Mal Nix Machen

Jakob Gerhardt | Freitagsrunden Techtalks | 08.06.2023

Wer bin ich?

- Jakob
- Ersti im Master Computer Science
- verwende Nix seit gut nem Jahr

- Kontakt:
  - Matrix: @drruhe:matrix.org
  - Email: jak.gerhardt@gmail.com

It Works!

It Works

on my machine

It Worked

on my machine

but not anymore..

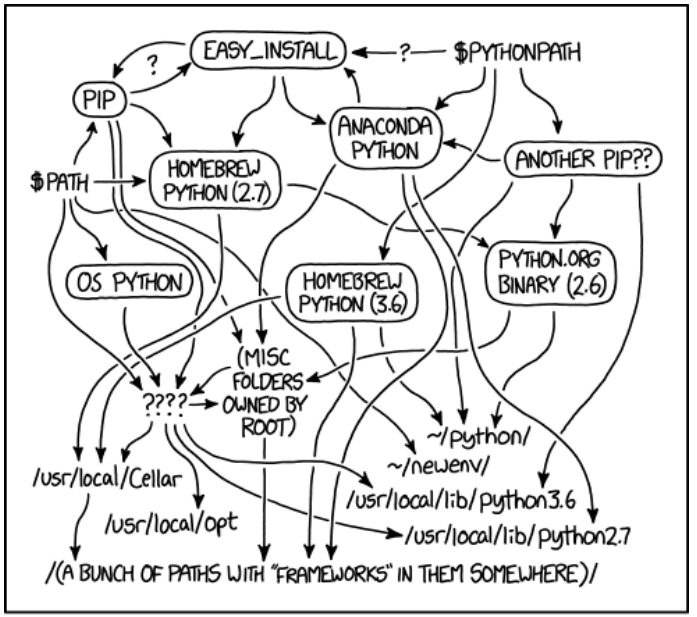
Der Grund

- jeder Computer ist unterschiedlich konfiguriert
- Annahmen über Systemzustand gelten nicht immer
  - Programm nicht installiert
  - User hat keine Berechtigung für eine Datei
  - Ordner existieren nicht
  - etc.

Das Resultat

Frankensteins Monster

- diverse installierte Versionen irgendwo im System
- überall Konfigurationsdateien
- man muss dauernd “operieren”



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Quelle https://xkcd.com/1987/

Was ist Nix?

Buildsystem + Programmiersprache + Paketmanager = Nix

Nix: Das Buildsystem

- Kompiliert Software, baut Dokumente, erstellt Konfigurationsdateien, etc
  - baut von den Quellen
- plattform- und architekturunabhängig
  - Linux (Distro unabhängig, auch WSL für Windows)
  - MacOS
  - x86, arm, powerpc, riscv, etc…
- Caches verhindern, dass ständig die Welt neu gebaut wird
- Reproduzierbarkeit durch Sandboxing
  - kein Zugriff auf nicht-deklarierte Abhängigkeiten
  - Downloads nur mit Checksumme möglich

Nix: Die Programmiersprache

- deklarative funktionale Programmiersprache
  - nicht: “installiere jetzt Python und dann Numpy”
  - sondern: “ich möchte Python mit Numpy haben”

→ **ist praktisch json mit Funktionen**

- Abstraktion und Komposition: man kann sich eigene Funktionen definieren
- Modifikation: man kann Paketdefinitionen anpassen
  - Compilerflags
  - eigene Patches
  - überschreiben von Dependencies

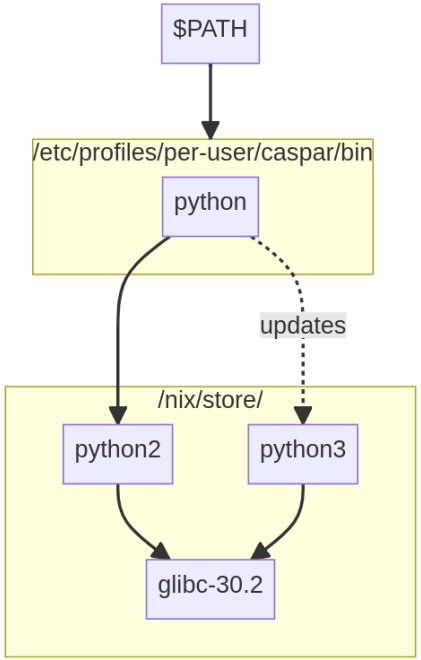
Nix: Der Paketmanager

- Pakete werden unter `/nix/store` installiert
  - mehrere Versionen paralell installierbar
  - über Symlinks oder Modifikation von `$PATH` erreichbar

```shell
~$ which typst
/nix/store/xibawahm3mzbc3h728vhi1m0qngn13kh-typst-0.4.0/bin/typst
```

- Einträge verweisen statisch aufeinander
  - benötigt also keinen File Hierarchy Standard mit `/bin`, `/etc`, `/lib`
- Atomare Updates/Rollbacks
  - Rollbacks durch austauschen der Symlinks

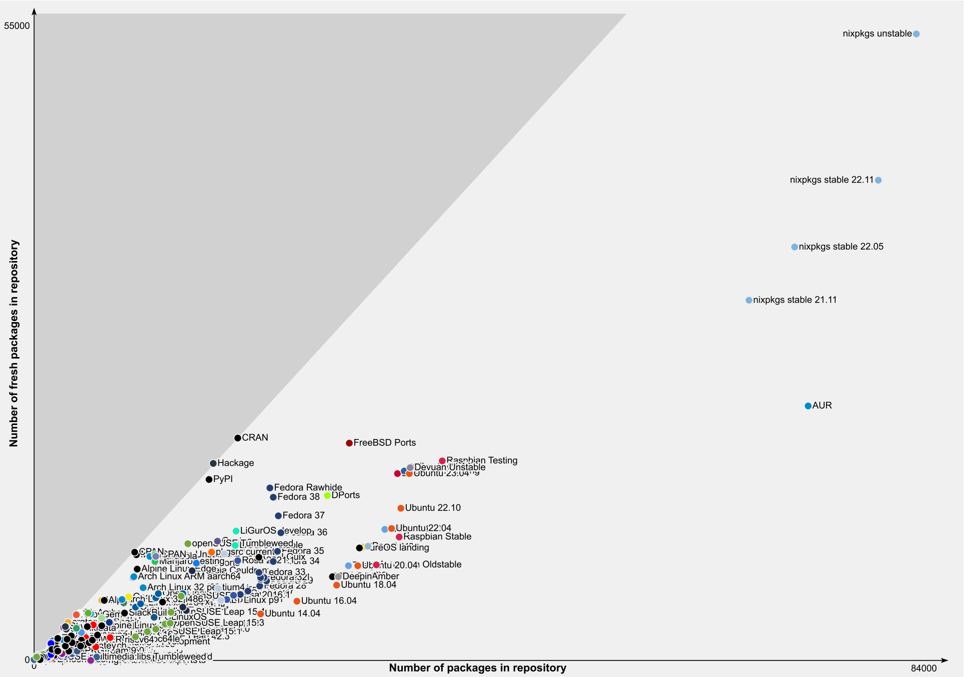
Nix: Der Paketmanager



Nixpkgs

- Zentrales Registry von Paketen mit from-source Bauanleitungen
- Fertige Build-Artefakte sind meist im Cache anzufinden
- Sourcecode wird mit Checksummen versehen
  - eine nixpkgs Version legt tausende Projektversionen fest

Nixpkgs ist groß



Quelle https://repology.org/repositories/graphs

Nixpkgs ist auch eine Library

- typische Optionen häufig nativ unterstützt
- enthält häufig verwendete Abstraktionen
- Erlaubt die Anpassung des Buildprozesses
  - Crosskompilierung
  - Statisches Linken
  - spezielle Compilerflags

Beispiel 1

```nix
{ pkgs ? import <nixpkgs> {},... }:
pkgs.mkShell {
  packages = [
    (pkgs.python3.withPackages (p:[
      p.numpy
      p.tensorflow
    ]))
  ];
}

```

Beispiel 1

```nix
{ pkgs ? import <nixpkgs> {},... }:
pkgs.mkShell {
  packages = [
    (pkgs.python3.withPackages (p:[
      p.numpy
      p.tensorflow
    ]))
  ];
}

```

- eine Shell mit Python 3
- mit Zugriff auf Numpy und Tensorflow

Shell betreten mit `nix-shell ./examples/python-env.nix`

Beispiel 2

```nix
{ pkgs ? import <nixpkgs> {},... }:
pkgs.mkShell (let
  fonts = pkgs.symlinkJoin {...}; // Deklaration der benötigten Fonts
in {
  packages = [ pkgs.typst pkgs.typst-lsp ];
  shellHook = ''
    export TYPST_FONT_PATHS=${fonts}
    echo Willkommen in der Shell zum Freitagsrunden Techtalk über Nix!
  '';
})
```

Beispiel 2

```nix
{ pkgs ? import <nixpkgs> {},... }:
pkgs.mkShell (let
  fonts = pkgs.symlinkJoin {...}; // Deklaration der benötigten Fonts
in {
  packages = [ pkgs.typst pkgs.typst-lsp ];
  shellHook = ''
    export TYPST_FONT_PATHS=${fonts}
    echo Willkommen in der Shell zum Freitagsrunden Techtalk über Nix!
  '';
})
```

- gibt mir eine Shell mit projektspezifischen Tools
- setzt auch Env Variablen und Willkommensnachricht
- betreten mit `nix-shell ./examples/devshell.nix`

Beispiel 3

```nix
{ pkgs ? import <nixpkgs> {} ,...}:
pkgs.dockerTools.buildLayeredImage {
  name = "hello";
  contents = [ pkgs.hello ];
  config = {
    Cmd = [ "${pkgs.hello}/bin/hello" ];
  };
}
```

Beispiel 3

```nix
{ pkgs ? import <nixpkgs> {} ,...}:
pkgs.dockerTools.buildLayeredImage {
  name = "hello";
  contents = [ pkgs.hello ];
  config = {
    Cmd = [ "${pkgs.hello}/bin/hello" ];
  };
}
```

- erstellt Docker-Container (ohne Dockerfile!)
- Image ist minimal
- somit sind Softwareversionen auch durch Nix festgelegt

Ein Rezept zur Reproduzierbarkeit

Wir brauchen:

- 250g aufgelistete Abhängigkeiten
- eine Hand voll festgelegte Versionen
- eine Prise Build-Skripte zum Bauen der Software

→ Gut vermengen und schon hat man ein reproduzierbares System

Offene Fragen

- Wo legt man Versionen fest?
- Wie organisiere ich den Nix Code?

⇒ Nix Flakes verwenden

Flakes

- Zwei Dateien `flake.nix`/`flake.lock` bilden eine Flake
- typischerweise eine Flake pro Projekt

Vorteile von Flakes

standardisiertes Schema

- ermöglicht Wiederverwendung
- Flake Libraries bieten Abstraktionen
- Tooling weiß damit umzugehen

ein Lockfile

- Lockfile ist in der Versionskontrolle
  - alle Teammitglieder/CI/Deployments verwenden dieselbe Umgebung

Flakes: Input Schema

```nix
{
  description = "Eine Nix Flake";
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
    flake-parts.url = "github:hercules-ci/flake-parts";
  };
  outputs = inputs: {...}
}
```

- Inputs sind per Lockfile festgelegt
  - Versionskontrolliertes Lockfile ermöglicht Rollbacks
- können komponiert werden

Flakes: Output Schema

```nix
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
  outputs = inputs: {
    packages.<system>.<name> = {...};  // Generischer Output
    apps.<system>.<name> = {...};      // Ausführbarer Output
    checks.<system>.<name> = {...};    // Tests
    devShells.<system>.<name>= {...};  // Entwicklungsumgebungen
    templates.<name>= {...};           // Vorlagen
    lib.<funktionsname>= args: {...};  // Funktionen
  }
}
```

- Outputs werden pro häufig pro Systemarchitektur definiert

Die Nix CLI verwendet Flakes:

```nix
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
  outputs = inputs: {
    packages.<system>.<name> = {...};  // nix build .#<name>
    apps.<system>.<name> = {...};      // nix run .#<name>
    checks.<system>.<name> = {...};    // nix flake check
    devShells.<system>.<name>= {...};  // nix develop .#<name>
    templates.<name>= {...};           // nix init -t .#<name>
  }
}
```

Library support für Flakes

Ich persönlich verwende und empfehle `flake-parts` und `systems` und `devenv`

nix

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
    systems.url = "github:nix-systems/default";
    flake-parts.url = "github:hercules-ci/flake-parts";
    devenv={
      url = "github:cachix/devenv";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };
  outputs = inputs:{...}
}
```

Library support für Flakes

```nix
{
  outputs = inputs@{ flake-parts,systems,devenv, ... }:
    flake-parts.lib.mkFlake { inherit inputs; } {
      systems = import systems;
      perSystem = { config, self', inputs', pkgs, system, ... }: {
        packages.<name> = {...};      // alle <system> spezifischen
        devenv.shells.<name> = {...}; // Outputs können so komfortabel
        apps.<name> = {...};          // für alle systeme
        checks.<name> = {...};        // konfiguriert werden.
      };
      templates.<name>= {...};           // Vorlagen und Libs sind
      lib.<funktionsname>= args: {...};  // weiterhin definierbar
  };
}
```

Werdet Nutzer:innen: Eure Ersten Schritte

- Nix installieren
- Flakes aktivieren

```sh
mkdir -p ~/.config/nix
echo "experimental-features = nix-command flakes" >> ~/.config/nix/nix.conf
```

Tipps von mir

- direnv installieren (Github: github.com/direnv/direnv, nixpkgs: direnv):
  - Automatisches betreten der Projektumgebung
- Stellt Fragen!
- Fordert bessere Dokumentation!
  - webbasierte Dokumentation (leider) spärlich/veraltet
  - Source lesen oft das beste Mittel…

Dann: Macht euer Umfeld reproduzierbar!

Dann endlich:

It works on any machine!

Ressourcen:

Eine Kollektion an Links:

Lernen

- Nix Pills
- Das Repo zu diesem Talk hat ein paar Beispiele/Templates gitlab.com/DrRuhe/techtalk-nix

Community Support

- Discord
- Matrix Channel
- Discourse
- NixOS Stammtisch auf der c-base

NixOS

- verwaltet das gesamte System mit nix
- baut die Konfiguration und symlinkt an die entsprechenden Stellen
- eine zentrale Systemkonfiguration bestimmt den Zustand
  - sofort einsatzbereit
  - furchtloses Ausprobieren
  - problemlose Rollbacks

Nix funktioniert wie es soll

Ich sehe Nix

Da kann man endlich mal Nix machen