *JSON with functions*.

This language is very powerful and we will see why over the rest of this week.

Trainer notes

Often we have devops/admins who don't code in the trainings and they ask "why can't this just be yaml? seems like complete over-engineering".

We had many admins that engaged the class with the right openness and they learned many really nice tricks. But there was some relatively dry suspension between now and thursday, because it all falls nicely together like a puzzle piece on thursday.

In classes where folks ask many such ground questions because they are really not sold yet on Nix(OS), it might help to show some cool demos and examples at the beginning/ end of the day or breaks so folks find motivation to fight through the upcoming huge stack of abstractions.

Nix Language - Need to mix some theory & practice

Trainer notes

Many Nix language tutorials are either very flat or go too deep in theory. We see the necessity to highlight some of the theoretical foundation of the Nix language (e.g. lazy evaluation things, fix points, etc.) because it helps understanding the pkgs.lib ecosystem.

Although this language was invented for Nixpkgs and NixOS, it builds on decades old foundations of languages like Lisp and Haskell. For that reason, many function names that may appear strange to imperative programmers actually make sense because they refer to the lingo used by functional programmers. In this own world, it all makes sense and folks will get some advantage out of it by opening up for this different old world.

Nix REPL

NIXCADEMY EXPERT EDUCATION

Play with code in the REPL

```
$ nix repl --file '<nixpkgs>'
Added 20817 variables.
nix-repl> :b pkgs.hello

This derivation produced the following outputs:
  out -> /nix/store/rnxji3jf6fb0nx2v0svdqpj9ml53gyqh-hello-2.12.1
[1 copied (0.2 MiB), 0.0 MiB DL]
nix-repl>
```

- `:b x` builds derivation x and prints its output path(s)
- `:?` lists all colon-commands
- Note that the REPL has tab-completion
- (`nix repl nixpkgs` also works but is not the same)

Trainer notes

Show folks how useful the REPL is with its tab completion etc.

Also show `:?` and `:doc` features. Folks who never use the REPL might miss out on its great potential.

NIXCADEMY EXPERT EDUCATION

Attribute sets are like Python dicts

```
nix-repl> x = 1

nix-repl> s = { a = 1; b = x; }

nix-repl> s
{ a = 1; b = 1; }

nix-repl> t = { inherit x; inherit (s) a b; }

nix-repl> t
{ a = 1; b = 1; x = 1; }
```

Forget what you knew about "inheritance" from OO languages

Trainer notes

If folks look riddled or don't get something on the next slides - keep a REPL ready to play with things!

Ask if anyone can taxe a guess at what "inherit" might me by looking at the output.

Logical and Set Operators

NIXCADEMY EXPERT EDUCATION

Trainer notes

Beware: `or` is not `||`

```
nix-repl> true && (false || true)
true

nix-repl> set = { a = 1; b = 2; }

nix-repl> set ? a
true

nix-repl> set ? c
false

nix-repl> set.c or 10
10
```

Nix Syntax: Functions

NIXCADEMY EXPERT EDUCATION

Currying

```
nix-repl> plusOne = x: x + 1

nix-repl> plusOne 10
11

nix-repl> (x: x + 1) 10
11

nix-repl> f = a: b: a + b

nix-repl> f 1 2
3

nix-repl> plusOne' = f 1

nix-repl> plusOne' 10
11
```


Trainer notes

That folks understand what currying is is really important here. See also similarity of curried functions to objects with constructor calls on the next slide.

Nix Syntax: Currying vs. object orientation

Python:

```
> class Plus:
  def __init__(self, a):
    self.a = a

  def __call__(self, b):
    return self.a + b

> Plus(1)(2)
3
```

Nix:

```
> plus = a: b: a + b

> plus 1 2
3
```

Currying is just another way to bind local/member variables.

Trainer notes

Nix Syntax: Attribute Set Parameters

```
nix-repl> f = { a, b }: a + b

nix-repl> f { a = 1; b = 2; }
3

nix-repl> g = { a ? 100, c ? 200, ... }: a + c

nix-repl> g { a = 1; b = 2; }
201

nix-repl> h = s@{ a, ... }: a + s.b

nix-repl> h { a = 1; b = 2; }
3
```


Trainer notes

This is relatively straight-forward to explain. Can be demonstrated in the REPL.

Mention that the `@` thing is quite an anti-pattern if not used e.g. to forward `pkgs` from one function to the other. We show this here because it's being used a lot in Nixpkgs so folks have seen it once.

Nix Syntax: Variable Scopes

```
nix-repl> let x = 1; y = 2; in x + y
3

nix-repl> s = { a = 1; b = 2; }

nix-repl> with s; [ a b ]
[ 1 2 ]

nix-repl> let
            s = { a = 1; b = 2; c = 3; };
          in
            with s; [ a c ]
[ 1 3 ]
```


Trainer notes

This is all about explaining how scopes can be created.

In general, we can only set one variable imperatively after the other in the REPL. In real nix scripts, you create local scopes.

Nix Syntax: Variable Scopes 2

Nested `with` scopes

```
nix-repl> s1 = { a = 1; }

nix-repl> s2 = { a = 2; }

nix-repl> with s1; with s2; a
2

nix-repl> let
            a = 3;
          in
            with s1; with s2; a
3
```

This behavior can be surprising - use `with` sparingly!

Trainer notes

Here, it is important that everyone understands how `with` works and why it should generally be avoided. It seems acceptable in list-scopes like `with pkgs; [ hello gnumake gcc ]`.

I often painted a "variable table" on a whiteboard and then developed its content over the lines.

```
nix-repl> x = 1
nix-repl> bla = "bla"

nix-repl> "x = ${builtins.toString x}"
"x = 1"

nix-repl> ''
              interpolated: ${bla}
              not interpolated: ''${bla}
          ''
"interpolated: bla\nnot interpolated: \${bla}\n"
```

Trainer notes

It's important that folks understand that magic quotes remove the common whitespace prefix as we will use this a lot in package recipes.

NIXCADEMY EXPERT EDUCATION

Unions of Sets and Recursive Updates

```
nix-repl> { x = 1; } // { y = 2; }
{ x = 1; y = 2; }

nix-repl> { x = 1; z = 1; } // { y = 2; z = 3; }
{ x = 1; y = 2; z = 3; }

nix-repl> :p { x = { y = 1; }; } // { x = { z = 2; }; }
{ x = { z = 2; }; }

nix-repl> :p pkgs.lib.recursiveUpdate
               { x = { y = 1; }; }
               { x = { z = 2; }; }
{ x = { y = 1; z = 2; }; }
```


Trainer notes

Note that all the builtins functions and operators in Nix strive to be as primitive as possible as this simplifies the Nix implementation in C++ which must be fast and correct above everything else.

For "better"/higher functions look into `pkgs.lib`

Note that these operators return a new attrset, not a mutation.

Nix Syntax: Import function

NIXCADEMY EXPERT EDUCATION

File: x.nix

```
1 + 1
```

File: y.nix

```
x: 1 + x
```

Nix REPL

```
nix-repl> import ./x.nix
2

nix-repl> import ./y.nix
«lambda @ /path/to/y.nix:1:1»

nix-repl> import ./y.nix 10
11

nix-repl> f = import ./y.nix

nix-repl> f 10
11
```

Trainer notes

Trace values at evaluation time

```
nix-repl> x = 1

nix-repl> builtins.trace
            "value of x: ${builtins.toString x}"
            (x + x)
trace: value of x: 1
2
```


Trainer notes

There is no real debugging in Nix apart from tracing stuff. (There is a debugger in the Nix REPL but it's still buggy.)

There is no "stepping" through the code because there are no "steps" as in imperative languages.

Ideally, step to the whiteboard and paint a graph of an expression like `(1 + 2) * (3 + 4)`. The evaluation can be parallelized or done in any order as long as each node has no side effects.

All nix programs always are basically just graph construction that is then reduced

Why doesn't this fail?

```
let
  x = 1 / 0;
  y = [ (1 / 0) (2 / 0) ];
in
  if builtins.length y > 1
    then "all good!"
    else x
```

See also "Weak head normal form". (Look it up)

Let's play with this one in the REPL.

Trainer notes

Let folks guess what this program evaluation will look like.

Also go to the whiteboard and paint the whole memory layout that this program results in like here:
https://video.nixcademy.com/w/8prAmX6afHW7CiFUXjnLBU

This connects the theory with some graspable practice that actually looks useful to many folks who are still not sure why we need all this "magic".

Head vs. Tail

```
nix-repl> mylist = [ 1 2 3 4 5 ]

nix-repl> builtins.length mylist
5

nix-repl> builtins.head mylist
1

nix-repl> builtins.tail mylist
[ 2 3 4 5 ]
```

What are the *types* of `head` and `tail`?

Trainer notes

We will need this in the next exercises

One **must** use `length` to check the bounds of lists since neither `head` nor `tail` are 'safe'.

File: `fibonacci.nix`

```
let
  fib = x:
    if x == 0 then 0
    else if x == 1 then 1
    else fib (x - 1) + fib (x - 2);
in
  fib
```

Let's play with this one in the REPL.

Trainer notes

This should really be played with in the REPL. Folks find this regularly confusing but this is how we bootstrap recursion.

To master the Nix syntax and iteration in purely functional programming once and forever, let's do **2** quick language exercises.

After that, we are done with learning this seemingly academic language and start doing very practical things.

Trainer notes

Some folks asked "I thought this is a nix class - why functional programming?" and were a bit impatient.

The functional prog class part is over after the next 2 exercises, which can be solved in 10 minutes - unless you have to fight the syntax… then they turn out to be actually useful!

We do this so that we don't stumble upon the Nix syntax any longer during the rest of the week.

Exercise Time: Write a Sum Function

NIXCADEMY EXPERT EDUCATION

Task:

- Write a function `sum` that accepts a list of numbers and returns the *sum*
- Put the implementation into a file, use it in the REPL
- Test cases:
  - `sum [ 1 2 3 ] = 6`
  - `sum [] = 0`

Notes:

- Functions from `builtins.*` that will be useful: `length`, `head`, `tail`.
- Solution: https://nixcademy.com/downloads/nix-functions-solutions.zip

Trainer notes

Before all exercises: Unpack the code together on the shared screen and show folks what's inside, explain the structures etc. before letting everyone go on their own.

It's important to show folks that they need to use `:r` between file changes and reruns of their programs.

Emphasize that it's totally fine to work in groups!

Typical mistakes:

- writing `[ head list ]` which should be `[ (head list) ]`

Exercise Time: Write a Take-N Function

NIXCADEMY EXPERT EDUCATION

Task:

- Write a function `takeN` that returns only the first N items of a list
- Put the implementation into a file, use it in the REPL
- Test cases:
  - `takeN 3 [ 1 2 3 4 5 ] = [ 1 2 3 ]`
  - `takeN 3 [ 1 2 ] = [ 1 2 ]`
  - `takeN 3 [] = []`

Notes:

- Functions from `builtins.*` that will be useful: `length`, `head`, `tail`.
- Concatenate lists with `[ 1 ] ++ [ 2 ] = [ 1 2 ]`
- Solution: https://nixcademy.com/downloads/nix-functions-solutions.zip

Trainer notes

Typical mistakes:

- `n-1` instead of `n - 1`. Folks find this weird and frustrating but many packages have `-` in their names…

Where Do Functions Come From?

Nix is a real language that is able to solve a big share of real-world algorithmic problems.

`builtins.*` Documented in the Nix Manual
`pkgs.lib.*` Documented in the Nixpkgs Manual

Nix developers should read at least once through these files:

- `nixpkgs/lib/attrsets.nix`
- `nixpkgs/lib/lists.nix`
- `nixpkgs/lib/customisation.nix`

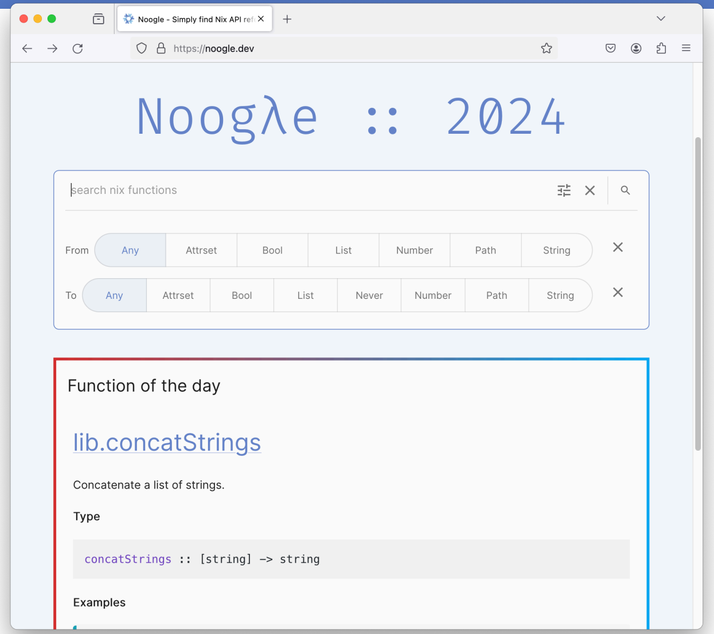
Many folks don't and end up reimplementing algorithms. Nearly everything you need is there already!

Trainer notes

Folks reinventing nix functions is actually quite the cost factor in orgs. Many code review hours per year pile up when seniors end up explaining juniors politely that they again and again reinvented the wheel.

Search for functions in `builtins.*` and `pkgs.lib.*`

- by name
- by type



Trainer notes

It might be interesting to show folks how hoogle works and to demonstrate them that noogle initially tried to provide type-based search to find names for given function characteristics described by their type.

Nixpkgs interfaces & implementation style

Trainer notes

Nix newcomers often wonder where those strange function names and type hints come from. It's all basically influenced by Haskell, which has a rich history and provides very elegant solutions for the problem space that Nix is in.

Importing Nixpkgs

Ways to import Nixpkgs:

```
pkgs = import ./path/to/nixpkgs { };
```

or

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

or

```
pkgs = import (builtins.getFlake "nixpkgs") { };
```


Trainer notes

That is finally it from the maybe boring functional prog 101 part. Let's finally *use* this language for package related stuff!

Show the participants that the Nixpkgs GitHub repository basically just contains a huge nix expression and importing that folder opens its `default.nix`.

Importing Nixpkgs with configuration options

NIXCADEMY EXPERT EDUCATION

Options for importing Nixpkgs

```nix
pkgs = import ./path/to/nixpkgs {
  # optional parameters:
  config = {
    allowBroken = true;
    allowUnfree = true;
    allowUnsupportedSystem = true;
    # e.g. a package contains meta.knownVulnerabilities ["CVE-XXXX-XXXX"]
    permittedInsecurePackages = [
      "openssl-1.2.3"
    ];
  };
};
```

See also Nixpkgs Manual, Chapter 2.6: Config Options Reference

Trainer notes

As `nixpkgs` the repo basically contains a huge nix function, let's see what's its arguments are.

- `permittedInsecurePackages` is actually nice for security things.
- Also demonstrate that pkgs.steam or pkgs.hello-unfree don't evaluate by default and then we need `allowUnfree`. Explain that this is because otherwise the nixpkgs CI would build and cache proprietary software which might be illegal for some packages.
- if folks are interested, show the meta fields that describe license stuff etc. — this might trigger interest in some security-oriented participants.

Packaging Basics 📦

Trainer notes

NIXCADEMY EXPERT EDUCATION

Trainer notes

- Evaluation translates `.nix` expressions to `.drv` *derivations*
- A derivation is a machine-readable recipe without dynamic parts
- A *realized* derivation results in a read-only output path with build products: `/nix/store/<hash>-<packagename>`

Evaluation & realization phases

`.nix` ⟶ `.drv` ⟶ `/nix/store/<hash>-<packagename>`

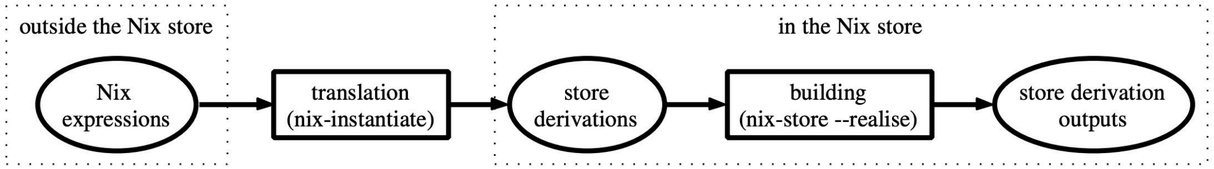
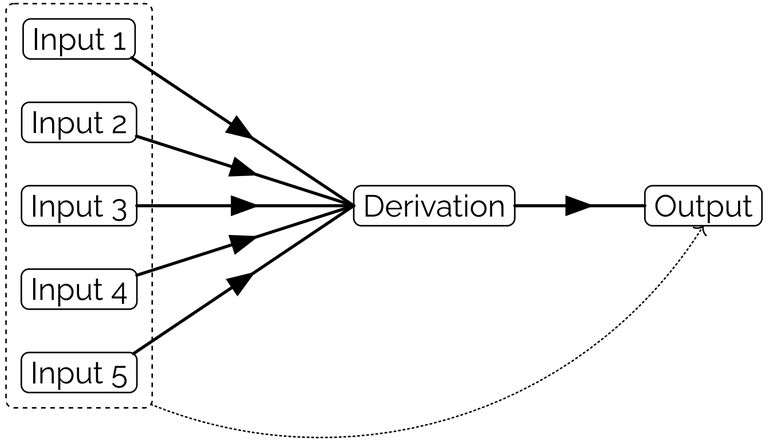


Figure 2.12.: Two-stage building of Nix expressions

from Nix PhD thesis

Trainer notes

For a US audience, it would be good to quickly note that the Nix flag is `--realise`, with an 's', which will trip up American users. Some CLIs build in localization to handle English variants, but Nix is not one of them. Nix is also inconsistent with its English variant usage based on the whoever wrote the code.



Hash parameters for output path

Trainer notes

It's worthwhile for the understanding that "Derivation" is like a pure function: With the same inputs, it will produce the same output. Nix tries to model exactly that for software builds!

Also, we address things from its input hashes, unlike Docker which uses output hashes. This empowers us to simply rebuild everything that isn't cached anywhere, making nix a source+binary distro, kind of.

A Minimal Nix Derivation

NIXCADEMY EXPERT EDUCATION

File: `builder.sh`

```
echo "foobar" > $out
```

File: `default.nix`

```
let pkgs = import <nixpkgs> { };
in
derivation {
  name = "myDerivation";
  builder = "${pkgs.bash}/bin/bash";
  args = [ ./builder.sh ];
  system = builtins.currentSystem;
}
```

Derive: `nix-instantiate default.nix`
Build: `nix-build default.nix`

Trainer notes

Super important: Demonstrate this in the terminal. Add coreutils, discover the content of the sandbox thoroughly so folks understand what it is all about. How do we get nix store paths inside? what happens with file references?

Show with the `nix derivation show` command that derivations basically map to:

1. list of nix store paths that need to be mounted into the sandbox
2. env + builder + args map 1:1 to the `execve` linux system call (show man page of this system call in the browser, quick google)

Nix is a not a build system but a dependency manager for running other build systems.

Exercise Time: Make it Build!

NIXCADEMY EXPERT EDUCATION

Download 💾

https://nixcademy.com/downloads/mini-hello.zip

Task:

- Use `builtins.derivation` to package `mini-hello` (don't use `stdenv.mkDerivation`)
- Inspect `nix derivation show $(nix-instantiate)`
- Provide the binary in the `$out/bin/` folder and install it with `nix profile install <output path>`

Notes:

- Needed packages are: `gcc`, `gnumake`, `coreutils`
- All derivation attributes become **environment vars** at build time
- Solution: https://nixcademy.com/downloads/mini-hello-solution.zip

Trainer notes

Some teams need 20 minutes for this exercise, some need 2 hours.

It's always the ones who try to do shortcuts who take longer.

Prepare to re-explain how bash + `$PATH` work for less experienced participants.

Show that `nix-build` will change the hash unless you move the sources to something like `src/`. Filesets will be discussed later.

Dependency Tree - Calculating Closures

NIXCADEMY EXPERT EDUCATION

Let's have a look at the dependency tree of our app!

Print the dependency tree

```
$ nix-store --query --tree <store path>
```

Explain the difference between these two variants:

Run Time vs Compile Time Dependencies

```
$ nix-store --query --tree $(nix-build ...)
$ nix-store --query --tree $(nix-instantiate ...)
```

Also: Check out the command `nix why-depends <package> <dep>`

Trainer notes

Show folks the dep trees.

Ask "How does nix know which runtime dependencies are left over in the output package? Why does it know that gnumake is not needed any longer?"

Run `grep -a "/nix/store" result/bin/hello` on the executable from the exercise to show folks how Nix is able to scan for runtime dependencies in the output. This information is not literally obtained with grep, but similar. It is later stored in a database.

Use `ldd` on a executable to see its library dependencies.

Moving Closures with `nix copy`

NIXCADEMY EXPERT EDUCATION

Trainer notes

Copy Closure via SSH

```
$ nix copy --to ssh://server $(nix-build ...)
```

Copy *build* or *runtime* closure to other host and build or run the package there.

Pinning 📌

Trainer notes

☝️ 📌 Note about Flakes and Pinning

NIXCADEMY EXPERT EDUCATION

Pinning: Don't Flakes do this for us already?

Yes, but here we understand *how* it works, what the caveats are, and it's still relevant for individual packages.

Trainer notes

Nix PATH

`<nixpkgs>` is expanded with a path that is looked up from `NIX_PATH`

Content of the `NIX_PATH`

```
$ echo $NIX_PATH
nixpkgs=/nix/var/nix/profiles/per-user/root/channels/nixos:nixos-config=/
etc/nixos/configuration.nix:/nix/var/nix/profiles/per-user/root/channels

$ nix repl

nix-repl> <nixpkgs>
/nix/var/nix/profiles/per-user/root/channels/nixos/nixpkgs
```

- The nixpkgs from here are simply a git clone of the original repo
- See also Nix Manual, Chapter 18: Common Environment

Trainer notes

This is cool to use but potentially different on every computer…

Pinning expressions 📌

To make nix expressions build *forever*, it's possible to *pin* all the inputs.

```
$ cat nixpkgs.nix
builtins.fetchTarball {
  url = "https://github.com/nixos/nixpkgs/archive/bd4e35e14a0130268c8f7b253
a15e09d76ec95b7.tar.gz";
  sha256 = "0k9ffj84mq21w17jfdh0rcr9pcvnbmaq9y3r7yd9k1mvry708cq5";
}

$ nix repl
nix-repl> nixpkgsSrc = import ./nixpkgs.nix
nix-repl> pkgs = import nixpkgsSrc {}
```

- The hash references a specific Git commit
- If the commit is so old that no binaries are available on `cache.nixos.org`, Nix will simply rebuild the packages

Trainer notes

This is how we can make software build forever.

Pinning app `npins` in Action

```
$ nix-shell -p npins
$ npins init
$ npins add github nixos nixpkgs -b nixos-unstable --name nixpkgs
$ npins add github nixos nixpkgs -b nixos-21.11 --name nixpkgs-old
$ nix repl

nix-repl> sources = import ./npins

nix-repl> pkgs = import sources.nixpkgs { }
nix-repl> pkgs.gcc.version
"14.3.0"

nix-repl> pkgsOld = import sources.nixpkgs-old { }
nix-repl> pkgsOld.gcc.version
"11.2.0"
```


Trainer notes

`niv` is also a good tool but `npins` is more famous now because it is well maintained and has a few advantages over niv.

Look how easy it is to mix multiple variants of nixpkgs in the same project!

Pinning Repo Inputs with Flakes 📌

NIXCADEMY EXPERT EDUCATION

Flakes in Action

```
$ nix flake init

$EDITOR flake.nix # need to edit manually

$ nix repl

nix-repl> :lf .
Added 9 variables.
nix-repl> inputs.nixpkgs.legacyPackages.x86_64-linux.gcc.version
"13.3.0"
nix-repl> inputs.nixpkgs-old.legacyPackages.x86_64-linux.gcc.version
"12.2.0"
```


Trainer notes

We are mostly ignoring flakes until we really introduce them later. This is just to show that this is very similar to pinning via flakes and we actually don't need flakes for the same functionality.

Building Software - The Typical Dance 👯

unpack, configure, build, test, install

```
wget https://company.com/product-1.2.3.tar.gz
tar xf product-1.2.3.tar.gz
cd product-1.2.3
./configure
make
make test
make install
```

It's always similar, no matter the language, OS, etc.

Trainer notes

The idea here is to show that all software always is built the same way with small deviations. The GNU project already structured this in "phases".

We are going to learn how `stdenv.mkDerivation` makes building 100k different packages easy and elegant by modeling these phases.

We look at this with some example C projects, but these phases are relevant for all the languages.

A Typical Nix Derivation

A Standard Derivation with mkDerivation

```
pkgs.stdenv.mkDerivation {
  name = "myPackage";
  src = ./.;
  # ...

  installPhase = ''
    # only necessary if there is no install target
    mkdir -p $out/bin
    cp myApp $out/bin/
  '';
}
```

See also Nixpkgs Manual, Section 6.1: Using stdenv

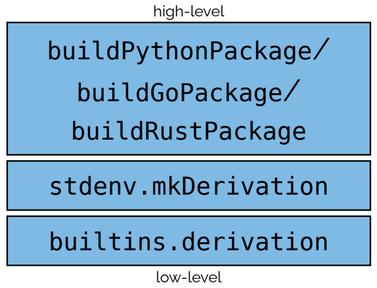
Trainer notes

The essense here is that we don't need to provide any information about a phase of a package build if the package uses standard tools. We only need to describe the phases that a package may be lacking.

Note about `mkDerivation`

I need Go/Rust/Python builds, why are we looking at mkDerivation and C projects?

All the `buildGoModule`, `buildRustPackage`, `buildPythonPackage` functionality builds on top of `mkDerivation`. To master it (especially in the cases when we need to debug things), we need to understand `mkDerivation`'s flow.



Trainer notes

Exercise Time - Build GNU hello 👋

NIXCADEMY EXPERT EDUCATION

Download 💾

https://gnu.mirror.constant.com/hello/hello-2.9.tar.gz

Goals:

- Create a `default.nix` and get it to build with `nix-build`
- Use `pkgs.stdenv.mkDerivation`
- Keep this folder, we will revisit it

Trainer notes

Folks shall see that mkDerivation actually works with this package only using the `name` and `src` attributes.

Most folks have the `default.nix` in the folder of the unpacked hello package. This leads to the interesting effect that the build can't be cached because of the `result` symlink being part of the closure. Let folks find out a bit why we get no caching and how to fix it. Will need some pointers.

stdenv Contents

The `stdenv` comes with the following dependencies by default:

- Bash
- C Compiler, configured with C and C++ support, (GNU on Linux, Clang on macOS)
- coreutils
- findutils
- diffutils, `patch`
- `patchelf`
- `awk`, `grep`, `sed`
- `tar`, `bzip2`, `gzip`, `xz`
- Make

All tools come as the GNU variants.

Trainer notes

Defining Build Inputs

Dependencies are listed as parameters to `mkDerivation` in the following attributes:

`nativeBuildInputs`	compile time deps
`buildInputs`	run time deps
`propagatedBuildInputs`	run time deps of scripts
`checkInputs`	test deps

These are the most important ones.
See also Nixpkgs Manual, Section 6.3.1. Variables specifying dependencies

Trainer notes

Folks should understand `nativeBuildInputs` and `buildInputs` and why they should choose this correctly (mostly cross comp etc.)

From the other things they should have heard, especially scripters, because they might end up needing this at some point.

mkDerivation Phases

`mkDerivation` uses *phases*:

1. `unpackPhase`
2. `patchPhase`
3. `configurePhase`
4. `buildPhase`
5. `checkPhase`
6. `installPhase`
7. `fixupPhase`
8. `installCheckPhase`
9. `distPhase`

Each phase …

- … acts as a *hook* with a most-of-the-time useful default implementation
- … can be overriden if the default behavior doesn't apply for the use case
- … can be augmented with additional behavior using `pre<phase>` and `post<phase>`

We will learn how to customize or override these.

Trainer notes

While explaining the phases and that they can all be overridden, point out that 8 and 9 are actually not widely used.

It might be cool to show folks the original builder script. It can be found in the `$stdenv` path of any derivation.

Enabling/Disabling Phases

- `dontUnpack = true`
- `dontPatch`
- `dontConfigure`
- `dontBuild`
- `doCheck`
- `dontInstall`
- `dontFixup`
- `doInstallCheck`
- `doDist`

The phases of...

`do*` `dont*`

...attributes are...

disabled enabled

❌ ✅

...by default

Trainer notes

this dont and do stuff is a bit of a naming fail but it's easy to memorize if you ignore phase 8 and 9 and then you only need to remember that checks are off by default. this might be due to the observation that most software is not tested well.

Fetching Code with `fetchurl`

```
stdenv.mkDerivation {
  name = "myPackage";

  src = pkgs.fetchurl {
    url = "ftp://alpha.gnu.org/gnu/sed/sed-4.2.2-pre.tar.bz2";
    sha256 = "11nq06d131y4wmf3drm0yk502d2xc6n5qy82cg88rb9nqd2lj41k";
  };

  # ...
}
```

The `src` attribute is a **fixed output derivation**

Trainer notes

Fixed output derivations actually have access to the internet.

Fetching Code: Caveats ⚠️

NIXCADEMY EXPERT EDUCATION

Let's update an existing package

```
stdenv.mkDerivation {
  name = "sed";

  src = pkgs.fetchurl {
-   url = "ftp://alpha.gnu.org/gnu/sed/sed-4.2.2-pre.tar.bz2";
+   url = "ftp://alpha.gnu.org/gnu/sed/sed-5.0.0.tar.bz2";
    sha256 = "11nq06d131y4wmf3drm0yk502d2xc6n5qy82cg88rb9nqd2lj41k";
  };

  # ...
}
```

What could go wrong?

Trainer notes

Fetching Code: Caveats ⚠️

NIXCADEMY EXPERT EDUCATION

Let's update an existing package

```
stdenv.mkDerivation {
  name = "sed";

  src = pkgs.fetchurl {
-   url = "ftp://alpha.gnu.org/gnu/sed/sed-4.2.2-pre.tar.bz2";
+   url = "ftp://alpha.gnu.org/gnu/sed/sed-5.0.0.tar.bz2";
    sha256 = "11nq06d131y4wmf3drm0yk502d2xc6n5qy82cg88rb9nqd2lj41k";
  };

  # ...
}
```

What could go wrong? → Yes, we need to change the **hash**, too!
(See also `pkgs.lib.fakeHash`)

Trainer notes

Obtaining the Correct Hash

NIXCADEMY EXPERT EDUCATION

Trust on first use (TOFU) with `lib.fakeHash`

```
unpacking source archive /build/hello-2.9.tar.gz
error: hash mismatch in fixed-output derivation '/nix/store/....drv':
         specified: sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
            got:    sha256-1aFSTdB6F9qR8hch6QTZzQm5rgEhh1SbNIfFs61FsK8=
error: 1 dependencies of derivation '/nix/store/....drv' failed to build
```

Avoiding TOFU: Hash local file before upload

```
$ nix hash file /.../myfile.tar.gz
sha256-1aFSTdB6F9qR8hch6QTZzQm5rgEhh1SbNIfFs61FsK8=
```


Trainer notes

This avoiding-TOFU thing only exists because some folks pointed out that TOFU is unsafe and asked if nix supports safe pinning. The command line there however is deprecated and i stopped fixing it because it kept changing. Better look up the manual for the exact command.

More About Fetchers

There are many more fetchers:

- `fetchzip` - *auto unpack (any tarball format)*
- `fetchpatch` - *normalizes patches*
- `fetchgit` - *supports submodules*
- `fetchFromGitHub`
- `fetchFromGitLab`
- …

See also Nixpkgs Manual, Section 11: Fetchers

Trainer notes

Mention, no need to demonstrate.

`nix-prefetch-scripts` to find hashes for fetchers

`pkgs.nix-prefetch-scripts` contains a number of scripts to help get hashes for many types of fetchers. More advanced prefetchers echo JSON to `stdout`.

```
$ nix-prefetch-git https://git.savannah.gnu.org/git/hello.git
…
{
  "url": "https://git.savannah.gnu.org/git/hello.git",
  "rev": "1a04646423406df85b4c41609e55eb8482806247",
  "date": "2025-07-25T12:56:49+01:00",
  "path": "/nix/store/xhkn46xbc4ci9rv4ww1if0q71ga3clyl-hello",
  "sha256": "1wz74gb0xsjlwfd7a783hbzp7m0zhx5yy8wjfr7jj3izj1fw5i95",
  "hash": "sha256-JcXCXZA/DilPdpIj70uHH9Rz/4IDHXWa41TqDtYj5/M=",
  "fetchLFS": false,
  "fetchSubmodules": false,
  "deepClone": false,
  "fetchTags": false,
  "leaveDotGit": false,
  "rootDir": ""
}
```


Trainer notes

Also mention, no need to demonstrate (the code block should make it clear enough).

Exercise Time - Extend the GNU hello derivation 👋

NIXCADEMY EXPERT EDUCATION

Download 💾

https://gnu.mirror.constant.com/hello/hello-2.9.tar.gz

Goals:

- Write the configure, build, check and install phases yourself
- Use `mkDerivation`'s `patches = [ ... ];` parameter to customize this app.

Notes:

- You might need the `--prefix` parameter at configure time
- `make check` runs the unit tests
- use `diff -ru folder1 folder2 > my.patch` creates a patch file
- Solution: https://nixcademy.com/downloads/hello-solution.zip

Trainer notes

The goal is that everyone sees how easy it is to override phases.

The patching often goes wrong. so ideally write down the exact commands that should be used for each step on a whiteboard, because surprisingly many folks have never done something like this in their life.

Debugging Failing Builds

NIXCADEMY EXPERT EDUCATION

- `nix-build --keep-failed`
- `nix-build --debug`
- Set `NIX_DEBUG` in the derivation's attrs (see nixpkgs manual 6.4.1.1)
- Add `pkgs.breakpointHook` to `nativeBuildInputs` (see nixpkgs manual 16.4.)

Trainer notes

the breakpoint hook is ideal for a little demonstration. linux only of course because it's not supported anywhere else.

Language-Specific Builders

Nixpkgs comes with 50 language-specific builders/frameworks:

Agda, Android, Astal, BEAM Languages (Erlang, Elixir & LFE), Bower, CHICKEN, Coq and coq packages, COSMIC, Crystal, CUDA, Cue (Cuelang), Dart, Dhall, D (Dlang), Dotnet, Emscripten, Factor, GNOME, Go, Gradle, Hare, Haskell, Hy, Idris, Idris2, iOS, Java, Javascript, Julia, lisp-modules, Lua, Maven, Nim, OCaml, Octave, Perl, PHP, pkg-config, Python, Qt, R, Ruby, Rust, Scheme, Swift, Tcl, TeX Live, Typst, Vim, Neovim

If you work with one of these languages/techstacks, *read* the whole chapter about it. It helps **a lot**. 📚

Manual: https://nixos.org/manual/nixpkgs/stable/#chap-language-support

Trainer notes

Python Builder Example

```
pkgs.python3Packages.buildPythonPackage rec {
  pname = "mypackage";
  version = "1.0.0";

  pyproject = true;
  build-system = [ setuptools ];

  src = pkgs.python3Packages.fetchPypi {
    inherit pname version;
    sha256 = "08f...ef0";
  };

  doCheck = false;
}
```

See also Nixpkgs Manual, Python Specific Builder

Trainer notes

We sneaked in `rec` in this slide. quickly explain it but refer to `finalAttrs` which is better an will be shown on a later slide.

Also explain `pname` + `version` as an alternative to `name`.

Go Builder Example

```nix
pkgs.buildGoModule {
  pname = "mypackage";
  version = "1.2.3";

  src = pkgs.fetchFromGitHub {
    owner = "my-org";
    repo = "my-repo";
    rev = "v1.2.3";
    hash = "sha256-abc123...";
  };

  vendorHash = "sha256-def456...";
  # set to `null` for vendored deps
}
```

See also Nixpkgs Manual, Go Specific Builder

Trainer notes

This is interesting because folks familiar with that toolchain will immediately understand what the extra parameters are about. this is why folks should read the docs and then be up to speed very quickly.

Rust Builder

```
pkgs.rustPlatform.buildRustPackage {
  pname = "myproject";
  version = "1.0.0";

  src = ./.;
  cargoLock.lockFile = ./Cargo.lock;

  buildInputs = [ ... ];
}
```

See also Nixpkgs Manual, Rust Specific Builder

Trainer notes

Rust Builds: Consider `crane`

https://crane.dev

- Source fetching: Automatically done using a Cargo.lock file
- Pre-compiles Dependencies for Nix store
- Composable: Splits builds and tests into granular steps.
- Enables pure `clippy` use cases and much more

Usage:
Import `default.nix` via npins or `lib` attribute via flake.
Crane project function is similar to nixpkgs' `buildRustPackage`.

Trainer notes

Skip this if folks are not interested in rust.

If they are, talk about the advantages of this library but also that it's IFD and we will explain why IFD can be bad on a later slide.

Build Helpers 🏗️

NIXCADEMY EXPERT EDUCATION

"Build Helpers" (formerly "trivial builders") are wrapper functions around `mkDerivation` that are helpful for recurring tasks like:

- Create a preconfigured nix shell
- Running simple scripts to generate package outputs
- Creating prepackaged standalone scripts
- Creating text files
- Batch-Symlinking of scattered paths

See also Nixpkgs Manual, Chapter 12: Build Helpers, and `nixpkgs/pkgs/build-support/trivial-builders/default.nix`

Trainer notes

This whole subsection is about showing a little expose of functions and helpers to motivate looking into the linked source file so folks don't miss out on tools that might be very helpful to them.

Build Helpers: `mkShell`

Create a development shell

```
pkgs.mkShell {
  packages = with pkgs; [
    hello
    jq
  ];
  inputsFrom = with pkgs; [
    somePackage
  ];
  shellHooks = ''
    export FOOBAR=123
  '';
}
```


Trainer notes

Make sure to highlight the difference between `inputsFrom` and `packages`. Also note that `inputsFrom` does **not** include `checkInputs`.

Python Shell with Individual Python Versions

File: `shell.nix`

```nix
let
  pkgs = import <nixpkgs> { };
  myPython = pkgs.python313.withPackages (ps: with ps; [
    flask numpy requests
  ]);
in
  pkgs.mkShell {
    packages = [
      myPython                    # each of these can
      pkgs.python312.pkgs.black   # use a different
      pkgs.python314.pkgs.mypy    # python version.
    ];                            # -> no clash!
  }
```

Trainer notes

Make sure to highlight that each of these packages might end up using a different Python interpreter because they are all wrapped scripts. This is easy to show if there is no binaryWrapper involved (which it is on macOS).

Build Helpers: runCommand

The runCommand helper

```
let
  pkgs = import <nixpkgs> {};
  env = {
   nativeBuildInputs = [ pkgs.cowsay ];
  };
in
pkgs.runCommand "cowsay-output" env ''
  find "${pkgs.cowsay}/share/cowsay/cows" \
   -name "*.cow" \
   -exec cowsay -f {} "Hello Nix Users" >> $out \;
''
```


Trainer notes

run this and have a laugh over the output. then discuss how this is a useful helper for devops and admins who may just like to generate things for server configs.

Build Helpers: `writeShellApplication`

Package shell scripts

```
pkgs.writeShellApplication {
  name = "show-nixos-org";

  runtimeInputs = with pkgs; [ curl w3m ];

  text = ''
    curl -s 'https://nixos.org' | w3m -dump -T text/html
  '';
}
```

Most notable differences to `writeScript` et al:

- Runs `shellcheck` at build time
- `$out/bin/show-nixos-org` is *installable*

Trainer notes

This is a rock-solid distribution mechanism for shell scripts.

Build Helpers: `symlinkJoin` 1/2

NIXCADEMY EXPERT EDUCATION

Join folder structures

```
pkgs.symlinkJoin {
  name = "myFavoriteApps";
  paths = with pkgs; [ hello cowsay fortune ];
}
```

Result

```
$ ls result/bin/
cowsay  cowthink  fortune  hello  rot  strfile  unstr
```

Trainer notes

Nice example how bin folders for PATH environments can be assembled

Build Helpers: `symlinkJoin` 2/2

Synthesize folder structures

```
pkgs.symlinkJoin {
  name = "myLinkFarm";
  paths = [
    derivationWithTarballs1
    derivationWithTarballs2
    ...
  ];
  postBuild = ''
    sha256sum $out/* | sed 's|/nix/store/.*/||' > $out/hashes.txt
  '';
}
```

This derivation could now for example be served as-is by a webserver.

Trainer notes

Project Patterns 🎨

Trainer notes

Project Structure: Example Project

```
|
+-- a/
|   +- default.nix
|   +- ...
+-- b/
|   +- default.nix
|   +- ...
+-- c/
|   +- default.nix
|   +- ...
+-- nix/
|   +- overlay.nix
|   +- sources.nix
|   +- ...
+-- flake.nix
+-- default.nix
+-- release.nix
+-- shell.nix
```

- Either `flake.nix` and/or `release.nix` (or both)
- Each module in their own folder
  - Minimizes closures
  - Facilitates structure
- Sane default shell and default files
- Release/Flake file gives access to all build targets
  - All variants of binaries
  - Integration tests
  - Automatic linting, format checkers, etc.
  - Documentation

Trainer notes

This is kind of the archetype project structure suggestion. It's important to mention that submodule A and B and C can refer to each other via input definitions, but they don't know where they physically are.

This is a big contrast to real life projects where everything is often intermingled into a big spaghetti ball of code.

The release.nix/flake.nix basically are everything a CI would need to build everything, so the CI description can be actually just one command line. Easy to reproduce for devs!

The callPackage Pattern 📞 - Package Normal Form

NIXCADEMY EXPERT EDUCATION

Example `a/default.nix` *package function*

```
{ stdenv , boost, openssl, staticBuild ?
false }:

stdenv.mkDerivation {
  name = "my-app";
  src = ./.;
  buildInputs = [
    boost
    openssl
  ];
  # ...
}
```

Usage: `a = pkgs.callPackage ./a/default.nix { };`

Quick & Dirty: `nix-build -E "(import <nixpkgs> {}).callPackage ./default.nix {}"`

Trainer notes

The "package normal form" separates the **mechanism** of *how* to build a package from the **policy** of deciding *which* versions/variants of all the inputs are in use.

At first this might feel like a pointless indirection, but this buys us a lot of freedom.

The callPackage Pattern

How does `callPackage` even work?

Let's play with `callPackage`

```
nix-repl> f = { x, y, z }: x + y + z

nix-repl> myCallPackage = pkgs.lib.callPackageWith { x = 1; y = 10; }

nix-repl> myCallPackage f { z = 100; }
111

nix-repl> myCallPackage f { z = 100; x = 2; }
112
```


Trainer notes

This slide shall highlight how `callPackage` is a function that needs to be "preloaded" with a set of variables and then it can "auto-apply" functions with all the variables it knows.

Show this in the REPL

The callPackage Pattern on Sets

One more thing...

Let's redecide things after the fact

```
nix-repl> f = { x, y, z }: { result = x + y + z; }

nix-repl> myCallPackage = pkgs.lib.callPackageWith { x = 1; y = 10; }

nix-repl> foo = myCallPackage f { z = 100; }

nix-repl> foo.result
111

nix-repl> (foo.override { x = 2; }).result
112
```


Trainer notes

`callPackage` comes with a second feature that only unlocks if we use it with functions that return an attribute set: `override`!

Let's redecide things after the fact without knowing the original recipe!

Project Structure: release.nix

File: `release.nix`

```nix
let pkgs = import <nixpkgs> {};
    a = pkgs.callPackage ./a/default.nix { };
in
{
  inherit a;

  a-static = a.override {
    staticBuild = true;
    openssl = pkgs.openssl.override { static = true; };
  };

  a-patchedversion = a.overrideAttrs (oldAttrs: {
    patches = oldAttrs.patches or [] ++ [
      ./patches/specialstuff.patch
    ];
  });
}
```

`callPackage` and `override` unlock huge potential!

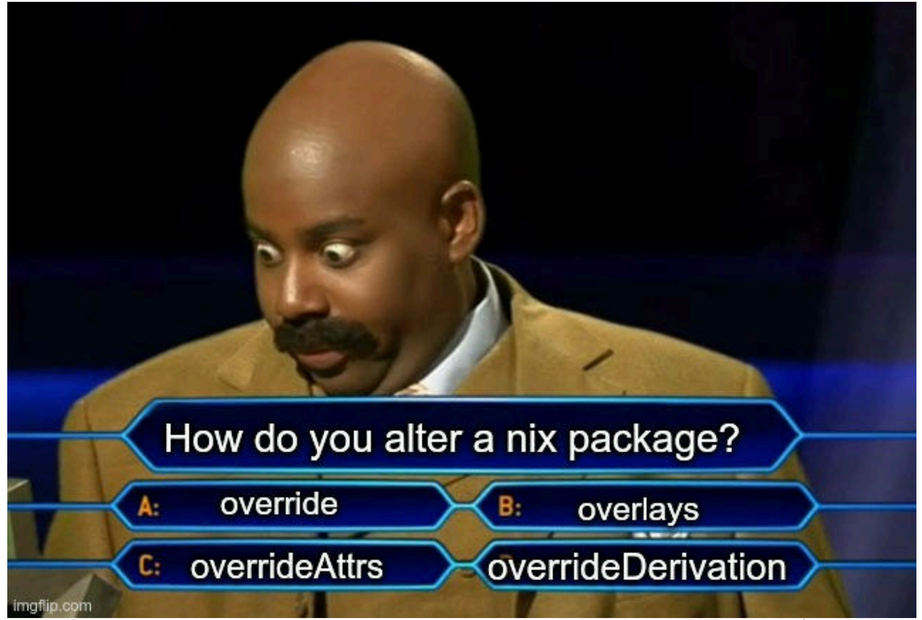
Trainer notes

A `release.nix` file contains the policy decisions:

- how do we import which nixpkgs version
- what variants of our attributes do we provide
  - these can be generated with override and overrideAttrs calls

In flake projects, flake.nix would do the same thing.

When to use `override`/`overrideAttrs`?



Trainer notes

overrideAttrs was just mentioned but not introduced - let's fix that.

override vs overrideAttrs

Override

```
drv.override {
  x = y;
}
```

overrideAttrs

```
drv.overrideAttrs (oldAttrs: {
  patches = oldAttrs.patches or []
    ++ [ ./my.patch ];
})
```

There are many legitimate use cases for `overrideAttrs`.

Still, design your build functions so that callers don't need `overrideAttrs` for *everything*.

Trainer notes

override vs overrideAttrs

```
{ stdenv , packageA , packageB }:        ← override changes these

stdenv.mkDerivation
{
  name = "my-app";
  src = ./.;
  buildInputs = [
    packageA                  ← overrideAttrs changes these
    packageB
  ];
  # ...
}
```

Trainer notes

release.nix - Selective Builds

File: `release.nix`

```
{
  package1 = ...;
  package2 = ...;
  package3 = ...;
}
```

`nix-build release.nix`	Builds all derivations in parallel
`nix-build release.nix -A package2`	Builds only package2

Trainer notes

Folks keep running `nix-build release.nix` and wondering why this builds multiple packages at the same time. So make sure that everyone remembers what `-A` is for.

Exercise Time - GNU Hello release.nix

Download 💾

https://gnu.mirror.constant.com/hello/hello-2.9.tar.gz

Goals:

- Rewrite your `default.nix` to the *normal* form
- Write a `release.nix` that provides different versions of `hello`:

`default`	Default variant, no patches
`hello-clang`	Built with Clang
`hello-patched`	Patched to print "Hello, Nix!"

Notes:

- Use `pkgs.clangStdenv` (Linux), `pkgs.gccStdenv` (macOS) or play with `pkgs.overrideCC`
- Solution: https://nixcademy.com/downloads/hello-release-solution.zip

Trainer notes

This exercise is difficult for many. Prepare to take more time and help.

The `with` Anti-Pattern

bad example

```
with import <nixpkgs> { };
with lib;

stdenv.mkDerivation {
  # ...
  buildInputs = foo [ a b c ];
}
```

good example

```
let
  pkgs = import <nixpkgs> { };
  inherit (pkgs) stdenv a b c;
  inherit (pkgs.lib) foo;
in
stdenv.mkDerivation {
  # ...
  buildInputs = foo [ a b c ];
}
```

- Unclear scope, clash probability
- Where do symbols come from?
- Evaluation performance penalty

Trainer notes

the best prep for these anti-pattern slides is reading the anti-patterns book

The rec Anti-Pattern

bad example

```
rec {
  x = 1;
  y = 2;
  z = x + y;
}
```

good example

```
let
  x = 1;
  y = 2;
in
{
  inherit x y;
  z = x + y;
}
```

Imagine this in large nix expressions

Trainer notes

rec in mkDerivation

NIXCADEMY EXPERT EDUCATION

bad example

```
stdenv.mkDerivation rec {
  pname = "bla";
  version = "1.2.3";

  src = someFetcher {
    url = "...${version}...";
    hash = "...";
  };
}
```

good example

```
stdenv.mkDerivation (finalAttrs: {
  pname = "bla";
  version = "1.2.3";

  src = someFetcher {
    url =
      "...${finalAttrs.version}...";
    hash = "...";
  };
})
```

Trainer notes

The Default Arguments Anti-Pattern

Accidental implicit singleton pattern

bad example

```
# file: foo.nix
{ pkgs ? import <nixpkgs> {}
, other
, parameters
}:

...
```

bad example

```
# File: `release.nix`
let
  pkgs = import ... {
    config.overlays = ...;
  };
  x = import ./foo.nix { };
in
{
  inherit (x) a b c;
}
```

Debugging experience: “Why does it always pick the wrong version of my inputs? I mean, I overrode it in an overlay, right?”

Trainer notes

The callPackage on Non-Derivations Anti-Pattern

bad example

```nix
# File: `foo.nix`
{ foo
, bar
, callPackage
}:
{
  x = callPackage ./x { };
  y = callPackage ./y { };
  ...
}
```

bad example

```nix
# File: `release.nix`
let
  pkgs = import ... { };
in
{
  # next line: bad
  foo = pkgs.callPackage ./foo.nix { };
}
```

Only use `callPackage` on *actual derivations*

See also `pkgs.callPackages`

Trainer notes

IFD: Import From Derivation - Mighty but with Tradeoffs

NIXCADEMY EXPERT EDUCATION

File: `generate-nix-expr.nix`

```
{ runCommand }:
runCommand "gen-nix-expr" {} ''
  mkdir $out
  sleep 5 # simulate slow generation
  cat << EOF > $out/hello.nix
  { stdenv, hello }:
  stdenv.mkDerivation {
    name = "hello";
    src = hello.src;
  }
  EOF
''
```

File: `default.nix`

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

  hello = pkgs.callPackage
    ./generate-nix-expr.nix
    {};
in

pkgs.callPackage
  "${hello}/hello.nix"
  {}
```

Change the `sleep` duration and watch the weird caching behavior.

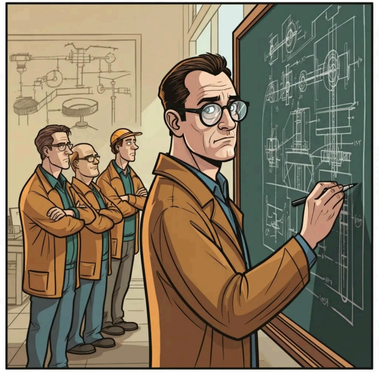
Trainer notes

Make sure to demonstrate this in the terminal and play with changes so folks learn and understand the caching characteristics.

Make sure that everyone gets that mostly individual devs think "cool this helps my project so that we can't lose sync between lock files and nix files". But what slows down the build for just a second for one project might create horror scenarios for downstream repos that consume 10 such projects. These observe builds that look like they are stuck for a minute.

Combined with bad source filtering/huge closures, IFD can really slow things down. There should be a pro/contra IFD policy in each org that weighs the tradeoffs.

Blog: IFD



Nix IFD: A Ticking Time Bomb in Your Build Pipeline?
https://nixcademy.com/posts/what-is-ifd-ups-and-downs/

Trainer notes

Advanced Packaging Topics 🚀

Trainer notes

NIXCADEMY EXPERT EDUCATION

Trainer notes

What is wrong here?

bad example

```
stdenv.mkDerivation {
  pname = "bla";

  src = ./.;

  # ...
}
```

Bad Practise: Unfiltered Source Folders

NIXCADEMY EXPERT EDUCATION

Everything is copied into the nix store:

- … all the Nix expressions
- … README files
- … build artifacts from local work
- … VCS metadata directories (for example: `.git/`)
- … editor state/backup files
- … `result` symlinks from previous builds (!!)
- The name of your folder becomes part of the nix store copy, too

Trainer notes

Bad Practise: Unfiltered Source Folders

NIXCADEMY EXPERT EDUCATION

Everything is copied into the nix store:

- … all the Nix expressions
- … README files
- … build artifacts from local work
- … VCS metadata directories (for example: `.git/`)
- … editor state/backup files
- … `result` symlinks from previous builds (!!)
- The name of your folder becomes part of the nix store copy, too

With source filters, we get:

- More cache hits
- Fewer rebuilds
- Fewer and smaller source copies into the nix store

Trainer notes

Source Filtering: cleanSource

NIXCADEMY EXPERT EDUCATION

`pkgs.lib.cleanSource`

Drops VCS files, `*.o` artifacts, editor backup/swap files, `result` symlinks

Example Usage

```
src = lib.cleanSource ./.;
```


Trainer notes

This function is a great low-effort alternative to no source filtering.

Source Filtering: Unions of Subfolders and Files

The `pkgs.lib.fileset` library

Example: https://github.com/tfc/pprintpp

```
src = lib.fileset.toSource {
  root = ./.;
  fileset = lib.fileset.unions [
    ./include
    ./test
    ./example
    ./CMakeLists.txt
    ./pkg-config.pc.cmake
  ];
};
```

Trainer notes

Source Filtering: Union with File Types

Only copy source files with certain file types

Example: `unions` + `hasExt`

```
let
  fs = lib.fileset;
  extensionOf = extensions: file:
    builtins.any file.hasExt extensions;
in
fs.toSource {
  root = ./.;
  fileset = fs.unions [
    ./.latexmkrc
    (fs.fileFilter (extensionOf [ "jpg" "pdf" "png" "tex" ]) ./.)
  ];
}
```


Trainer notes

This slide showcases how easy it is to write short but powerful filter functions.

However, it alienates many newcomers. So maybe explain on the whiteboard how it works and promise that with a bit of experience, folks will find it normal to read and write such code.

Source Filtering: Union and Difference

Subtract one file list from the other

Example: Composing filters

```
let
  fs = lib.fileset;
  nixFiles = fs.fileFilter (file: file.hasExt "nix") ./.;
in
fs.toSource {
  root = ./.;
  fileset = fs.difference (fs.gitTracked ./.) nixFiles;
}
```


Trainer notes

Debugging Source Filters

Source filters can be *traced*:

Example: Composing filters

```
src = let
  fs = lib.fileset;
  nixFiles = fs.fileFilter (file: file.hasExt "nix") ./.;
in
fs.toSource {
  root = ./.;
  fileset = fs.traceVal (fs.difference ./. nixFiles);
}
```

Remember: Complex source filtering *may* be indicator of clumsy project structure.

Trainer notes

In many orgs, it provokes less discussion to add *another* line to a too-complex filter function than restructuring the project in a same way.

But mind that often a complex filter function just alleviates the fallout of a clumsy project structure.

Fileset Library Docs

Nixpkgs Manual: `lib.fileset` file set functions

Implementation: `<nixpkgs/lib/fileset>`

Trainer notes

Cross compilation 🔀

Trainer notes

Nixpkgs's crossSystem Parameter

Importing nixpkgs with crossSystem for aarch64 builds

```
pkgs = import <nixpkgs> { };
systems = pkgs.lib.systems.examples;

aarch64Pkgs = import <nixpkgs> {
  crossSystem = systems.aarch64-multiplatform;
};
```


Trainer notes

Let's have a brief look at cross compilation so we can see

- how easy it is,
- how it *roughly* works,
- where to look if we need more information.

Some teams are *not* interested in cross compilation at all. If that is the case, feel free to skip this section.

Ideally, showcase this in the REPL and show how many platforms there are.

Dep Mgmt for Cross Compilation via Nixpkgs Splicing

NIXCADEMY EXPERT EDUCATION

Runs on build host, compiles for target

Runs on build host

Compatible with target

```
{ stdenv, cmake, openssl }:

stdenv.mkDerivation {
  name = "myproject";
  src = ...;

  nativeBuildInputs = [ cmake ];
  buildInputs = [ boost openssl ];

  # ...
}
```


Trainer notes

"Splicing" means that the same package function is called with `pkgs` for the build and the target platform. Then, a hybrid derivation is created with the blue/red parts.

The `stdenv` is purple because the compiler needs to run on the build platform but generate code for the target platform.

strictDeps: Disable legacy mode compatibility

NIXCADEMY EXPERT EDUCATION

- Many packages exist that mix up `nativeBuildInputs` and `buildInputs`. They don't cross-compile well.
- For this reason, `mkDerivation` puts both of them in `PATH`
- Use `strictDeps = true;` inside your `mkDerivation` call to disable this
- To enforce strict behavior on all packages, use the nixpkgs configuration option `strictDepsByDefault` (link to docs)

Trainer notes

Just use pkgsCross

Just use `pkgsCross`

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

windowsPkgs = pkgs.pkgsCross.mingwW64
```

pkgs exist for all attribute sets from `pkgs.lib.systems.examples`

Trainer notes

Nixpkgs's crossSystem Parameter: Static Linking

NIXCADEMY EXPERT EDUCATION

Trainer notes

Importing nixpkgs with isStatic

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

staticPkgs = import <nixpkgs> {
  crossSystem = {
    isStatic = true;
    # ... Select musl ...
  };
};
```

Just Use pkgsStatic

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

staticPkgs = pkgs.pkgsStatic
```

How Do pkgsCross/pkgsStatic Work?

Stdenv runs with different settings:

- Cross-Compiler
- compiler default flags (`NIX_CFLAGS`)
- configure default flags for autoconf, cmake, meson, etc.
- `glibc` substitute
- stdenv carries flags like `isStatic`

Most of the time these settings work. Some packages need tweaks.

See also
`nixpkgs/pkgs/top-level/stage.nix`
`nixpkgs/pkgs/stdenv/adapters.nix`

Trainer notes

Cross and Static Compilation Caveats

Many build system descriptions are hacked to work on specific systems but sacrifice *portability*.

General Advice:

- Do not mix dependency management with build management. Typical smells of non-portable projects:
  - Build system downloads and installs deps by itself
  - Build system looks for hardcoded compiler paths/names/versions
  - Subshells call build systems of git submodules
  - Build system sets static flags itself instead of providing mechanisms for both dynamic and static builds
- Know your tools, write portable build descriptions
- Keep your projects modular

Trainer notes

> The Nix Way™ is to basically provide just 1 compiler—& that being the > right compiler > > — Jacek

Some paths aren't even provided in the Nix sandbox & need to be created or patched.

Blog: More About Cross Compilation

Separation of Concerns in Cross-Compilation
https://nixcademy.com/posts/cross-compilation-with-nix/

Trainer notes

Altering parameters, env vars, etc. with `mkWrapper`

Example - This stops the "Java Duke" icon from jumping in macOS when using `plantuml` in the terminal:

`makeWrapper` example

```
let
  env = { nativeBuildInputs = [ pkgs.makeWrapper ]; };
  headless-jre = pkgs.runCommand "headless-jre" env ''
    mkdir -p $out/bin
    makeWrapper ${pkgs.jre}/bin/java $out/bin/java \
      --add-flags "-Djava.awt.headless=true"
  '';
in
pkgs.plantuml.override { jre = headless-jre; }
```

The best documentation of this tool is in its source code:
`nixpkgs/pkgs/build-support/setup-hooks/make-wrapper.sh`

Trainer notes

makeWrapper is immensely useful and powerful, but it's hard to motivate this so this is the best example i could come up with. It's from a real life story.

I have a similar with with OCaml. The ocaml-lsp project has a dependency on ocamlformat despite there being many formatters. I needed to patch in a the RPC only binary into a wrapper so that I didn't get loud Neovim warnings when opening a file.

I also find 2-space code very difficult to read so in my personal projects I often patch, or in the case of `nixfmt` wrap the binary to provide a `> 2 --indent` value.

“Dynamic” Patching

There are some advanced `sed/awk`-like facilities:

substitute script

```
export myVar=123

substitute ./in.txt ./out.txt \
  --replace-fail "foo bar" "baz qux" \
  --subst-var myVar \
  --subst-var-by otherVar lol
```

File: `in.txt`

```
foo bar
@myVar@
@otherVar@
```

File: `out.txt`

```
baz qux
123
lol
```


Trainer notes

This is cooler than `sed` or `awk` because `--replace-fail` fails loudly if it doesn't find a match.

NIXCADEMY EXPERT EDUCATION

Trainer notes

More Substitution Helpers

```
substituteInPlace a.txt b.txt c.txt \
  --replace-fail a b

# replace @x@ and @y@
export x=1
export y=2

substituteAll in.txt out.txt

substituteAllInPlace a.txt b.txt c.txt
```

Patching Shebangs

There is no `/usr/bin/{env,bash,python,...}` in the Nix sandbox.

All *executable* package outputs are patched automatically in the install phase. If you need to execute scripts in earlier phases, use:

Patching Shebang Lines Example

```
preBuild = ''
  patchShebangs ./scripts/
  ./scripts/foo.sh
'';
```

Patching `/usr/bin/env xxx` works if `xxx` is part of the build inputs.

Trainer notes

More Examples Online 1/3

https://galowicz.de/2023/01/23/mixed-cpp-monorepo-project/

- Organizing multiple C++ modules in one big project (CMake)

https://github.com/tfc/meson_cpp_example

- Organizing multiple C++ modules in one big project (Meson)

https://github.com/tfc/pprintpp

- The pprintpp C++ library, tested with multiple compilers

Trainer notes

Generally knowing what's in these articles is useful when suggesting examples to look at that the class might find relevant.

Feel free to provide your own examples!

More Examples Online 2/3

NIXCADEMY EXPERT EDUCATION

https://galowicz.de/2023/01/16/cpp-qt-qml-nix-setup/

- A QtQuick C++ Project with Nix

https://galowicz.de/2019/04/17/tutorial_nix_cpp_setup/

- Building a multi-dependency project with the cartesian product of compilers and input library versions

https://github.com/cyberus-technology/hedron

- OS kernel implemented in C++, built with multiple compilers

Trainer notes

More Examples Online 3/3

https://github.com/tfc/pandoc-drawio-filter

- Pandoc filter for draw.io images, includes nice self-test

https://github.com/tfc/electron-purescript-example

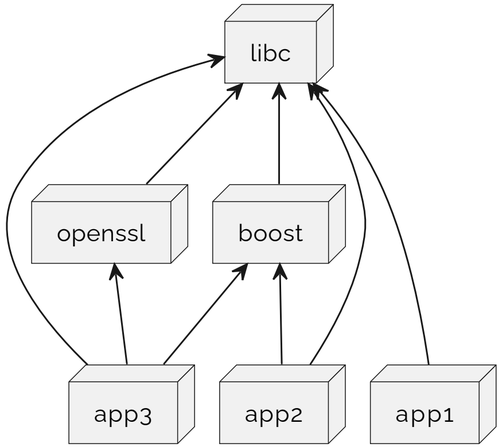
- Node.js + PureScript + Electron Example

Trainer notes

Overlays 🔁

Trainer notes

How to Patch Deps of multiple Packages?



`override` manipulates the inputs of one package.

How can we override a package for *all* downstream consumers?

Trainer notes

With `override` and `overrideAttrs` we are able to patch things "up" the dependency tree.

But how do we alter a package and make sure that everyone "down" the tree gets affected by this change?

This slide is more comprehensible when you write some pseudo code with overrides to show how to get a patched libc into all apps 1 2 and 3 without overlays.

What can overlays do?

- Add packages
- Update/Change packages
- All changes *can* affect pre-existing packages all along

Trainer notes

Nixpkgs Overlays - How to add one package

Example Overlay file `overlay.nix`

```nix
final: prev: {
  app = prev.callPackage ./app { };
}
```


Trainer notes

Emphasis here is that it "feels" like `app` was always part of nixpkgs although it never was.

For companies this has the implication that there are no "internal" and "external" packages anymore which makes it easier to integrate stuff.

Nixpkgs Overlays - How to add one package

NIXCADEMY EXPERT EDUCATION

Example Overlay file `overlay.nix`

```
final: prev: {
  app = prev.callPackage ./app { };
}
```

Example Evaluation

```
nix-repl> pkgs = import <nixpkgs> {
  overlays = [ (import ./overlay.nix) ];
}

nix-repl> :b pkgs.app
```


Trainer notes

Emphasis here is that it "feels" like `app` was always part of nixpkgs although it never was.

For companies this has the implication that there are no "internal" and "external" packages anymore which makes it easier to integrate stuff.

Nixpkgs Overlays - global package manipulation

Example Overlay file `overlay.nix`

```nix
final: prev: {
  opencv4 = prev.opencv4.override {
    enableGtk3 = true;
    enableTesseract = true;
  };
}
```

All packages that depend on `pkgs.opencv4` are now affected by this change.

Trainer notes

Nixpkgs Overlays - inject changes in specific pkgs

Example Overlay file `overlay.nix`

```
final: prev: {
  opencv4-xxx = final.opencv4.override {
    enableGtk3 = true;
    enableTesseract = true;
  };
  myApp = prev.myApp.override {
    opencv4 = final.opencv4-xxx;
  };
}
```

Only `myApp` is affected. `opencv4-xxx` is also prepared for *further* overrides.

Note `prev` vs. `final` in this example!

Trainer notes

How does it work? Enter Fix Point Recursion

Definition of `pkgs.lib.fix`

```
fix = f: let x = f x; in x;
```


Trainer notes

We are interrupting the practical overlay examples now to learn how fixpoint recursion works as this will be helpful to really grasp everything.

This example should be demonstrated in the REPL so that folks get an intuition for it.

How does it work? Enter Fix Point Recursion

Definition of `pkgs.lib.fix`

```
fix = f: let x = f x; in x;
```

```
nix-repl> f = self: {
  x = 1;
  y = self.x + 1;
  z = self.y + 1;
}
```


Trainer notes

We are interrupting the practical overlay examples now to learn how fixpoint recursion works as this will be helpful to really grasp everything.

This example should be demonstrated in the REPL so that folks get an intuition for it.

How does it work? Enter Fix Point Recursion

Definition of `pkgs.lib.fix`

```
fix = f: let x = f x; in x;
```

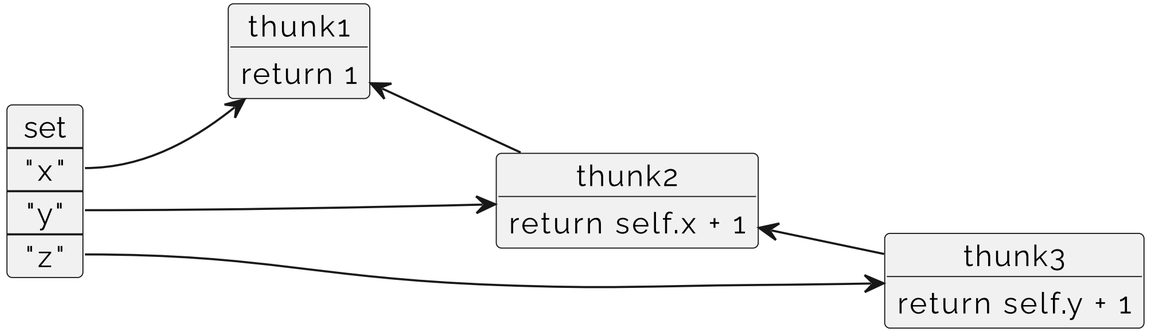
```
nix-repl> f = self: {
  x = 1;
  y = self.x + 1;
  z = self.y + 1;
}

nix-repl> pkgs.lib.fix f
{ x = 1; y = 2; z = 3; }
```


Trainer notes

We are interrupting the practical overlay examples now to learn how fixpoint recursion works as this will be helpful to really grasp everything.

This example should be demonstrated in the REPL so that folks get an intuition for it.

Understanding Laziness + Fixpoint "Recursion"



Trainer notes

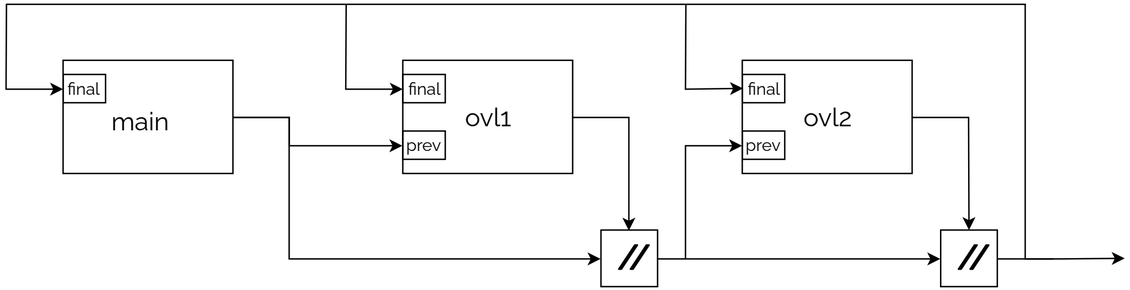
This is the memory image that will be (roughly) created like in this video: https://video.nixcademy.com/w/vNn7t3qX52kS9iGg3D8yTi

Do this ideally live on the whiteboard. It will help folks understand how this is not a completely theoretical thing. The essence is: Lazy fixpoint recursion on attribute sets is just an elegant way to create directed (acyclic) graphs.

Potential cycles can be created, but Nix detects that quickly using "blackholing", which means while traversing the graph, Nix leaves "bread crumbs" (bits set to 1) on every node on every visit.

- https://www.microsoft.com/en-us/research/wp-content/uploads/1992/04/spineless-tagless-gmachine.pdf
- the (linked in the next slides) laziness article on the nixcademy blog also explains this

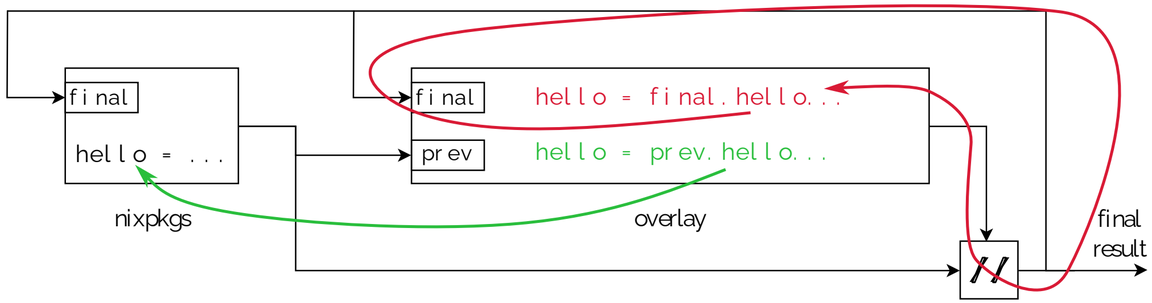
Nixpkgs Overlays: Final vs Prev



Trainer notes

Jumping hence and forth between the overlay examples and this "superloop" image generally helps a lot explaining when to use prev and final.

Infinite Recursion - How does it happen in overlays?



Trainer notes

Final vs Prev: Rules

NIXCADEMY EXPERT EDUCATION

The Graft-Preserving Rules Flavor:

- Use `prev` by default.
- Use `final` only if you reference a package/derivation from some other package.

The Everything-Is-Overridable Rules Flavor:

- Use `final` by default.
- Use `prev` if you override a symbol to avoid infinite recursion.

Either ruleset is fine. Be consistent within your project/org.

Trainer notes

These two sets of rules are the result of many hours of research. No one wrote this down before as clearly as us, and the inventor of overlays blessed this as correct.

What is graft-preserving? The idea of overlays initially was that when you e.g. patch libc, *everything* on top would have to be rebuild, which might take a few days. Grafting is a technique that can rebuild only the changed packages and then `patchelf`-relink the dependants in short time. This has been presented in a pull request but never merged. Companies which are big enough might still implement it internally.

There were talks about getting "grafting" in Nix in some flavor, but those the proposal never made it to the finish line.

To be able to talk more of this, listen to the Nicolas's talks linked to in the Nixcademy overlays article (linked in the next slides)

Composing overlays

```
ovl1 = final: prev: { ... };
ovl2 = final: prev: { ... };

combined1 = pkgs.lib.composeExtensions ovl1 ovl2;

combined2 = pkgs.lib.composeManyExtensions
              [ ovl1 ovl2 ];
```

See also `nixpkgs/lib/fixed-points.nix`

Trainer notes

Two or many overlays can be fused together so that they work like one overlay.

Maybe skip this and then give this as an extra task in the next exercise to everyone who finished early and likes a challenge.

Exercise Time: Play with Overlays

NIXCADEMY EXPERT EDUCATION

Get the Git repo 💾

```
$ git clone https://github.com/nixcademy/override-overlay-exercise
```

Tasks:

1. Use `override` & `overrideAttrs` to patch "abcd" to "xyzw" (only change `release.nix`)
2. Create an overlay file `overlay-patched.nix`, include it in the nixpkgs import, achieve the same effect with it.

Notes

- You can use `substituteInPlace` to patch return values of functions.
- Solutions are in the git branches `solution-1` and `solution-2`

Trainer notes

This is the most difficult exercise and everything gets easier after this point.

Prepare to help folks as some folks will be really quick and some really slow and helpless.

More about Overlays Online

(Thread on x.com)

Nicolas is the inventor of nixpkgs overlays

https://nixcademy.com/posts/mastering-nixpkgs-overlays-techniques-and-best-practice/

Trainer notes

More About Lazy Evaluation in Nix

Lazy Evaluation in Fixpoint Recursion
https://nixcademy.com/posts/what-you-need-to-know-about-laziness/

Trainer notes

Flakes ❄️

Trainer notes

What are Flakes? ❄️

NIXCADEMY EXPERT EDUCATION

🆕 🆒 🎁 ❄️

- 🚀 Cached Evaluation
- Concept of *Evaluation Closure*
- Hermetic *Evaluation* (no `NIX_PATH`, no `builtins.currentSystem`, …)
- Registry instead of Channels
- Built-in Pinning and overriding of Evaluation Inputs
- Nice tooling for attribute discovery
- Introduction of (opinionated) structure enforcement

Trainer notes

Until this point, we have looked at practically all the things that folks want/need to do with Nix and we did not need Flakes for that. We also learned about some features that regular Flakes users have not seen before.

This class was specifically designed in a way that we learn how to do the same things with and without Flakes. This way to teach Nix is also "blessed" by some NixOS foundation folks who work on the official documentation.

In the flakes section, we learn about how to put Flakes "on top" of what we've done so far.

Flakes Bring Some New Fancy Usability

Example: Composable nix shells

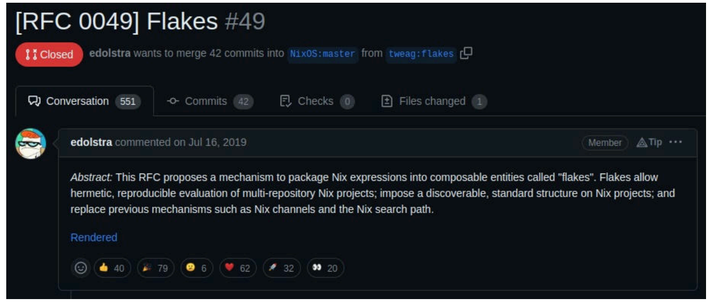
```
nix shell nixpkgs#{nodejs,ruby}

nix shell \
    nixpkgs#ruby \
    github:nixos/nixpkgs/nixos-20.03#nodejs
```

Example: Run apps from the internet w/o checkout

```
nix run github:nix-community/nixos-anywhere
```

Trainer notes

Experimental State? It's Complicated... 🙈 🙉 🙊



Lunch at NixCon October *2022* in Paris 🥐 🥖 :
Jacek asked Eelco "When will Flakes be stable and enabled by default?"
Eelco: "Probably something around 3 months." 🤦

However, Flakes are very usable and bring some nice benefits.

Trainer notes

Eelco started Nix as a project for research and experimentation. Flakes started as another experiment but the community was already too large and too many folks depend on Nix as a production tool already, to accept these changes as another experiment going into Nix without discussion.

Nowadays, when we say Flakes are "not stable", this means that the command line interface is not considered stable, yet.

Before you let 300 folks use Nix with flakes, think about how costly it is in your org (all projects, all CI descriptions etc.) to update Nix and potentially have to fix non-backwards-compatible changes.

NIXCADEMY EXPERT EDUCATION

Trainer notes

Non-NixOS

Add this line to `~/.config/nix/nix.conf` or `/etc/nix/nix.conf`:

```
experimental-features = nix-command flakes
```

NixOS

Add this to your NixOS configuration:

```
{
  nix.settings.experimental-features = [
    "nix-command" "flakes"
  ];
}
```

A Minimal Example Flake

File: `flake.nix`

```nix
{
  description = "A flake for building Hello World";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-23.11";

  outputs = { self, nixpkgs }: let
    pkgs = nixpkgs.legacyPackages.x86_64-linux;
  in {
    packages.x86_64-linux.default = pkgs.hello;

    devShells.x86_64-linux.default = pkgs.mkShell {
      packages = [ pkgs.something ];
    };
  };
}
```


Trainer notes

- `packages.${system}.` is an attribute set that shall *not* be nested.
- `pkgs...` *is* nested, however, so they put it in `legacyPackages`.

Look at all the boilerplate code of what could have been a 5 liner `release.nix`. The old-school `release.nix` would practically work on any platform while this Flake hardcodes everything to one platform…

Helper Libraries: Flake-utils

File: `flake.nix`

```nix
{
  description = "Flake utils demo";

  inputs.flake-utils.url = "github:numtide/flake-utils";

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let pkgs = nixpkgs.legacyPackages.${system}; in
      {
        packages.default = pkgs.hello;
      }
    );
}
```

Trainer notes

`flake-utils` helps with this hardcoding to platforms by consuming a platform-agnostic function from you and then it multiplies your outputs on all systems. (which ones? That's hidden in the `eachDefaultSystem` function)

Potential anti-pattern: Flake-utils

Flake-utils can be considered an anti-pattern for mostly these reasons:

- It's easy to implement yourself
- Flakes end up depending on 10 different versions of flake-utils transitively
- They "hide" which platforms are supported

1000 Instances of flake-utils
https://nixcademy.com/posts/1000-instances-of-flake-utils/

Trainer notes

Helper Libraries: Flake.parts

File: `flake.nix`

```nix
{
  description = "Flake.parts example";

  inputs.flake-parts.url = "github:hercules-ci/flake-parts";

  outputs = inputs@{ flake-parts, ... }:
    flake-parts.lib.mkFlake { inherit inputs; } {
      systems = [
        "x86_64-linux" "aarch64-linux" "aarch64-darwin" "x86_64-darwin"
      ];
      perSystem = { config, pkgs, system, ... }: {
        packages.default = pkgs.hello;
      };
    };
}
```


Trainer notes

flake.parts are a *powerful* library that does so much more than flake-utils. Using it for system-specific application like flake-utils can be considered over-engineering.

In a nutshell: flake.parts use the NixOS module library (which we learn about in the NixOS part) to make Flakes as composable as NixOS modules are.

This is quite a difficult topic and also controversial among the Nix community. We love to provide special trainings just for that.

Flake Inputs

```
inputs = {
  nixpkgs.url = "github:NixOS/nixpkgs/nixos-23.11";

  some-flake.url = "git+https://some.git.com/repo";
  some-flake.inputs.nixpkgs.follows = "nixpkgs";

  other-flake.url = "git+ssh://git@github.com/....git";
  nixpkgs-other.follows = "other-flake/nixpkgs";
};
```

`nix flake update` updates inputs and recreates the `flake.lock` lock file 🔒

no extra `npins` tooling needed 📌

See also Nix Manual, Chapter 7.5.1 nix flake

Trainer notes

- Flake inputs can be so many different protocols etc., it's really comfortable.
- The `follows` mechanism goes both ways (see both examples).
- `follows` is *similar* to overlays but overlays are much more powerful! Don't ignore one for the other.

Flakes Export Different Kinds of Outputs

`packages`	Buildable Packages: `nix build flake#myPkg`
`apps`	Runnable apps: `nix run flake#myApp`
`nixosModules`	Importable NixOS modules
`nixosConfigurations`	Deployable NixOS configs: `nixos-rebuild --flake flake#myConfig --target root@host`
`checks`	Tests and checks: `nix flake check`
`overlays`	Overlay functions
`devShells`	Development shells: `nix develop flake#myShell`

Trainer notes

This is an overview and if folks can't imagine what these are individually for, show them examples from our/your own repos.

Group Exercise Time: Convert Repo to Flakes!

NIXCADEMY EXPERT EDUCATION

Get the Git repo 💾

```
$ git clone https://github.com/nixcademy/override-overlay-exercise
```

Task:

- Let's port `release.nix` together to `flake.nix`

Notes:

- In git repos, nix flake commands don't see files without prior `git add`
- apps attrs look like this:
  `apps.x86_64-linux.myapp = { type = "app"; program = "/exec/path"; }`
- Solutions in branches `solution-flake`

Trainer notes

To do this, we need to import nixpkgs in the flake explicitly and apply the overlay. This shows how much work it is to get a Flake running properly.

This might support the case "Flakes are for users while developers better use non-flakes stuff daily."

Go Flake Example

Get the Git repo 💾

```
$ git clone https://github.com/tfc/nixery
```

- Flakified version of https://github.com/tazjin/nixery
- Original author is not a fan of flakes
- Works on GNU/Linux and macOS

Trainer notes

Only for those who are interested

Rust Flake Example

NIXCADEMY EXPERT EDUCATION

Get the Git repo 💾

```
$ git clone https://github.com/tfc/rust_async_http_code_experiment
```

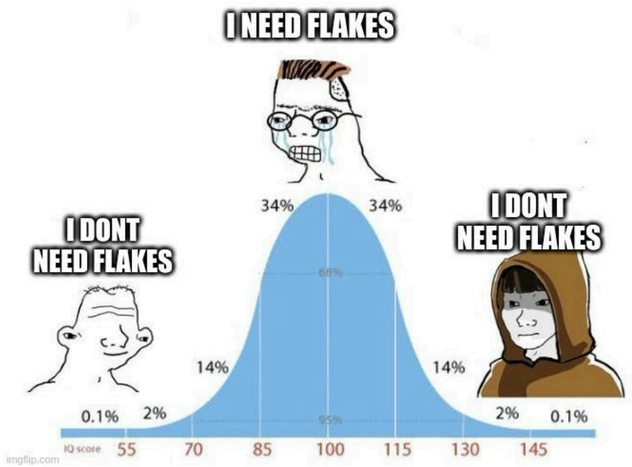
- Uses https://crane.dev
- Cached dependencies (*)
- Optional static linking
- Cross-compilation
- Pre-Commit checks
- Automatic dependency security audit
- Documentation Package
- Works on GNU/Linux and macOS

Trainer notes

Only for those who are interested

Flake tradeoffs & disadvantages

- Flakes are slow in huge monorepos
- Flakes are Git-aware (+/-)
- Flakes cannot be parameterized
- Flakes hardcode supported architectures
- `inputs.xxx.follows` is weaker than overlays



Trainer notes

A balanced overview of what advantages and disadvantages flakes have is crucial for every developer.

Especially because projects can support “dual use” flakes and non-flakes workflows when engineered correctly (e.g. by defining all packages and tests in an overlay and letting both worlds consume those overlays).

Some of our customers *stopped* using flakes for developer workflows after years.

Section: NixOS

About NixOS

Installing NixOS

NixOS Modules

NixOS Configuration Examples

The `nixos-rebuild` Command

NixOS Containers

NixOS Patterns

NixOS Integration Tests

Trainer notes

About NixOS

Trainer notes

NixOS: A Master Thesis that Escalated 👩‍🎓

NIXCADEMY EXPERT EDUCATION

"Builds would sometimes fail after updating the underlying system and that would be hard to reproduce and debug: *Don't touch or it will break*"

tiny.cc/@ackwil.com

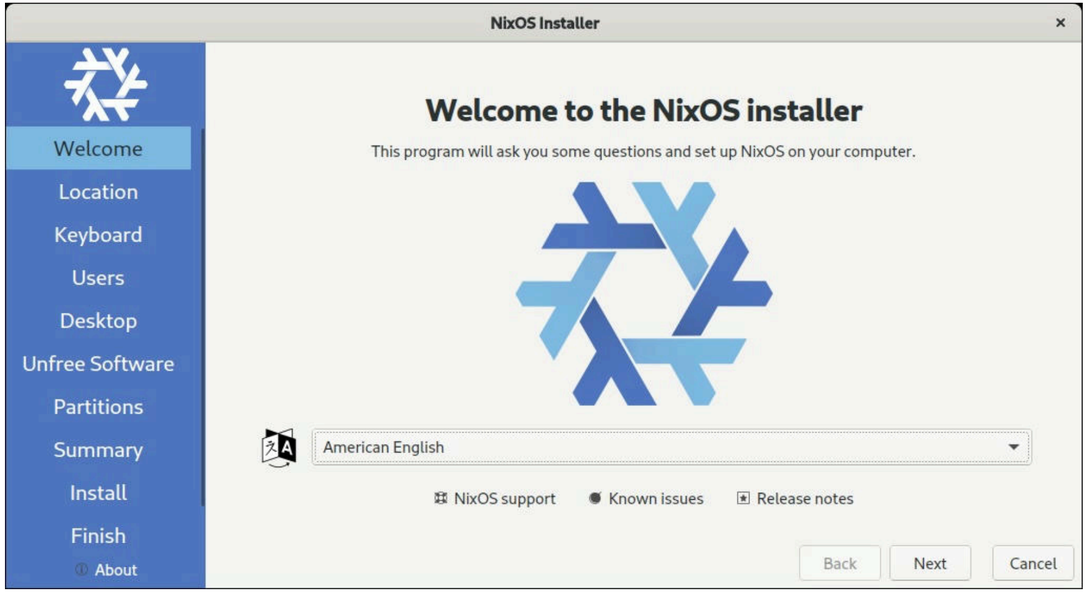
Trainer notes

It makes sense to have watched this talk and describe the mentality if Armijn how he tackled creating NixOS and especially why this project existed at all. NixOS solved some big problems for the Nix researchers and the same way it solves big problems for everyone who uses it now.

Up until now, we just built dead packages in the nix store. But you can't boot a nix store. So NixOS is all about copying files to real partitions and making packages ready to compose complex "living" systems.

Installing NixOS

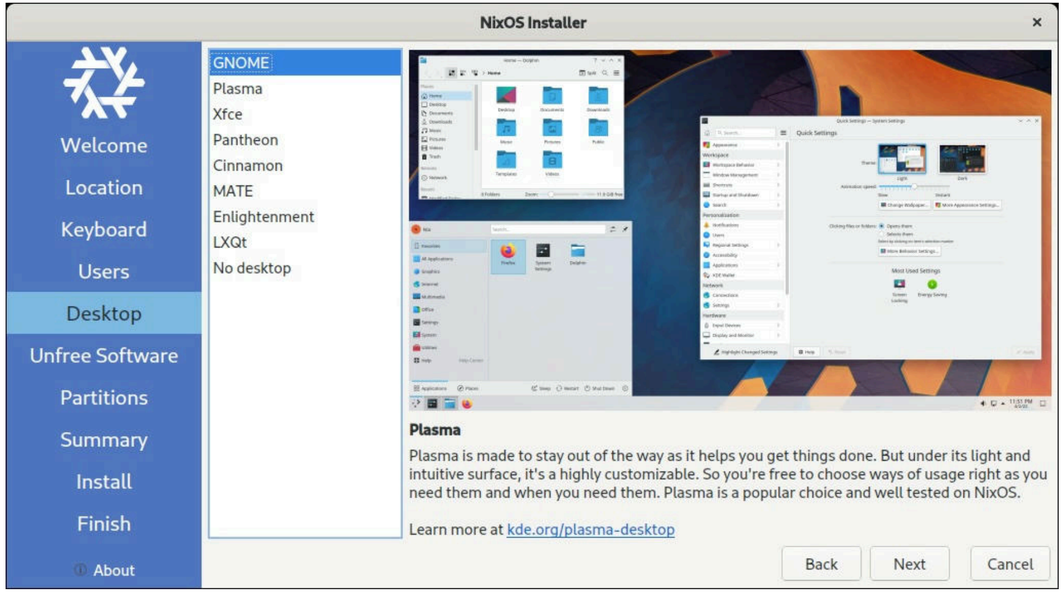
Trainer notes



Trainer notes

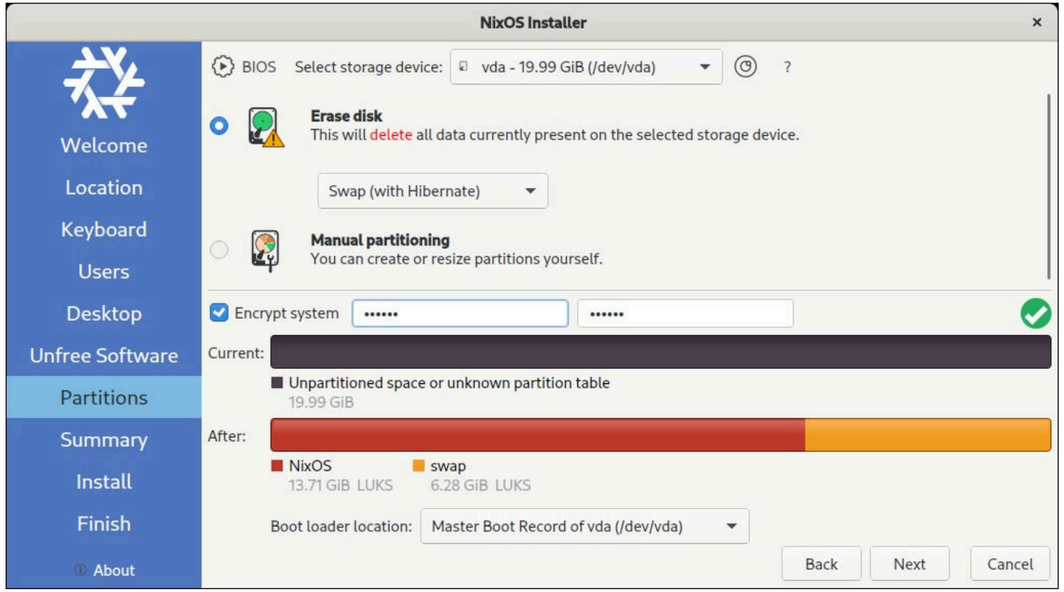
The next slides are just about showing that starting at the primitive state NixOS once was in, it is now a fully fledged systemd based distro with a graphical installer and the biggest package collection in the world.

The Graphical NixOS Installer



Trainer notes

The Graphical NixOS Installer



Trainer notes

The official download page provides multiple ways to install or try NixOS:
https://nixos.org/download.html#nixos-iso

- ISO image: Graphical and Minimal Variant
- VM as Virtualbox `.ova` file
- Amazon EC2 Image for Launch in Cloud

We will see different ways to install NixOS. Let's see what they all have in common.

Trainer notes

Without extra tooling, NixOS provides downloads for different image formats. That is exactly the kind of support we will be able to provide with our own images, too.

The NixOS Install Dance 👯

1. **Boot Live System**
   - Set up network
2. **Partition, Format & Mount Disk**
   - Use your favorite tools
3. **Create Initial System Config**
   - `nixos-generate-config`
4. **Install**
   - `nixos-install`
   - Generates *Top-Level Closure*
   - Runs Install Script of Top-Level Closure

See also NixOS Manual, Chapter 2 Installing NixOS

Trainer notes

These are the phases that are also similar to all the distros, although this is mostly automated away from the user.

Arch and Gentoo Linux might be pretty close to this.

The biggest difference here is that you can actually spin off from an existing NixOS config from some Git repo and end up with an identical system.

NixOS Minimal Installer Manual

https://nixos.org/manual/nixos/stable/index.html#sec-installation-manual-partitioning

Tasks:

- Let's create a NixOS VM from scratch to learn about the UX better
- Let's change the config and run `sudo nixos-rebuild switch`
- Rebuild into *older* generations

Notes:

- Only follow the UEFI *or* MBR/BIOS steps, according to your VM
- Mind the disk space: Choose e.g. at least 20GB
- You can skip/shrink the suggested 8GB swap partitioning to save space

Trainer notes

It seems like most folks nowadays have never partitioned disks themselves, which leads to some lack of foundation regarding what we are going to automate with Nix.

Ideally, present how to install NixOS with the minimal terminal installer, which should just take 20 minutes with all explanation and download times. Start with a minimal installation and then crank it up with a desktop after installation, boot back into the non-desktop generation etc. so folks get a feeling how to administer NixOS systems.

Prepare to draw partitioning schemes on the whiteboard if folks don't know what MBR/EFI is.

Trainer notes

NixOS Modules

How Do NixOS Configurations Look Like?

Let's first have a look at configurations in general.

Then, we write our own.

Trainer notes

A NixOS Configuration: configuration.nix

```
1 { config, pkgs, ... }:
2 {
3   imports = [
4     ./hardware-configuration.nix
5   ];
6
7   networking.hostName = "jongepad";
8   networking.networkmanager.enable = true;
9
10  environment.systemPackages = with pkgs; [
11    git
12    vim
13    wget
14  ];
15
16  # ...
17
18  system.stateVersion = "22.05"; # This needs some discussion
19 }
```


Trainer notes

Folks have already seen this in the NixOS installation, but explain a bit what the parameters in line 1 mean, what the imports line does, what the significance of the other configuration lines is (meaning that they both install *and* configure stuff, which other distros don't do like this).

Only explain this roughly as we will go more into detail later anyway.

The system.stateVersion variable needs some explanation. It's about preparing important migration routines, not about upgrading the system. If folks don't understand, the most important part is that this is supposed to be left unchanged for every system after initial installation.

A NixOS Configuration: hardware-configuration.nix

```nix
{ lib, modulesPath, ... }:
{
  imports = [ (modulesPath + "/installer/scan/not-detected.nix") ];

  boot.initrd.availableKernelModules =
    [ "xhci_pci" "nvme" "usb_storage" "sd_mod" "sdhci_pci" ];
  boot.initrd.kernelModules = [ "dm-snapshot" ];
  boot.kernelModules = [ "kvm-intel" ];

  fileSystems."/" = {
    device = "/dev/disk/by-uuid/6fdeb208-6446-4943-9a7d-e8a5d78ade50";
    fsType = "ext4";
  };
  fileSystems."/boot" = {
    device = "/dev/disk/by-uuid/091D-4B27";
    fsType = "vfat";
  };

  powerManagement.cpuFreqGovernor = lib.mkDefault "ondemand";
}
```


Trainer notes

This is just a random example for hardware-specific NixOS configuration that is necessary on the given platform.

Explain that it is important to separate into “application specific” and “platform specific” configurations so that the same application specific configuration can be run on different platforms by composition of different NixOS modules.

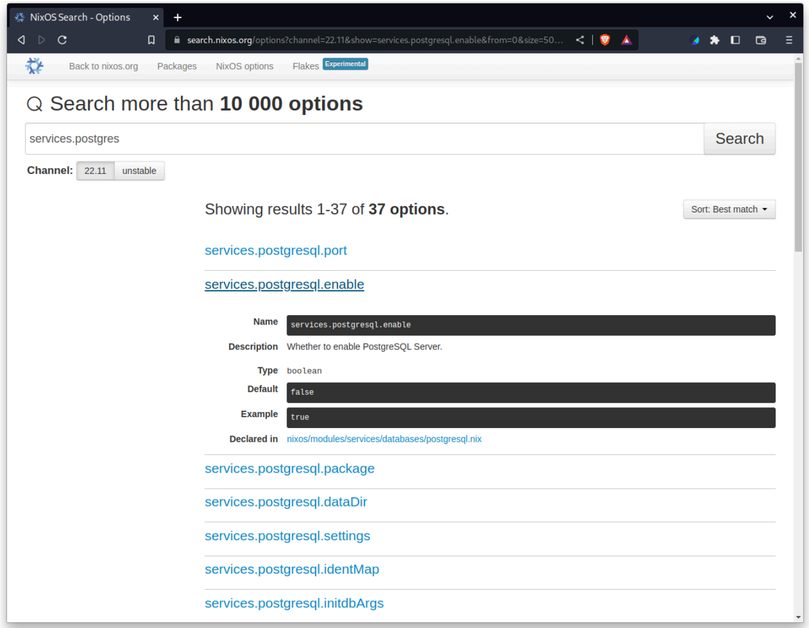
All NixOS Modules Always

```
{ config, pkgs, ... }:
{
  imports = [
    # paths of other modules
  ];

  attribute.path = this;
  other = {
    path = that;
  };
}
```

Where do we know these *magic* attribute paths from?

Trainer notes

Shopping for NixOS Settings: search.nixos.org



Trainer notes

Ask “what services do you typically configure?” and take a random example to show how to search for it.

Here it is important to grasp that we need some NixOS knowledge to use that in NixOS, but also *per service* need the right domain specific knowledge about this service, otherwise we wouldn’t know what the attributes even mean.

NixOS configuration attributes hide the “real” configuration from us, which means that nix is the one language to “master all configuration”.

NixOS Configuration Examples

Trainer notes

Immutable User Accounts

```
{
  users.users.alice = {
    isNormalUser = true;  # default
    home = "/home/alice"; # default
    description = "Alice Foobar";
    extraGroups = [
      "networkmanager"
      "wheel"
    ];
    openssh.authorizedKeys.keys = [
      "ssh-dss AAAAB3Nza... alice@foobar"
    ];
  };
}
```


Trainer notes

Users are typically *not* added imperatively. This is still possible but can (and should most of the time) be deactivated.

It's most interesting that you can set SSH keys for all users on remote machines from the beginning and completely disable password based logins for everyone.

Emulating other Platforms

```
{
  boot.binfmt.emulatedSystems = [
    "aarch64-linux"
    "wasm32-wasi"
    "x86_64-windows"
  ];
}
```

Run foreign binaries in your shell!

NixOS did not invent this feature - but it's the only distro where it's so easy to use!

Trainer notes

Trainer should know what binfmt is and why it's cool (https://en.wikipedia.org/wiki/Binfmt_misc) and be able to explain why it's easier to use than on other distros.

Most notably if you run a foreign executable (e.g. intel executable on ARM) legacy distros don't have `/usr/lib64` filled with the right libraries and then you need at least an overlay filesystem just for that. Binaries built with Nix are hardcoded to specific nix store paths which don't interfere with each other.

```
{
  boot.kernelPatches = [
    {
      name = "custom-config";
      patch = ./my-patch-file.patch;
      extraConfig = ''
        DEBUG_INFO y
        ENABLE_MY_FUNCTION y
      '';
    }
  ];
}
```

Note how patch-specific config snippets are organized with the patch.

Composition: Kernel version and more patches can be set in other modules.

Trainer notes

- Kernel patches can be defined in different modules and composed together
- config snippets are stored together with the patch so you don't have a huge config where it's unclear which line exists because of which patch
- Kernel version can be set somewhere else and it still all works nicely together

More Examples Online

https://galowicz.de/2023/03/13/quick-vms-with-nixos/
- How to quickly set up a VM and extend it

https://galowicz.de/2023/04/05/single-command-server-bootstrap/
- Automate partitioning, formatting, mounting, and installing NixOS on foreign resources

https://nixcademy.com/2023/09/26/nixos-openstreetmap/
- Run OpenStreetMap on NixOS

https://nixcademy.com/2023/09/04/nixos-multi-php-version-container/
- Run Nginx + multiple PHP versions on NixOS

Trainer notes

These examples are for further reference:

- The "quick vms" stuff is part of the next exercises anyway
- The trainer ideally does/did a nixos anywhere demo in the week anyway
- The open street map is a bigger real life example that shows how to nicely encapsulate a relatively complex service
- Multi PHP version container is cool because it's so easy with NixOS. It's also a nice showcase for NixOS containers

The `nixos-rebuild` Command

Trainer notes

Remote Deployment with `nixos-rebuild`

Build and/or deploy on/to different host

nixos-rebuild example

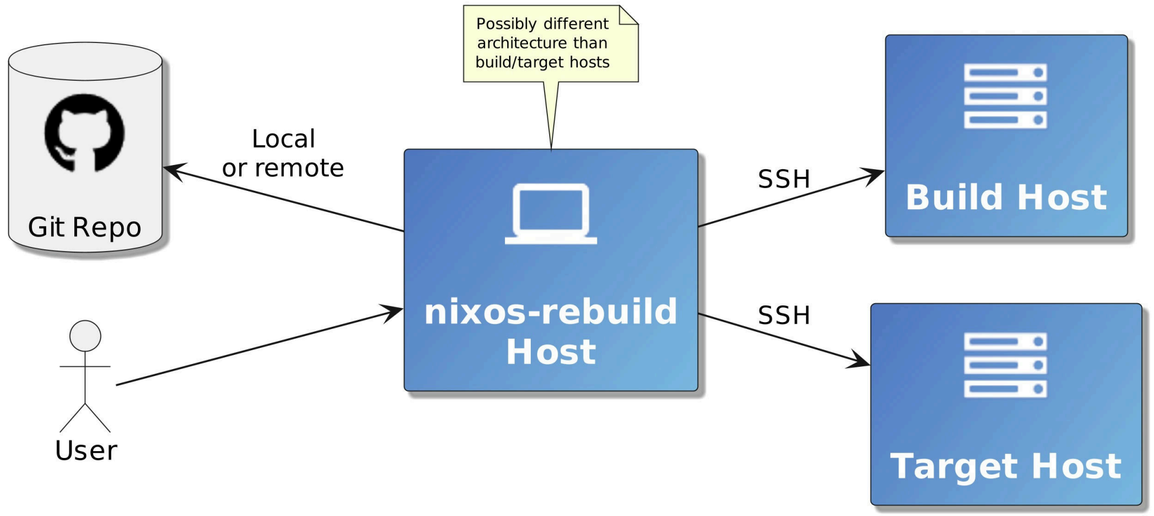
```
nixos-rebuild \
  --flake .#mySystem \
  --build-host builduser@buildhost \
  --target-host deployuser@deployhost \
  --sudo \
  --use-substitutes \
  switch
```


Trainer notes

The next slides explain `nixos-rebuild` in a bit more detail. Goal is to show folks that they *don't* need one of the many specialized deploy tools out there for most cases.

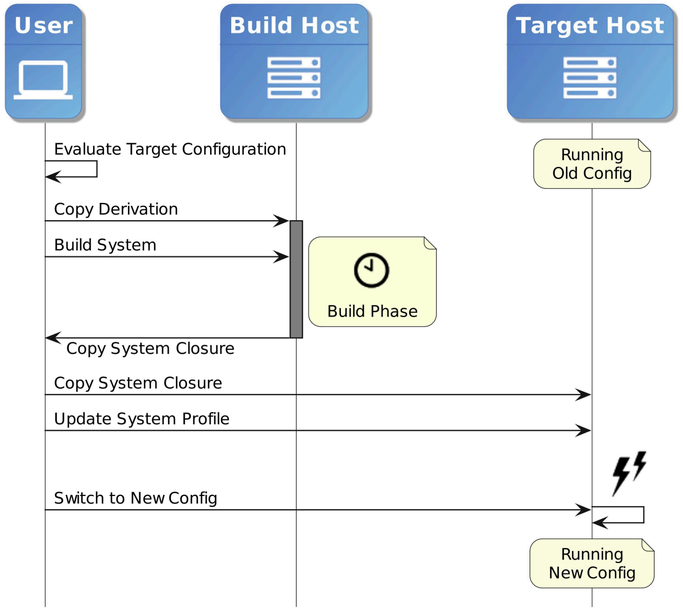
Explain what nixos-rebuild can do remotely and locally for folks on this and the next slide.

NixOS and Remote Builds and/or deployment



Trainer notes

Remote Deploy Sequence with `--target-host`



Trainer notes

What's cool is that no matter how complex a system switch is and how long it takes building/transfering it: The running system is only switched over at the very last moment and if that doesn't work we can always roll back.

Participants should be able to understand with a bit of explanation how to do the individual steps on the diagram manually (roughly) if they had to.

`nixos-rebuild` under the hood

1. `nix build .#nixosConfigurations.mySystem.config.system.build.toplevel`
2. Copy the result to the target host with `nix copy`
3. `nix-env -p /nix/var/nix/profiles/system --set <toplevel storepath>` (on the target host)
4. `sudo <toplevel storepath>/bin/switch-to-configuration switch`

Trainer notes

nixos-anywhere is a huge shell script but in the end it only does those 4 steps.

Everyone who understands those could build it themselves or knows how to adapt it to their needs.

All the other deployment tools also roughly work like this. This understanding will help selecting the right deployment tool.

NixOS Containers

Trainer notes

Declarative NixOS in NixOS Containers

NIXCADEMY EXPERT EDUCATION

Not Containerized

```
{ config, pkgs, lib, ... }:
{
  imports = [
    ./some-service.nix
  ];
}
```

Containerized

```
{ config, pkgs, lib, ... }:
{
  containers.webserver = {
    autoStart = true;
    privateNetwork = false;
    config = { config, pkgs, ... }: {
      imports = [
        ./some-service.nix
      ];
    };
  };
}
```


Trainer notes

NixOS containers are based on systemd-nspawn containers. This is much cooler than Docker because Docker are single-process containers whereas systemd based containers have their own complete systemd process tree.

You can login into systemd containers, get fine-grained control, etc. etc.

NixOS didn't invent systemd containers but makes them really easy to use. The example shows the nice matroszka-like composability.

The multi-PHP-version blog article nicely showcases this, also with screenshots of the systemd-status views with and without containers.

Imperatively start and maintain NixOS containers

```
$ nixos-container create foo --config '
  services.openssh.enable = true;
  users.users.root.openssh.authorizedKeys.keys = ["ssh-dss AAAAB3N…"];
'
$ nixos-container create test --config-file test-container.nix \
  --local-address 10.235.1.2 --host-address 10.235.1.1

$ nixos-container start foo
$ systemctl status container@foo
$ nixos-container root-login foo
$ nixos-container destroy foo
```


Trainer notes

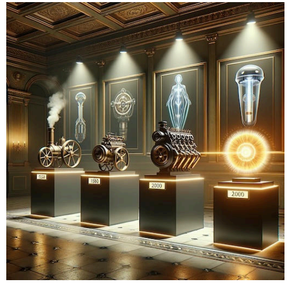
Last slide showed how to use NixOS containers declaratively. But we actually don't need NixOS, there's also `nixos-container` which works on any Linux distro with systemd.

Don't explain in too much detail here, only if folks are interested.

But *this* is the way to quickly get a nice LAMP stack running for development for example.

Blog: systemd containers

Running NixOS from any Linux Distro in systemd-nspawn Containers
https://nixcademy.com/posts/nixos-nspawn/



Multi-Version PHP with Nginx on NixOS: Containerized and Not
https://nixcademy.com/posts/nixos-multi-php-version-container/

Trainer notes

NixOS Patterns

Trainer notes

NIXCADEMY EXPERT EDUCATION

Extended Module Structure

```nix
{ config, pkgs, ... }:
{
  imports = [
    # paths of other modules
  ];

  options = {
    # option declarations
  };

  config = {
    # option definitions
  };
}
```

See also NixOS Manual, Chapter 67 Writing NixOS Modules

Trainer notes

So far we only used modules that have a function header, an `imports` clause and then random attributes set.

This structure here is the one we need to adhere to if we write our own modules that provide new configuration options for others.

`options` is the *declarative* part that describes which options we provide.

The `config` part describes what happens with them if the user actually sets them.

We look at everything in detail now.

NixOS Module Option Structure

NIXCADEMY EXPERT EDUCATION

Option Structure

```
options = {
  name = mkOption {
    type = # type specification
           # one of pkgs.lib.types.*
    default = # default value
    example = # example value
    description = "cool explanation";
  };
};
```

Automatic docs use these descriptions, too

See also: NixOS Manual, Chapter 67.1 Option Declarations
NixOS Manual, Chapter 67.2 Options Types

Trainer notes

Options have names, types, defaults, examples, descriptions.

It is cool to go in the repl and show the type description of types like `pkgs.lib.types.port` and then `pkgs.lib.types.listOf pkgs.list.types.port` etc. to showcase how they can be magically combined.

Make sure to mention that all this information is used to generate the content of pages like search.nixos.org and you can do that for your own internal modules, too!

NIXCADEMY EXPERT EDUCATION

`mkEnable` Helper

```
lib.mkEnableOption "the ultimate service"

# is equivalent to:

lib.mkOption {
  type = lib.types.bool;
  default = false;
  example = true;
  description = "Whether to enable the ultimate service";
}
```

See also 67.1.1. Utility functions for common option patterns

Trainer notes

This will be useful in the exercise later.

NixOS Module Options are Monoids

File: `module-a.nix`

```
{ ... }:
{
  a.b.c = [ 1 ];
}
```

File: `module-b.nix`

```
{ ... }:
{
  a.b.c = [ 2 ];
}
```

File: `top-module.nix`

```
{ ... }:
{
  imports = [
    ./module-a.nix
    ./module-b.nix
  ];
  a.b.c = [ 3 ];
}
```

Resulting Configuration

Trainer notes

What happens if multiple modules set the same attribute? Let folks guess what happens in the final configuration.

It depends on the type. Let's discuss how lists can be regarded as a "monoid" with a concatenation operation and a neutral operand for concatenation (empty list). Numbers are monoids to multiplication and addition, too. Let folks guess the neutral elements.

Whole NixOS modules are also monoids with a merging operation - this is exactly what `imports` does: It merges modules! And merging can only work if merging is defined for overlapping attribute path types, too.

All this weird theoretical functional programming stuff actually helps working practicioners by allowing them to compose systems of different modules!

NixOS Module Options are Monoids

File: `module-a.nix`

```
{ ... }:
{
  a.b.c = [ 1 ];
}
```

File: `module-b.nix`

```
{ ... }:
{
  a.b.c = [ 2 ];
}
```

File: `top-module.nix`

```
{ ... }:
{
  imports = [
    ./module-a.nix
    ./module-b.nix
  ];
  a.b.c = [ 3 ];
}
```

Resulting Configuration

```
{
  config.a.b.c = [ 1 2 3 ];
}
```


Trainer notes

What happens if multiple modules set the same attribute? Let folks guess what happens in the final configuration.

It depends on the type. Let's discuss how lists can be regarded as a "monoid" with a concatenation operation and a neutral operand for concatenation (empty list). Numbers are monoids to multiplication and addition, too. Let folks guess the neutral elements.

Whole NixOS modules are also monoids with a merging operation - this is exactly what `imports` does: It merges modules! And merging can only work if merging is defined for overlapping attribute path types, too.

All this weird theoretical functional programming stuff actually helps working practicioners by allowing them to compose systems of different modules!

Our First Own NixOS Module

```nix
{ config, pkgs, lib, ... }:
let cfg = config.services.hello; in
{
  options.services.hello = {
    enable = lib.mkEnableOption "GNU-Hello as a Service";
    port = lib.mkOption {
      type = lib.types.port;
      default = 8081;
      description = "Port to listen on";
    };
  };
  config = lib.mkIf cfg.enable {
    systemd.services.hello = {
      wantedBy = [ "multi-user.target" ];
      serviceConfig.ExecStart = ''${pkgs.socat}/bin/socat \
        TCP4-LISTEN:${builtins.toString cfg.port},reuseaddr,fork \
        EXEC:${pkgs.hello}/bin/hello
      '';
    };
  };
}
```


Trainer notes

It is imporant to explain this module file line by line because we are going to use it in the next exercise.

Trainer must know how socat roughly works, otherwise might not look good while explaining it.

Exercise Time: Run & Extend the Hello aaS 👋

NIXCADEMY EXPERT EDUCATION

Download 💾

https://nixcademy.com/downloads/hello-as-a-service.zip

Tasks:

- Run the VM: `$(nix-build)/bin/run-nixos-vm`
- Inspect the shell script, answer this question:
  - 👉 Why is rebuilding the whole VM after config changes so quick?
- Enable the firewall
- Add an option: `services.hello.openFirewall`

Notes:

- Look up `networking.firewall` on https://search.nixos.org
- Set `virtualisation.graphics = false;` to run on training-machine.nixcademy.com
- Solution: https://nixcademy.com/downloads/hello-as-a-service-solution.zip

Trainer notes

In general folks find this exercise relatively easy in hindsight but during the exercise they are confused.

Do a thorough explanation of all the modules before everyone goes on their own. Also, explain folks that the graphics option can be set to `true` on KVM enabled laptops but not on the training machine.

Super important: Show and explain the run script. Why is building the VM so fast? It does not have a full image, it just uses the host's Nix store!

Ideally, trainer creates the target configuration.nix visible for everyone and then hints with `# ?` lines in the hello-as-a-service.nix file where to edit it to solve the exercise to give folks some minimal structure to start with.

The pkgs.nixos Function

NIXCADEMY EXPERT EDUCATION

The nixos Function

```
let
  pkgs = ...;
  nixos = pkgs.nixos [
    module1 module2 module3 # ...
  ];
in {
  inherit (nixos.config.system.build)
    toplevel # always available
    isoImage # modulesPath + "/installer/cd-dvd/iso-image.nix"
    vm       # modulesPath + "/virtualisation/qemu-vm.nix"
    # whatever the nixos config exports
    ;
}
```

Endless Possibilities:
Images for Amazon, Azure, Google Cloud Engine, VirtualBox, kexec, netboot, KubeVirt, LXC, Proxmox, CloudStack, many more, and *your own*!

Trainer notes

Emphasize how the separation between application specific and platform specific modules helps creating different modules for the same applications.

Also, showcase some of the profiles in nixpkgs/nixos/modules/virtualisation.

There's often aha-effects when folks see modules/profiles for their favorite virtualization/container solutions. You don't know how to use them all but explain how you would start if you had to learn about one of them.

⚠️ NixOS Module Rules 1/2 ⚠️

NIXCADEMY EXPERT EDUCATION

- Modules must not know the paths of other modules
  - Exception: *Internal* module imports of modules that are split up in sub-modules for structure
- The import of a module must always be free of side-effects
  - i.e. modules only export their options by default
  - Modules with side-effects are called *profiles*
- Do not add overlays or nixpkgs configuration changes in a module
  - Consider providing `services.myservice.package` attrs
- The final NixOS configuration should pick all the imports and nixpkgs overrides
  - Composition over inheritance ☝️

These rules are for **module writers**.
Consumers may do whatever they want in final NixOS configs.

Trainer notes

These rules sound a bit synthetic for everyone who has no experience building big piles of modules, yet.

However, these rules are typically unknown to folks who end up with projects that are full of structural disadvantages that cancel out the nice things that NixOS was supposed to bring us.

If there's time, give an example on how to add a `services.hello.package` option to our service and override it with some patch in `configuration.nix` to show how easy this is when you're somewhat used to module syntax.

⚠️ NixOS Module Rules 2/2 ⚠️

NIXCADEMY EXPERT EDUCATION

🚫 The most important and dangerous antipattern regarding modules 🤦

bad example

```
# File: `my-module.nix`
some: parameters:
# bad: taking "pre-parameters"
#      in NixOS modules

{ config, pkgs, lib, ... }:
{
  # using "some" and "parameters"
}
```

bad example

```
# File: configuration.nix`
{ config, pkgs, lib, ... }:
{
  imports = [
    (import ./my-module.nix
      somePkg
      parameterExpr
    ) # bad antipattern
  ];
}
```

Instead, do:

- Inject values via new module options ✅
- Use `nixpkgs.overlays` for package level overrides ✅
- Use composition instead of inheritance for module additions ✅

Trainer notes

Folks are often tired from too much new information already when we are showing this.

It's really important that folks know that it's generally **bad bad bad** if a NixOS module takes parameters upfront before returning a module.

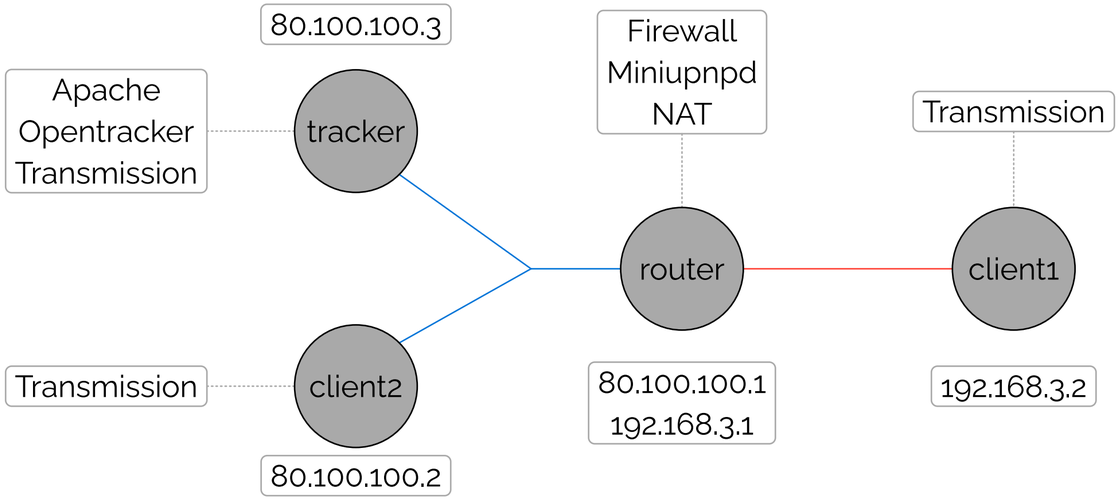
You cannot get rid of these parameters any longer! But you could if you followed the rules from the last slide and added proper options for example.

Trainer notes

NixOS Integration Tests

NixOS Integration Test Example: Bittorrent Service

Complex Network & Service Definitions in ~50 LOC



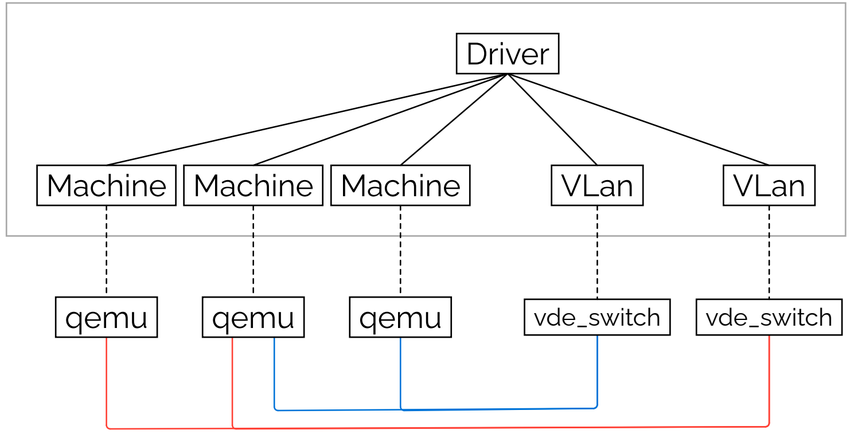
Trainer notes

Please note that many young folks don't know what bittorrent is any longer, so ask before you explain anything from this slide.

This is an awesome example. Trainer should have read the original test file in nixpkgs/nixos/tests/bittorrent.nix and be able to explain what test steps are actually performed, because this is an amazing demo.

Most folks have never seen how to write such a complex test with only so few code lines.

The NixOS Integration Test Driver Architecture



Trainer notes

This is the overall architecture of a running test.

It's important to understand that all the python code is typically from the perspective of the driver object. At this point we can literally play "god" by manipulating machines and networks directly.

The qemu processes

What are the VMs built of?

```
node.config.system.build.vm
```

We already know why it's so **quick** to rebuild and rerun 4 different VMs!

Trainer notes

In the first VM exercise we should have had a look at how the VM is started and why it is so slick (has no own image but basically has access to host store).

This is also why it's so quick and easy to change a few VMs and have a slightly different network configuration running in less than a minute.

Structure of a NixOS Integration Test

NIXCADEMY EXPERT EDUCATION

File: `old-way.nix`

```nix
let pkgs = import <nixpkgs> { };
in pkgs.nixosTest ./test.nix
```

File: `new-way.nix`

```nix
let pkgs = import <nixpkgs> { };
in
pkgs.testers.runNixOSTest {
  imports = [
    ./test.nix
  ];

  defaults = {
    networking.firewall.enable = false;
  };
}
```

File: `test.nix`

```nix
{
  name = "My NixOS Test";

  nodes = {
    machine1 = { ... }: { ... };
    machine2 = { ... }: { ... };
  };

  testScript = ''
    start_all()
    machine1.succeed(
      "ping -c 1 machine2"
    )
    machine2.succeed(
      "ping -c 1 machine1"
    )
  '';
}
```

See also NixOS Manual, Chapter 71 NixOS Tests

Trainer notes

It's important to have read the documentation of the tests as a trainer and be able to understand the general concept.

Hint about old-way.nix vs new-way.nix:

- new way enables *module system* for test file compositon
- new way only evaluates `pkgs` once and injects it read-only into all VMs. This improves eval speed. Can be disabled for indidivual overlays.

Folks might ask: Why is `runNixOSTest` not used in nixpkgs? Reason is mostly legacy but the documentation points this out, too. Outside of nixpkgs, use `runNixOSTest`.

More Interesting `runTest` Parameters

```
pkgs.testers.runNixOSTest {
  imports = [ ... ];
  defaults = { ... };

  skipLint = true;
  skipTypeCheck = true;
  extraPythonPackages = p: [ p.numpy ];
}
```

All options are defined in module files in `nixpkgs/nixos/lib/testing/*`

See also NixOS Manual, Chapter 71.1.6 Test Options Reference

Trainer notes

There are many folks who rather patch out linting and type checking to have faster iteration cycles instead of reading the docs and finding out that this can be disabled.

Also, extra python packages can be added.

Folks often ask how to write these tests in a real IDE with syntax highlighting, python language server etc:

1. write a test library that has a function `def test(m1: Machine, m2: Machine): -> None`
2. add this library to `extraPythonPackages`
3. import and call it in your test

NixOS Test Machine Objects

NIXCADEMY EXPERT EDUCATION

The `Machine` class can do many things

```
machine.start()
machine.wait_for_unit("my-systemd-unit.service")
machine.wait_for_open_port(123)

output = machine.succeed("command that shall succeed")
assert "foobar" in output, "command outputs the right things"
machine.fail("command that shall fail")

@polling_condition
def my_service_running():
    machine.succeed("pgrep -x my-service-process")

with my_service_running:
    machine.succeed("this takes time and may crash the service")
    machine.succeed("that might make it go OOM")
```

All these methods and *many more* are implemented in `nixpkgs/nixos/lib/test-driver/src/test_driver/machine/`

See also NixOS Manual, Chapter 71.1.3 Machine Objects

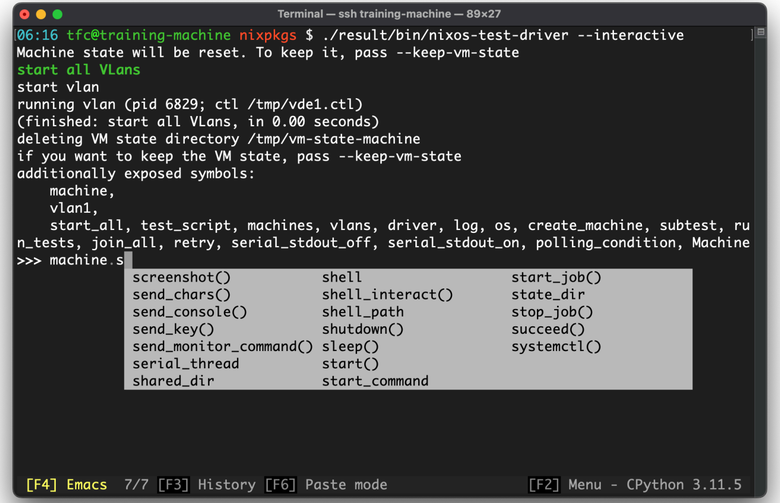
Trainer notes

This is just to provide an overview. It might be overwhelming what's possible here because this is a whole new library, but we will play with it step by step in the next exercise anyway, so don't worry everyone.

Teacher should be able to explain what `polling_condition` does and motivate folks to read the docs so that folks find out the *full* potential of the test driver, instead of just hacking copy&paste style and get used to living below their potential.

Interactive Mode

```
$ nix build .#myTest.driverInteractive
$ ./result/bin/nixos-test-driver
```



Trainer notes

Demonstrate the interactive mode on some random test.

Folks might point out that this is a macOS window. Yes, the driver works on macOS but is often declared broken by changes of upstream maintainers who don't use macOS. It generally works.

VM Config only for debugging

NIXCADEMY EXPERT EDUCATION

Settings for interactive mode

```
# test.nix ...

interactive.nodes.machine1 = {
  services.openssh = {
    enable = true;
    settings = {
      PermitRootLogin = "yes";
      PermitEmptyPasswords = "yes";
    };
  };
  security.pam.services.sshd.allowNullPassword = true;
  virtualisation.forwardPorts = [
    { from = "host"; host.port = 2222; guest.port = 22; }
  ];
};

# ...
```


Trainer notes

This is huge: By only adding `interactive.` as a prefix to configuration lines, they are going to be ignored in non-interactive-mode builds and added in interactive mode.

So just keep tons of debuggability config in your tests, the production tests will run without it. Super useful.

Existing NixOS Integration Tests

>1000 integration tests exist in `nixpkgs/nixos/tests`.

These test partitioning, booting, apps on the desktop, take screenshots, perform OCR, run distributed services with sharding and redundancy, etc., etc…

Find the code in `nixpkgs/nixos/tests` and run from within the `pkgs.nixosTests` on different platforms.

Make sure to have a look at existing tests to see how others use this complex allrounder test framework.

Trainer notes

It's important that folks know that whatever weird test they need to develop, it's unlikely that noone has done this before in nixpkgs.

So make sure to inspire from the existing tests.

There's a nice nixcademy article (testing part 1) about some more test examples with screenshots. Show these if you like - many potential aha-effect unlocks.

Trainer notes

🚨 Important Notice for fast tests 🚨

The NixOS test driver needs write access to `/dev/kvm`:

```
$ lsmod | grep kvm
kvm                  1355776  1 kvm_amd
...
$ ls -lsa /dev/kvm
0 crw-rw-rw- 1 root kvm 10, 232 Aug  8 16:31 /dev/kvm
```

If the `nixbldXX` users don't have write access to `/dev/kvm`, virtualisation will still work but it will be **slow**!

Quick fix (not persistent over reboots):

```
$ sudo chmod 777 /dev/kvm
```

Exercise Time: Test Hello-aaS

NIXCADEMY EXPERT EDUCATION

Download 💾

https://nixcademy.com/downloads/nixos-test.zip

Task:

- Run the test using `nix-build`
- Read and understand the configuration files
- Copy `hello-as-a-service.nix` and `configuration.nix` from last exercise and use it
- Test the service meaningfully

Notes:

- Wait for a systemd service: `machine.wait_for_unit("bla.service")`
- Wait for a port to open: `machine.wait_for_open_port(123)`
- Solution: https://nixcademy.com/downloads/nixos-test-solution.zip

Trainer notes

Tell folks to unpack this test and copy `configuration.nix` and `hello-as-a-service.nix` from the last exercise to the new one as we're going to reuse these files.

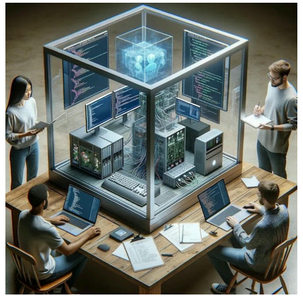
Often folks don't do that because they are not attentive and then when the exercise starts they ask where they get the service from or think they shall rewrite it. Make sure that everyone gets how to start.

Let them add a second VM so we actually test the firewall option.

Let folks use `nc <hostname> <port> < /dev/null` because the call would otherwise block forever. Draw it on a whiteboard, you will have to explain multiple times.

Emphasize the discussion why we need `wait_for_unit` *and* `wait_for_open_port`: if the systemd unit goes online, it does not guarantee that the port is open just yet (slow machines vs. fast machines flakiness). If we only wait for the port we get a very long timeout even if a service failed immediately due to some e.g. configuration mistake.

Blog: More about Integration Tests



Unveiling the Power of the NixOS Integration Test Driver (Part 1)
https://nixcademy.com/posts/nixos-integration-tests/

Unveiling the Power of the NixOS Integration Test Driver (Part 2)
https://nixcademy.com/posts/nixos-integration-tests-part-2/

Trainer notes

Blog: NixOS Integration Tests on macOS

Build and Deploy Linux Systems from macOS
https://nixcademy.com/posts/macos-linux-builder/

Run NixOS Integration Tests on macOS
https://nixcademy.com/posts/running-nixos-integration-tests-on-macos/

Trainer notes

Integration tests generally run on macOS but you need the linux builder as described in these articles.

The linux builder only builds the VMs but they are run on native macOS qemu binaries.

Section: Additional resources

Trainer notes

Exercise Time: Find and Fix Anti-Patterns 💪

Download 💾

https://nixcademy.com/downloads/anti-patterns.zip

Tasks:

- Study the code: it's full of anti-patterns!
- Discuss the disadvantages in a group
- Fix the anti-patterns
- Look into `hints.md` when you don't see any more anti-patterns

Notes:

- Solution: https://nixcademy.com/downloads/anti-patterns-solution.zip

Trainer notes

This is ideal for crowd-programming. Folks often end up really liking it.

NIXCADEMY EXPERT EDUCATION

The 3 Manuals:

Nix	https://nixos.org/nix/manual
Nixpkgs	https://nixos.org/nixpkgs/manual
NixOS	https://nixos.org/nixos/manual

Additional Resources:

Nix pills articles	https://nixos.org/guides/nix-pills
Official tutorials	https://nix.dev
NixOS Wiki	https://wiki.nixos.org
NixOS Forum	https://discourse.nixos.org
Chat	https://matrix.to/#/#community:nixos.org

Trainer notes

The folks in the forums and chat are typically most helpful if you explain how you failed first.

NIXCADEMY
EXPERT EDUCATION

Trainer notes

We work hard on being a first-class high-quality information resource.
Bookmark!

Monthly links + summaries of the Nix(OS) Blogosphere.

The *one* newsletter you need to subscribe to in order to stay up to date in the NixOS sphere.

Trainer notes

The newsletter regularly lists new articles from the blogosphere and provides a quick summary about the target group and content, each one sentence. Very useful to find what's interesting without wasting time.

https://fulltimenix.com

Trainer notes

Shahar's podcast is cool because he interviews folks who work fulltime on/with Nix. Often his guests are well-known maintainers of widely used Nix things.

Also, he regularly hosts Nix release note episodes with the original maintainers!