Presentation Copy (Educational Use)

This copy may be used and shared for **educational purposes** free of charge.

Want to sponsor or support my work? You can buy my slide/material packs for a self-chosen price on Gumroad: https://wieerwill.gumroad.com/

github.c github.com/awesome-selfhosted/awesome-selfhosted tart.md

Dezentrales Internet durch Self-Hosting

DSGVO, Ausfälle, Preissprünge und fragwürdige AGBs: Wer zentrale Plattformen nutzt, bekommt Komfort… aber auch Abhängigkeiten. Self-Hosting ist der Versuch, diese Abhängigkeiten wieder gestaltbar zu machen. Dieser Linux-Info-Tag-Vortrag ist die erweiterte Praxisversion meines Easterhegg-Talks: mehr Architektur, mehr Nix, mehr konkrete Services. Vom ersten nginx auf dem Raspberry Pi bis zur Multi-Host-Flake mit NixOS, Home Manager, SOPS, Caddy, Backups und reproduzierbaren Deployments.

Created: 29 April 2026

LINUX INFO TAG 2026 · AUGSBURG

Dezentrales Internet durch Self-Hosting

Vom Raspberry Pi zur deklarativen Infrastruktur

Vom ersten Raspberry Pi bis zur reproduzierbaren Serverlandschaft

Self-Hosting ist keine Romantik

Es ist Dependency-Management für Erwachsene

Wer Abhängigkeiten nicht
gestaltet, bekommt sie trotzdem

Zentralisierung skaliert Komfort (und Ausfälle)

Ein Rechenzentrum reicht, damit sehr viele fremde Probleme plötzlich auch deine Probleme sind.

Bitwarden verdoppelt Preis

Cloud ist nur der Computer von jemand anderem

Und dessen Ausfall.
Und dessen Preiserhöhung.
Und dessen ToS.

Dezentral heißt nicht: alles zuhause

- Ein einzelner VPS ist **nicht** plötzlich dezentral.
- Ein Homeserver ist **nicht** automatisch souverän.
- Wichtiger sind: offene Protokolle, Datenexport, klare Zuständigkeiten.
- Danach kommt erst die Frage: Heimnetz, VPS oder gemischt?
- Gute Architektur ist oft hybrider als die Ideologie dahinter.
- **Aber:** Mehr Verteilung heißt auch mehr Zustände, mehr Fehlerbilder, mehr Arbeit.

Kapitel 01

Die Story bis zum Easterhegg

unspektakulär: eine statische Website zuhause

30€

ungefähr

Raspberry Pi 2 + Strom + viel Neugier

- Erste Website auf einem Raspberry Pi zuhause.
- Statisches HTML + CSS, kein Framework, kein Build-Zirkus.
- DynDNS gegen die wechselnde Heim-IP.
- Wenig Software bedeutete: wenig Überraschungen.
- Für eine persönliche Seite war das völlig ausreichend.
- **Aber:** Heimnetz, Upload, Router und Backups waren jetzt mein Problem.

Stufe 1: nginx nativ auf Raspbian oder Ubuntu

```
// install ubuntu/raspbian
sudo apt update
sudo apt install -y nginx

sudo mkdir -p /var/www/wieerwill.dev
printf '%s\n' '<h1>Hallo Internet</h1>' \
  | sudo tee /var/www/wieerwill.dev/index.html

sudo tee /etc/nginx/sites-available/wieerwill.dev >/dev/null <<'EOF2'
server {
    listen 80;
    server_name _;
    root /var/www/wieerwill.dev;
    index index.html;
    location / { try_files $uri $uri/ =404; }
}
EOF2

sudo ln -s /etc/nginx/sites-available/wieerwill.dev /etc/nginx/sites-enabled/
```

- direkt.
- nachvollziehbar.
- ein guter Anfang.

wenig Magie, wenig bewegliche Teile

- Ein Verzeichnis mit Dateien
- Ein Webserver, der genau diese Dateien ausliefert
- Keine App-Plattform, kein Orchestrator, kein Dashboard-Karaoke
- Gut zum Lernen, Debuggen und Verstehen des Stacks
- ABER: schnell Grenze erreicht, wo Heimhosting nervig wird

Stufe 2: mehrere Dienste? Dann wird Docker schnell vernünftig

```yaml
services:
  traefik:
    image: traefik:v3.4
    command:
      - --providers.docker=true
      - --entrypoints.web.address=:80
      - --entrypoints.websecure.address=:443
      - --certificatesresolvers.le.acme.email=admin@example.com
      - --certificatesresolvers.le.acme.storage=/letsencrypt/acme.json
      - --certificatesresolvers.le.acme.tlschallenge=true
    ports: ["80:80", "443:443"]
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - ./letsencrypt:/letsencrypt

  site:
    image: nginx:alpine
    volumes:
      - ./site:/usr/share/nginx/html:ro
```

Docker ist der pragmatische Sweet Spot

- Dienste, Netzwerke und Volumes werden beherrschbar.
- Compose ist hervorragend, solange du dein Setup versionierst.
- `.env`, Tags, Volumes und Migrationspfade werden schnell zur zweiten Konfigurationsebene.
- “Neu deployen” ist leicht.
- **Daten korrekt behalten** ist der eigentliche Job.
- **Aber:** Container lösen Paketkonflikte, nicht Verantwortung.

Stufe 3: auf NixOS wird das plötzlich langweilig

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

{
  services.caddy = {
    enable = true;
    virtualHosts."wieerwill.dev".extraConfig = ''
      root * /var/www/wieerwill.dev
      file_server
    '';
  };

  networking.firewall.allowedTCPPorts = [ 80 443 ];
}
```

P.S.: “langweilig” ist hier ein Kompliment

Der Easterhegg-Talk endete ungefähr hier

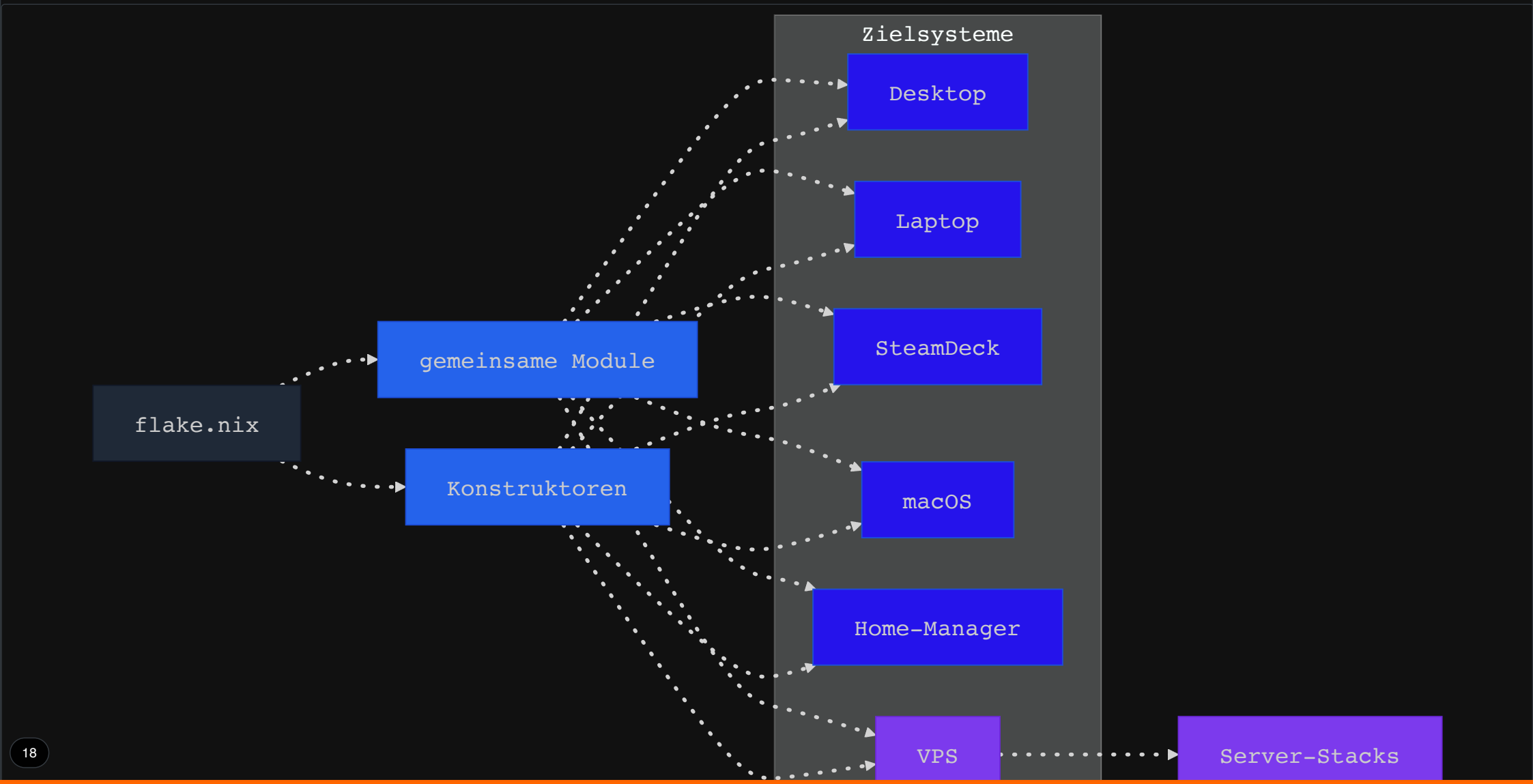
Die eigentliche Frage begann danach

"Ja schön... aber wie sieht dein System jetzt wirklich aus?"

Kapitel 02

Eine Flake für viele Rechner und viele Szenarien

Kernidee: nicht “ein Host = ein Repo”



Ziel: verständliche Zuständigkeitskarte

- Ein deklarativer Ort für Systemzustand.
- Klare Host-Grenzen.
- Wiederverwendbare Module statt Copy-Paste-Konfig.
- Gepinnte Inputs statt “gestern lief es noch”.
- Services als beschriebene Szenarien.
- **Aber:** Je mächtiger das Setup, desto wichtiger wird Struktur und Benennung.

In meiner Flake leben mehrere Zielsysteme nebeneinander

```
outputs = { self, nixpkgs, home-manager, nix-darwin, ... }@inputs: {
  nixosConfigurations = {
    t440p = mkNixosHost { ... };
    steamdeck = mkNixosHost { ... };
    vps02_04 = mkNixosHost { ... };
    vps04_08 = mkNixosHost { ... };
    vps08_32 = mkNixosHost { ... };
    ...
  };

  darwinConfigurations = {
    macbook_m3 = mkDarwinHost { ... };
  };

  homeConfigurations = {
    xaorus = mkHomeHost { ... };
  };
};
```

- Eine Codebasis.
- Mehrere Laufzeitszenarien.
- Explizite Outputs statt lose Ordnerfolklore.

Der Trick sind Konstruktoren

```
mkNixosHost = { system, modules, ... }:
  nixpkgs.lib.nixosSystem {
    inherit system;
    specialArgs = sharedSpecialArgs;
    modules = modules ++ [
      home-manager.nixosModules.home-manager
      sops-nix.nixosModules.sops
      { nixpkgs.overlays = [ overlay ]; }
    ];
  };

mkDarwinHost = { system, modules, ... }: ...;
mkHomeHost   = { system, modules, ... }: ...;
```

Gleiches Wiring, gleiche Semantik, weniger Streuung.

Shared args sind stille Infrastruktur

- Inputs und Policy werden zentral in den Modulgraph injiziert.
- Damit greifen Hosts auf dieselben Pins und dieselben Regeln zu.
- Lokale Pakete, Sonderquellen und Rollen müssen nicht global “magisch” importiert werden.
- Das macht Module lesbarer und testbarer.
- Und es verhindert, dass eine Host-Datei heimlich ihre eigene Welt baut.

Home Manager wird zum User-Kontrollplane

```
homeModules = {
  git.enable = true;
  terminal.enable = true;
  syncthing.enable = true;
  secrets.enable = true;
  firefox.enable = false;
};
```

- Hosts schalten Features an oder aus.
- Module definieren Verhalten.
- Host-Dateien beschreiben Szenarien statt Dotfiles bis zum Bitteren Ende.
- Das ist für Desktop, Laptop und Server-Userprofile extrem angenehm.

Gute Host-Dateien komponieren

- `hosts/<host>/configuration.nix` wählt Module.
- `modules/*.nix` enthalten wiederverwendbare Policy.
- `home/*.nix` enthalten userseitige Funktionsblöcke.
- `services/*.nix` kapseln VPS-Stacks.
- Wenn Host-Dateien anfangen, überall Logik zu duplizieren, ist das ein Geruch!

Kapitel 03

Wie daraus ein echter Server wird

Jetzt geht es um den VPS und die Service-Architektur.

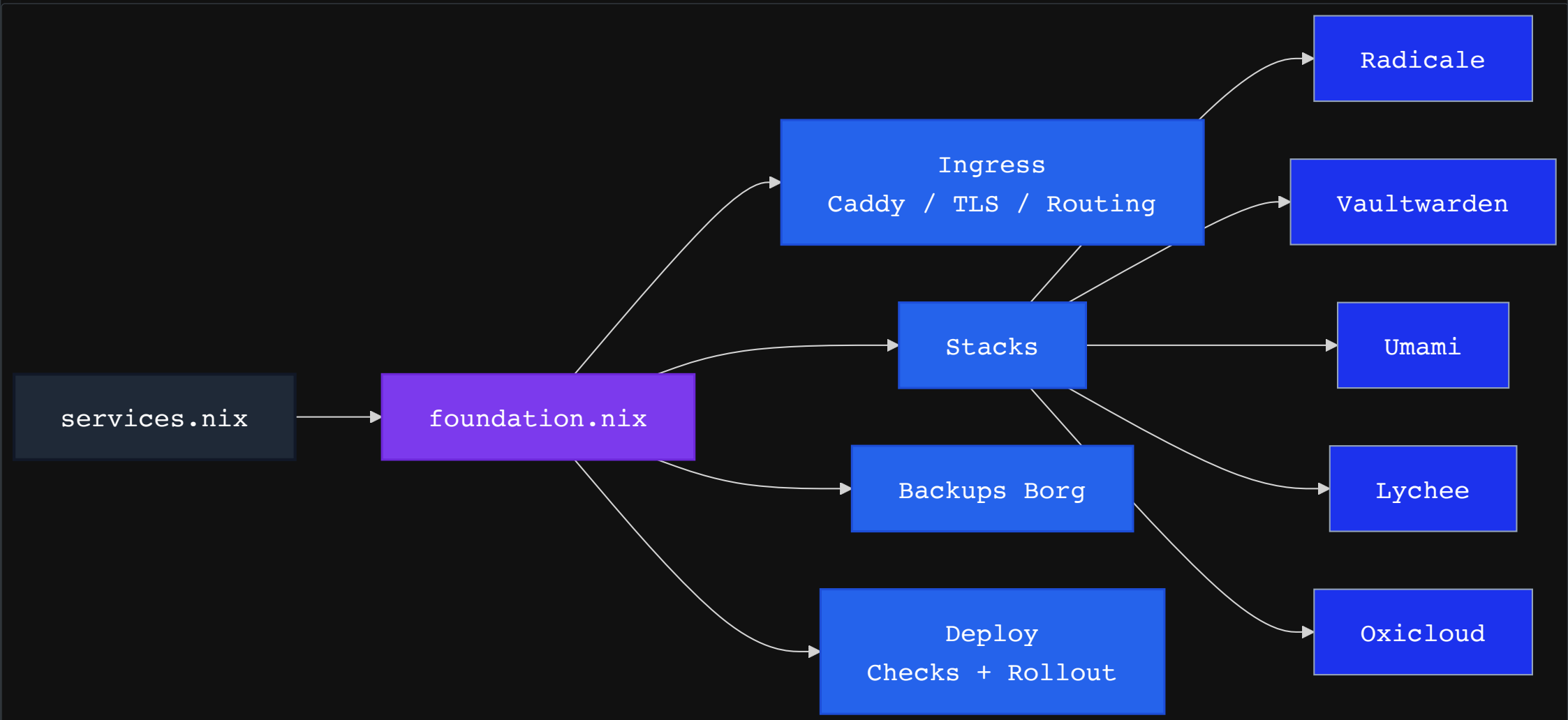
Mein VPS hat eine Control Plane und viele Stacks

```nix
# hosts/vps04_08/services.nix
{
  imports = [
    ../../services/vps/foundation.nix
    ../../services/vps/borgbackup.nix
    ../../services/caddy/default.nix
    ../../services/umami/default.nix
    ../../services/gitea/default.nix
    ../../services/radicale/default.nix
  ];

  vps.services.caddy.enable = true;
  vps.services.umami.enable = true;
  vps.services.gitea.enable = true;
  vps.services.radicale.enable = true;
}
```

- Eine Datei entscheidet, **was** laufen soll.
- Die Module definieren, **wie** es läuft.

Dahinter liegt eine gemeinsame Runtime-Basis



`foundation.nix` definiert den gemeinsamen Vertrag: rootless OCI, Datenpfade, Image-Locks, Bootstrap, Unit-Overrides, gemeinsame Defaults

Rootless OCI für Internetdienste

- Podman ist daemonless und weitgehend Docker-kompatibel.
- Viele Kommandos laufen als normaler User.
- Das passt gut zu service-spezifischen Laufzeit-Usern.
- Image-Pins und Datenverzeichnisse lassen sich sauber kontrollieren.
- **Aber:** Rootless ist kein Zauber. Volumes, Rechte und Networking bleiben offen.

Stack-Modul hat (bei mir) immer dieselbe Grundform

```nix
{ config, lib, pkgs, ... }:
let
  cfg = config.vps.services.umami;
in {
  options.vps.services.umami = {
    enable = lib.mkEnableOption "Umami analytics";
    host = lib.mkOption { type = lib.types.str; };
  };

  config = lib.mkIf cfg.enable {
    assertions = [
      { assertion = config.vps.services.caddy.enable;
        message = "Umami requires Caddy ingress."; }
    ];

    # users, dirs, secrets, containers, caddy, healthz ...
  };
}
```

- Optionen deklarieren.
- Mit `mkIf` aktivieren.
- Voraussetzungen hart prüfen.
- Danach Runtime-Artefakte erzeugen.

Caddy ist gemeinsame Kante nach außen

```
services.caddy.virtualHosts.${cfg.host}.extraConfig = ''
  encode zstd gzip
  reverse_proxy 127.0.0.1:${toString cfg.port}
'';

systemd.services.${cfg.systemdName}.serviceConfig = {
  Restart = "always";
};
```

- Eine Stelle für Domains, TLS und Routing.
- Eine Stelle für konsistente öffentliche Erreichbarkeit.
- Eine Stelle für `/healthz`, Logs und Debugging.
- Genau so wird aus vielen Diensten ein System.

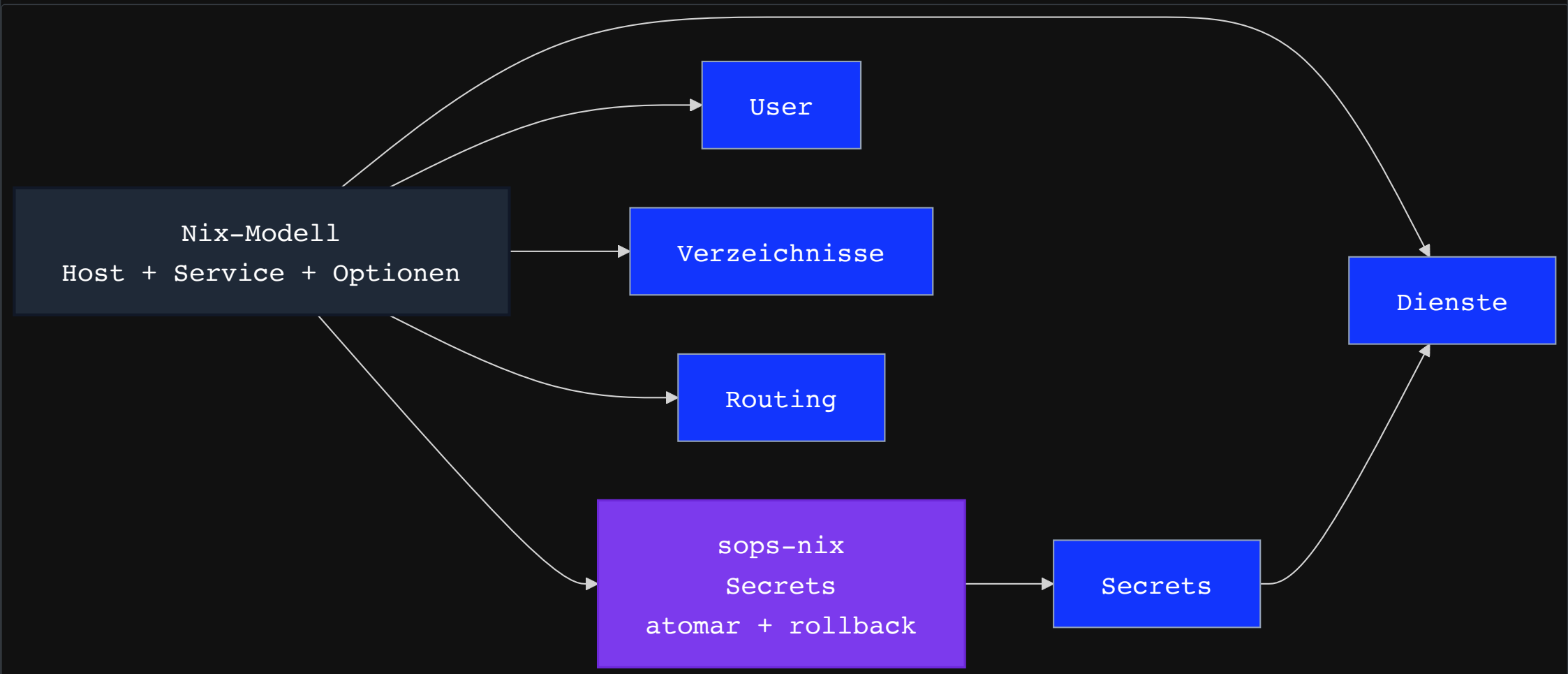
Geheimnisse nicht ins Repo (im Klartext)

```
sops.secrets."umami/app-secret" = {
  owner = "vps-umami";
};

sops.templates."umami.env".content = ''
  APP_SECRET=${config.sops.placeholder."umami/app-secret"}
  DATABASE_URL=${config.sops.placeholder."umami/database-url"}
'';
```

- SOPS verschlüsselt.
- `sops-nix` materialisiert zur Laufzeit.
- Templates verbinden Secrets mit Diensten.
- Secrets werden dadurch Teil des deklarativen Modells, ohne plaintext im Git.

der größte Unterschiede zu “nur Docker Compose”



Secrets, Dienste, User, Verzeichnisse und Routing kommen aus einem Modell.

Backups sind Teil der Architektur

```
services.borgbackup.jobs.vps-data = {
  paths = [ "/srv/data" "/var/lib/radicale" ];
  repo = "ssh://backup@backupbox/./vps04_08";
  encryption.mode = "repokey-blake2";
  compression = "zstd,6";
  prune.keep = {
    daily = 7;
    weekly = 4;
    monthly = 6;
  };
};
```

- Backups müssen mit dem Setup altern.
- Retention gehört beschrieben.
- Restore gehört geübt.
- Alles andere ist Optimismus.

Neue Maschinen komen per SSH ins System

```
nix run github:nix-community/nixos-anywhere -- \
  --generate-hardware-config nixos-generate-config ./hardware-configuration.nix \
  --flake .#vps04_08 \
  --target-host root@203.0.113.42
```

- Zielhost per SSH erreichen.
- Disk layout und Systemdefinition aus der Flake anwenden.
- NixOS installieren.
- Danach ist der Host Teil derselben Architektur wie alle anderen.

Disko macht Festplatten deklarativ

```
disko.devices.disk.main = {
  device = "/dev/nvme0n1";
  type = "disk";
  content = {
    type = "gpt";
    partitions = {
      ESP = {
        size = "500M";
        type = "EF00";
        content = {
          type = "filesystem";
          format = "vfat";
          mountpoint = "/boot";
        };
      };
    };
  };
};
```

Wenn ich einen Server neu aufsetze, möchte ich keine Partitionsfolklore erinnern müssen.

Luxus ist die Prüfkette davor

```
bash ./scripts/nix.sh check
bash ./scripts/nix.sh hosts
bash ./scripts/check-nixos-hosts.sh
```

- Erst Evaluation.
- Dann Host-Invarianten.
- Dann Service-Konsistenz.
- Dann Deployment.

“Reproduzierbar” wird im Alltag erst mit Checks wirklich belastbar.

NixOS ist nicht die Magie

Die Magie ist, dass ich Dinge nicht mehr erraten muss

Und das ist im Betrieb erstaunlich viel wert

Kapitel 04

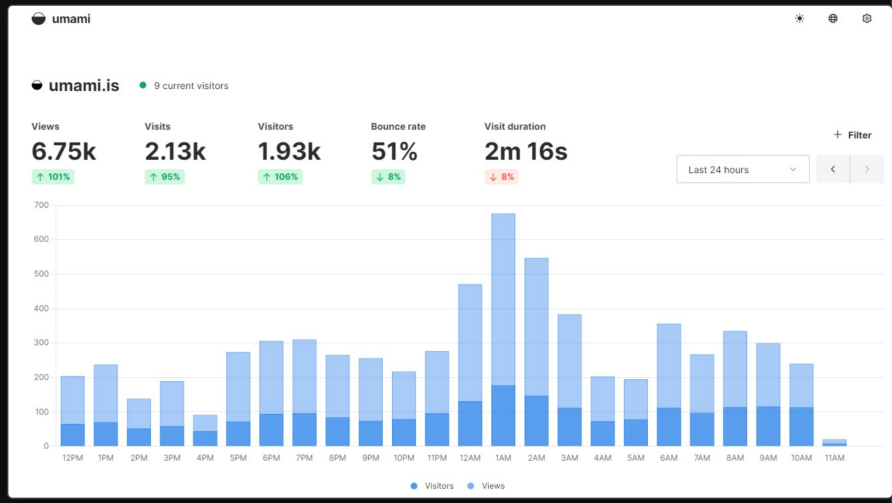
Was darauf dann konkret läuft

Nicht alles ist gleich kritisch.

Nicht alles muss selbst gehostet werden.

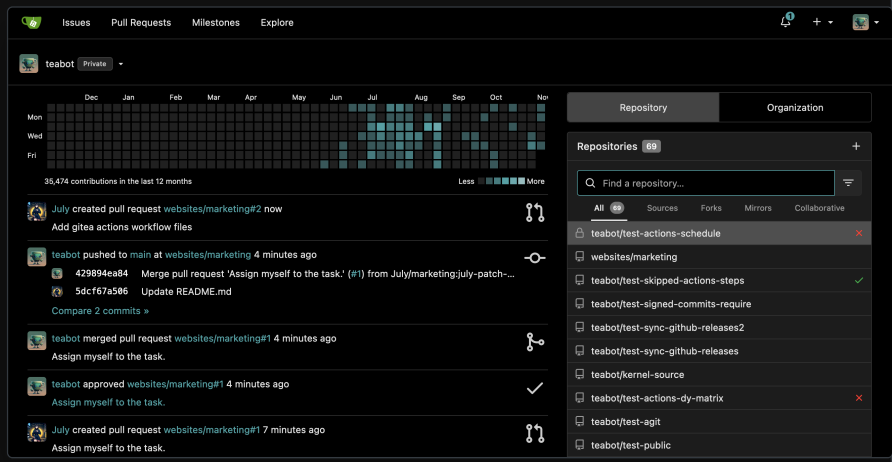
Website bleibt Anker – Umami hält Metriken

- Die eigene Website ist die stabilste Heimat.
- Analytics muss dafür nicht automatisch bei Google wohnen.
- Umami ist klein, flott und datensparsamer gedacht.
- Für mehrere Seiten reicht oft eine Instanz.
- Das ist Self-Hosting, das im Alltag sofort Nutzen bringt.



Gitea ist für private und kleine Team-Setups richtig gut

- Eigene Repositories ohne Plattformlogik von außen.
- Issues, Reviews, Packages und CI/CD direkt daneben.
- Ressourcenbedarf und Admin-Aufwand bleiben vernünftig.
- Gerade für interne Projekte oder sensible Repos angenehm.
- **Aber:** Auch hier bist du für Updates, Backups und Rechte zuständig.



Im Alltag gewinnen oft langweilige Dienste

Kalender & Kontakte

Radicale für CalDAV/CardDAV ohne Monster-Stack.

Dateisync

Syncthing für Geräte – nicht als Ausrede gegen echte Backups.

Bibliothek

Kavita für EPUB, PDF, Comics und Manga.

Fotos

Immich/Lychee, wenn die eigene Galerie nicht Trainingsdaten werden soll.

Manche Dienste sind kein Hobby mehr

Vorsicht bei

- Vaultwarden
- Mail
- Mastodon-Instanzen mit echter Öffentlichkeit

Warum

- hoher Schadensradius
- Missbrauch und Angriffsfläche
- Moderation, Zustellbarkeit, Recovery
- soziale und rechtliche Verantwortung

> Der technische Deploy ist oft der einfache Teil. Der Betrieb ist der eigentliche Vertrag.

offene Protokolle wichtiger als eine App

- Website und Feed auf eigener Domain
- Kalender über CalDAV
- Kontakte über CardDAV
- soziale Vernetzung über ActivityPub
- Git als offenes Datenmodell statt Plattformgefängnis
- Protokolle überleben Produkte eher als umgekehrt

Kapitel 05

Wie man anfängt, ohne sich selbst zu hassen

Mein realistischer Einstiegspfad

1. **Eigene Website oder einen einzelnen Dienst hosten.**
2. **Dann TLS und Reverse Proxy zentralisieren.**
3. **Dann erst mehrere Dienste bündeln.**
4. **Backups bauen, bevor sie nötig sind.**
5. **Secrets und Konfiguration versionieren.**
6. **Gefährliche Dienste bewusst spät angehen.**

Verständnis, Wiederholbarkeit und Recovery vor Funktionsvielfalt.

Nicht jeder Dienst braucht denselben Betriebsmodus

Nativ in NixOS

- kleine, stabile Systemdienste
- eng mit Systemintegration
- wenig Overhead

OCI / Podman

- Third-Party-Webapps
- klar gekapselte Laufzeit
- guter Pragmatismus

Extern gehostet

- wenn Verfügbarkeit wichtiger ist als Souveränität
- wenn Haftung nervt
- wenn Betrieb dich vom eigentlichen Ziel abhält

Regel

- nicht dogmatisch entscheiden
- nach Risiko, Daten, Aufwand und Recovery wählen
- Architektur einhalten

Wann Self-Hosting sich lohnt

- bei klaren Datenschutz- oder Exportanforderungen
- bei offenen Protokollen und langen Datenlebenszyklen
- wenn du dieselben Workflows oft wiederholst
- wenn du Admin-Arbeit tatsächlich tragen willst
- wenn du Infrastruktur lieber beschreibst als erinnerst

Die eigentliche These ist ziemlich unspektakulär

- Nicht alles selbst hosten.
- Aber die wichtigen Dinge **verstehbar, exportierbar und ersetzbar** machen.
- Eine eigene Website ist dafür ein sehr guter Anfang.
- Docker oder Podman sind oft der pragmatische Mittelweg.
- NixOS wird stark, sobald du mehr als eine Maschine oder mehr als ein Gedächtnisproblem hast.
- Dezentralisierung beginnt nicht im Rack, sondern bei deinen Abhängigkeiten.

Mehr?

- Nix & NixOS
- Home Manager
- sops-nix
- nixos-anywhere + disko
- Borg
- Caddy
- und dann: den nächsten Dienst

END OF BRIEFING

MAIL:
robert.jeutter@wieerwill.dev

MASTODON:
https://chaos.social/@wieerwill

LINKEDIN:
https://www.linkedin.com/in/wieerwill

WEB:
https://wieerwill.dev