Secure Infrastructure Lab

Linux, Container & sichere Server-Setups für KMUs

📍 Lakeside Science & Technology Park
📅 22.–23.01.2026

Eine Kooperation von
Coding School Wörthersee (CSAW) & **HoliSec**

Herzlich Willkommen 👋

Worum geht es hier?

- Praxis statt Theorie
- Reale Server, reale Fehler, reale Lösungen
- Fokus auf **verstehen**, nicht blind kopieren

Für wen ist dieser Workshop?

- IT-affine Entscheider:innen
- Admins & Tech-Verantwortliche in KMUs
- Entwickler:innen mit Infrastruktur-Verantwortung

👉 **Kein Vorwissen notwendig – Neugier reicht wir starten bei 0**

Wie wir arbeiten

- Kurze Inputs
- Viel Hands-on
- Fehler sind **Teil des Lernprozesses**
- Fragen jederzeit erlaubt 👍

💡 **Slides führen – erklärt wird im Terminal**

Was ihr am Ende könnt

Nach diesen zwei Tagen könnt ihr:

- Linux-Server sicher betreiben
- Container sinnvoll einsetzen
- Applikationen sauber hosten
- Risiken realistisch einschätzen
- bessere Infrastruktur-Entscheidungen treffen

Struktur des Workshops

Tag 1 – Fundament

- Linux verstehen & absichern
- SSH, User, Updates
- Docker & Docker Compose
- Container & Netzwerke verstehen

Tag 2 – Anwendungen & Betrieb

- Applikationen inbetriebnehmen
- Reverse Proxy & HTTPS
- Monitoring & Backups
- Best Practices für KMUs
- Ausblick: Advanced Security

Ablauf & Organisation

- Hands-on auf eigenen VMs
- Arbeiten im eigenen Tempo
- Trainer unterstützen aktiv
- Pausen nach Bedarf

☕ Bitte meldet euch, wenn ihr hängt – **nicht warten!**

Tag 1 – Linux & Container-Fundament

Ziel von heute:

- Ein sicherer Linux-Server
- Saubere Zugänge
- Docker läuft
- Verständnis statt copy & paste

Tag 1 – Linux & Container-Fundament

Secure Infrastructure Lab · Lakeside Park · CSAW × HoliSec

Heute bauen wir das Fundament:

- Linux sicher verstehen
- Zugänge sauber aufsetzen
- Docker ohne Magie begreifen

Zielbild: Was soll heute Abend stehen?

Am Ende von **Tag 1** habt ihr:

- ✅ einen **harten, sauberen Linux-Host**
- ✅ **User- und Rechte-Konzept**, kein root-login
- ✅ **SSH-Keys**, SSH Hardening
- ✅ **Updates & Wartung** aufgesetzt
- ✅ Docker installiert + Basics verstanden
- ✅ Docker Networking + Compose als Grundlage

> Tag 2 baut darauf auf: Reverse Proxy, TLS, Cloudflare, Apps, Monitoring, Backup

Arbeitsweise heute (wichtig)

- **Hands-on > Theorie** (ca. 70/30)
- Ich erkläre kurz — dann macht ihr’s selbst
- Fehler sind normal: wir debuggen live
- Wenn du hängen bleibst: **Hand heben** (nicht warten)

Regel: Slides führen — gearbeitet wird im Terminal

Setup: Wie arbeiten wir?

Ihr habt eine Netcup VM mit Public IP (ggf. 2er-Teams).

Tools:

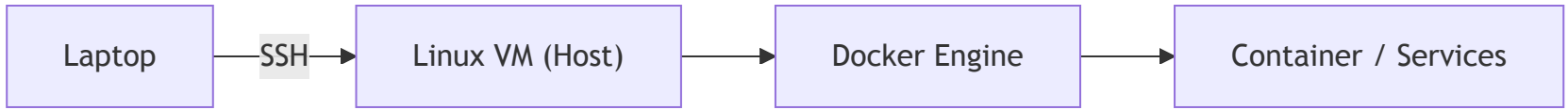
- Terminal (Pflicht)
- VS Code Remote SSH (optional, aber sehr praktisch)

Ziel: Ihr könnt alles auch ohne VS Code im Terminal.

Unsere VMs bei Netcup

ID	IP-Adresse	Login-User	Teilnehmer	Passwort
1	xxx.xxx.xxx.101	root	Teilnehmer 1	********
2	xxx.xxx.xxx.102	root	Teilnehmer 2	********
3	xxx.xxx.xxx.103	root	Teilnehmer 3	********
4	xxx.xxx.xxx.104	root	Teilnehmer 4	********
5	xxx.xxx.xxx.105	root	Teilnehmer 5	********
6	xxx.xxx.xxx.106	root	Teilnehmer 6	********
7	xxx.xxx.xxx.107	root	Teilnehmer 7	********
8	xxx.xxx.xxx.108	root	Teilnehmer 8	********

Mental Model: Host, Netzwerk, Dienste



Heute & morgen

Heute:

- Linux VM + Docker Engine verstehen

Morgen:

- Container sicher exponieren

Agenda Tag 1

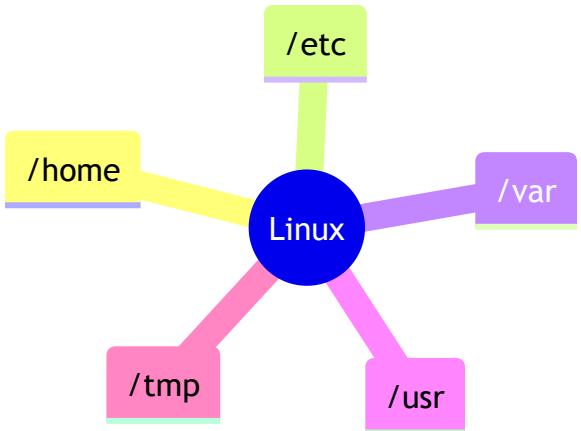
- Orientierung & Linux Basics (Navigation, Files, Editor)
- Users, Gruppen, Rechte (Permissions!)
- SSH Keys & Hardening
- Updates, UFW, Basis-Hardening
- Docker ohne Magie (docker run)
- Docker Networking & Docker Compose

Abschnitt 1

Linux Orientierung: „Wo bin ich hier?“

Ziel: Du kannst dich im System bewegen, Dateien lesen & anlegen
Ohne das wird jeder Hardening-Schritt später frustig.

Das Linux-Filesystem (nur das Nötigste)



Hands-on: Orientierung im Terminal

```
pwd
whoami
hostname
uname -a
ls -lah
```

Mini-Ziel:

- Ich weiß, wer ich bin
- Ich weiß, wo ich bin
- Ich sehe, was hier liegt

Navigation: cd, ls, less (Profi-Minimum)

```
cd /
ls -lah
cd /etc
ls -lah | head
```

Dateien ansehen ohne zu zerstören:

```
less /etc/ssh/sshd_config
```

Shortcut:

- q = quit
- /wort = suchen

Hands-on: Workspace anlegen (Ordnerstruktur)

Wir arbeiten geordnet, damit man’s später wiederfindet:

```
mkdir -p ~/workshop/{labs,notes,compose,scripts}
tree -a ~/workshop || true
```

Falls tree fehlt:

```
sudo apt update && sudo apt install -y tree
```

Dateien erstellen: touch, cat, heredoc

```
cd ~/workshop/notes
touch commands.txt
echo "Tag 1 - Command Log" > commands.txt
```

Mehrzeilig (super für Configs):

```
cat <<'EOF' >> commands.txt
pwd
whoami
ls -lah
EOF
```

Editor: vim oder nano (du entscheidest)

Vim Quickstart (Minimum):

- `i \→ insert`
- `Esc \→ normal mode`
- `:wq \→ speichern & beenden`

```
vim ~/workshop/notes/commands.txt
```

Oder nano:

```
nano ~/workshop/notes/commands.txt
```

VS Code Remote SSH (optional, aber nice)

Wenn du willst: Remote SSH nutzen, um schneller Files zu bearbeiten.

Wichtig:
Auch mit VS Code musst du die Commands verstehen.

Wenn VS Code nicht geht: kein Stress - Terminal reicht.

Checkpoint ✅ Abschnitt 1

Wenn das passt, bist du richtig:

- ✓ du kannst navigieren (cd, ls, less)
- ✓ du hast ~/workshop/ Struktur
- ✓ du kannst Files erstellen und editieren

Abschnitt 2

Users, Gruppen & Berechtigungen (EXTREM wichtig)

Ziel: Du verstehst warum „Permission denied“ passiert
und wie du’s sauber löst (ohne alles 777 zu machen).

Mini-Theorie: Ownership & Permissions

Jede Datei hat:

- Owner (User)
- Group
- Mode Bits: rwx für user/group/others

Beispiel:

```
-rw-r----- 1 dihuser admin  531 Jan  8 12:01 secrets.txt
```

Merksatz:
Owner/Group bestimmen wer, rwx bestimmen was.

Hands-on: Permissions lesen lernen

```
cd ~/workshop
touch fileA
ls -l fileA
stat fileA
```

Zeig mir:

- Owner?
- Group?
- Rechte?

Cooles Bash-Snippet: Rechte hübsch anzeigen

```
perm() { stat -c "%A  %U:%G  %n" "$@"; }
perm ~/workshop/fileA
```

Optional gleich in ~/.bashrc speichern:

```
echo 'perm(){ stat -c "%A  %U:%G  %n" "$@"; }' >> ~/.bashrc
source ~/.bashrc
```

Hands-on: chmod / chown / chgrp (Basics)

```
mkdir -p ~/workshop/labs/perm-demo
cd ~/workshop/labs/perm-demo
touch secret.txt
perm secret.txt
```

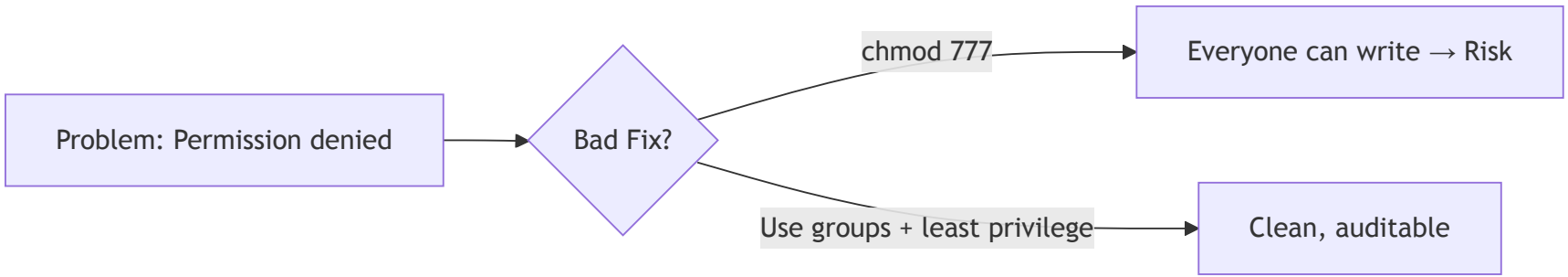
Rechte ändern:

```
chmod 600 secret.txt
perm secret.txt
```

Owner ändern (nur mit sudo):

```
sudo chown dihuser:dihuser secret.txt
perm secret.txt
```

Warum "chmod 777" verboten ist



Wir arbeiten mit:

- Gruppen
- gezielten Rechten
- nachvollziehbaren Changes

Checkpoint ✅ Abschnitt 2 (bis hier)

- ✓ du kannst ls -l interpretieren
- ✓ du kennst chmod/chown/chgrp
- ✓ du weißt, warum 777 Mist ist

Nächster Schritt

Jetzt bauen wir das sauber in ein Server-Setup ein:

- User dihuser + sudo
- SSH Keys
- SSH Hardening
- Updates & Firewall

➡️ weiter mit Abschnitt 3

Abschnitt 3
User, sudo & SSH Hardening

> Ziel:
> Kein Root-Login mehr, saubere User, **nur SSH-Keys**, nachvollziehbare Zugänge.

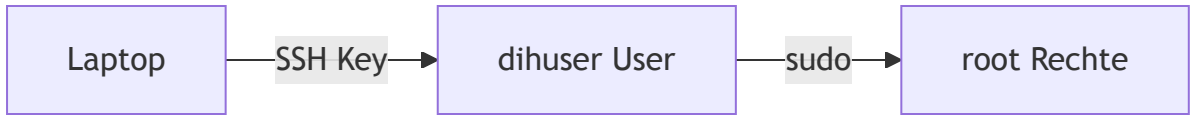
Warum wir Root loswerden müssen

- Root hat **keine Sicherheitsleine**
- Jeder Fehler = Full Compromise
- Logs sind schlechter nachvollziehbar

Best Practice:

> **Root nur für Notfälle – nie für Daily Work**

Zielbild: Zugriffskonzept



- SSH Login nur als normaler User
- Root-Zugriff nur via sudo
- Alles ist logbar

Hands-on: Root Passwort setzen (falls nicht erfolgt)

```
passwd
```

💡 Auch wenn Root später nicht loginfähig ist:

- Starkes Passwort
- Sicher im Passwortmanager

Benutzer anlegen: dihuser

```
adduser dihuser
```

Fragen:

- Passwort setzen (temporär ok)
- User-Infos optional

dihuser User zu sudo hinzufügen

```
usermod -aG sudo dihuser
```

Test (noch als root):

```
groups dihuser
```

Wechsel zu dihuser

```
su - dihuser
whoami
```

💡 Ab jetzt: wir arbeiten nicht mehr als root

sudo verstehen (wichtig!)

```
sudo ls /root
sudo whoami
```

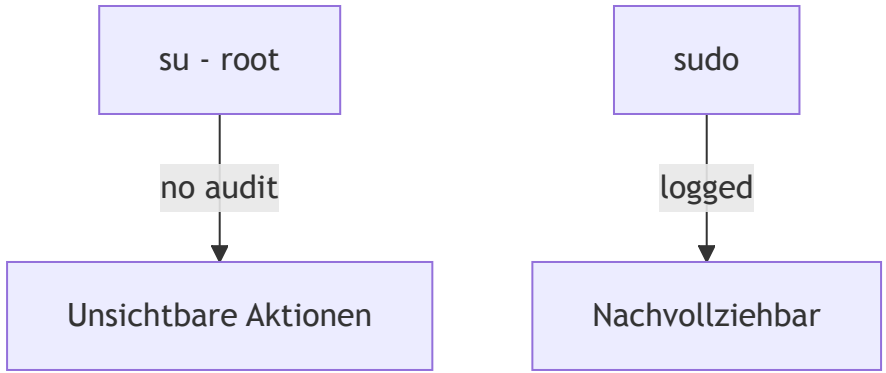
Was passiert hier?

- sudo fragt dein Passwort
- Aktion wird geloggt
- Zeitlich begrenzter Cache

Logs:

```
sudo journalctl -u sudo
```

Warum sudo besser ist als su



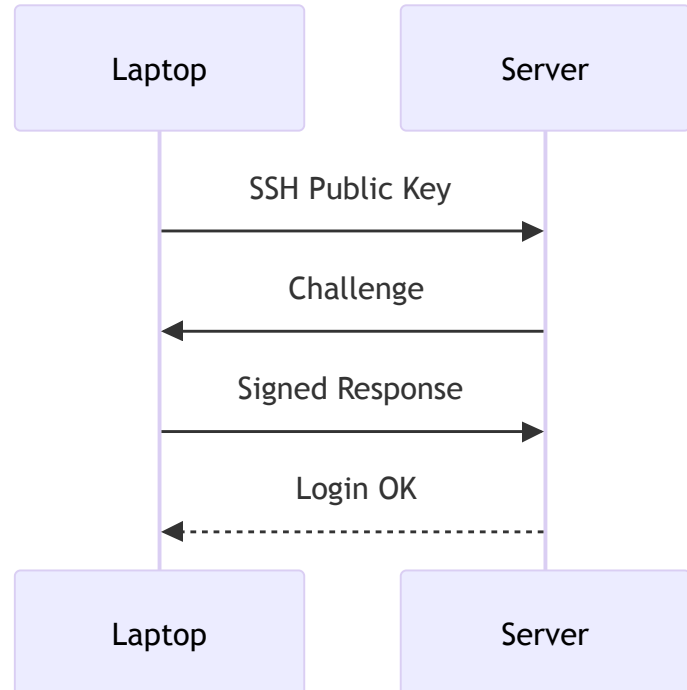
In Firmenumgebungen:

- sudo = Pflicht
- su = verboten

SSH: Authentifizierung verstehen

Zwei Wege:

- ❌ Passwort
- ✅ SSH Key (asymmetrisch)



Hands-on: SSH Key lokal erzeugen

👉 Auf deinem Laptop, nicht auf dem Server!

```
ssh-keygen -t ed25519 -C "secure-infra-lab"
```

Empfehlungen:

- ed25519
- Key-Passphrase setzen

Public Key auf Server kopieren

```
ssh-copy-id dihuser@<SERVER-IP>
```

Test:

```
ssh dihuser@<SERVER-IP>
```

➡️ Login ohne Passwort

SSH Konfiguration verstehen

Config-Datei:

```
sudo vim /etc/ssh/sshd_config
```

Wir ändern gezielt, nicht blind.

SSH Hardening – Pflicht-Settings

```
PermitRootLogin no
PasswordAuthentication no
PubkeyAuthentication yes
ChallengeResponseAuthentication no
UsePAM yes
```

💡 UsePAM bleibt an → Login Accounting

Sichere Änderung mit Backup & Test

```
sudo cp /etc/ssh/sshd_config /etc/ssh/sshd_config.bak
sudo vim /etc/ssh/sshd_config
```

Config testen:

```
sudo sshd -t
```

⚠ Nur wenn kein Output kommt, ist alles OK

SSH neu starten (vorsichtig!)

```
sudo systemctl restart ssh
```

WICHTIG:

- Aktive SSH-Session offen lassen
- Zweite Session zum Testen öffnen

Typische Fehler & Debugging

❌ Login geht nicht mehr?

```
sudo journalctl -u ssh -n 50
```

Checkliste:

- Tippfehler?
- sshd -t vorher ok?
- Richtiger User?
- Public Key richtig?

authorized_keys richtig prüfen

```
ls -lah ~/.ssh
ls -lah ~/.ssh/authorized_keys
```

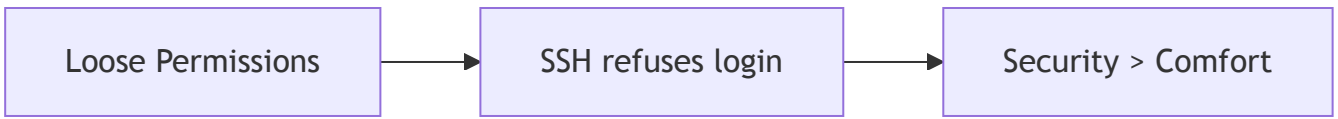
Rechte müssen sein:

```
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
```

Mini-Exkurs: Warum Permissions hier kritisch sind

SSH ist streng:

- falsche Rechte → Login verweigert
- Security > Convenience



Checkpoint ✅ Abschnitt 3

- ✓ Login nur als dihuser
- ✓ SSH Key funktioniert
- ✓ Root Login deaktiviert
- ✓ Passwort-Login aus
- ✓ sudo verstanden & nutzbar

Ausblick nächster Abschnitt

Jetzt, wo der Zugang sicher ist:

➡️ Systempflege & Schutz:

- Updates
- unattended-upgrades
- UFW Firewall
- Minimal Exposure

👉 weiter mit Abschnitt 4

Abschnitt 4
Systempflege, Updates & Firewall

> Ziel:
> Ein Server, der **aktuell**, **wartbar** und **minimal exponiert** ist.

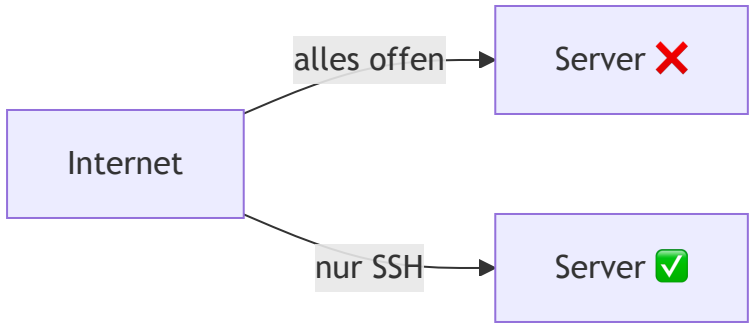
Warum Updates Security sind

- 80 % der erfolgreichen Angriffe nutzen:
  - bekannte Schwachstellen
  - ungepatchte Systeme
- Zero-Days sind selten
- **Updates schlagen fancy Security-Tools**

Merksatz:

> **Patchen ist die billigste Security-Maßnahme.**

Mental Model: Angriffsfläche reduzieren



Je weniger erreichbar:

- desto weniger Risiko
- desto weniger Stress

Hands-on: Paketverwaltung verstehen

```
sudo apt update
sudo apt list --upgradable
```

Upgrade durchführen:

```
sudo apt upgrade
```

Optional (Kernel / libc):

```
sudo apt full-upgrade
```

Autoremove & Cleanup (Hygiene)

```
sudo apt autoremove
sudo apt autoclean
```

💡 Hält das System schlank & übersichtlich

unattended-upgrades: Automatische Security-Patches

Installation:

```
sudo apt install -y unattended-upgrades
```

Interaktive Basiskonfiguration:

```
sudo dpkg-reconfigure unattended-upgrades
```

➡️ „Yes“ für automatische Updates

unattended-upgrades – was passiert wirklich?

Konfig-Datei:

```
/etc/apt/apt.conf.d/50unattended-upgrades
```

Wichtig:

- Security Updates automatisch
- Reboots nicht blind tagsüber

Automatische Reboots kontrollieren

```
sudo vim /etc/apt/apt.conf.d/50unattended-upgrades
```

Empfohlene Settings:

```
Unattended-Upgrade::Automatic-Reboot "false";
```

💡 Reboots später bewusst durchführen

Status & Logs prüfen

```
systemctl status unattended-upgrades
less /var/log/unattended-upgrades/unattended-upgrades.log
```

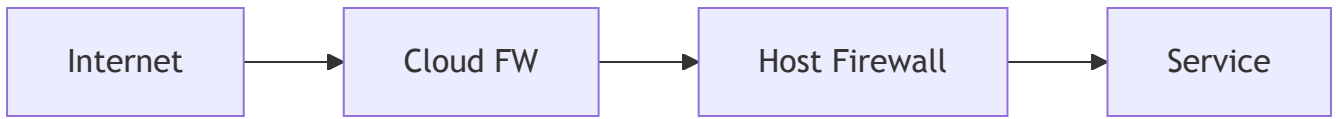
Checkpoint ✅ Updates

- ✓ Updates durchgeführt
- ✓ unattended-upgrades aktiv
- ✓ System ist patchfähig

Firewall am Linux Host

Warum Firewall trotz Cloud wichtig ist

- Cloud ≠ Firewall
- Security Groups ≠ Host Firewall
- Defense in Depth



UFW – unkompliziert & effektiv

Status prüfen:

```
sudo ufw status verbose
```

Default-Policy setzen:

```
sudo ufw default deny incoming
sudo ufw default allow outgoing
```

SSH absichern (Pflichtregel)

```
sudo ufw allow OpenSSH
```

Oder explizit:

```
sudo ufw allow 22/tcp
```

Firewall aktivieren (bewusst!)

```
sudo ufw enable
```

Warnung lesen → bestätigen

UFW Regeln anzeigen

```
sudo ufw status numbered
```

Regel löschen:

```
sudo ufw delete <NUMMER>
```

Typische Firewall-Fallen

❌ SSH vergessen erlaubt
❌ „allow all“ aus Bequemlichkeit
❌ Ports offen „für später“

Merksatz:

> **Was nicht gebraucht wird, bleibt zu.**

Mini-Demo: Blockierte Verbindung

Test (von extern):

- HTTP-Port nicht erreichbar
- SSH erreichbar

💡 So soll es sein.

- kann getestet werden mittels netcat oder telnet
- alternativ auch über https://shodan.io

UFW + Docker (wichtig!)

Problem:

- Docker umgeht UFW teilweise

Lösung (nur erwähnen):

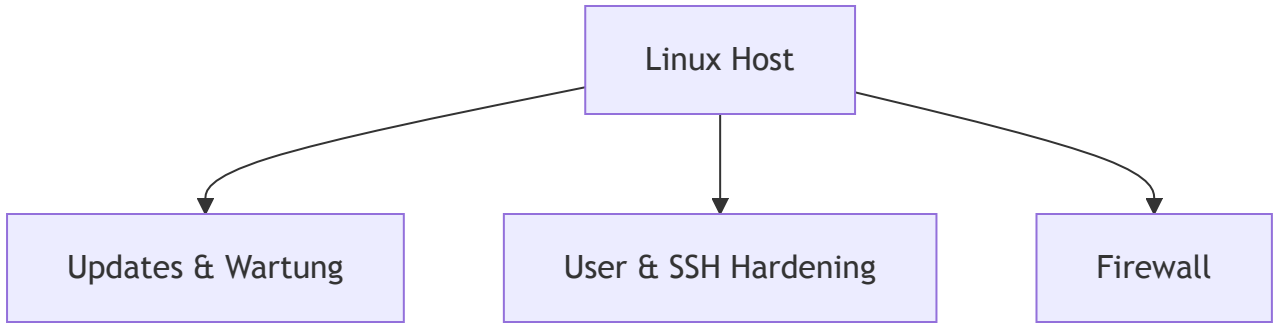
- Docker-Userland-Proxy
- später: Reverse Proxy + nur 80/443

👉 Wir fixen das konzeptionell morgen

Checkpoint ✅ Firewall

- ✓ Default deny incoming
- ✓ SSH erlaubt
- ✓ keine unnötigen Ports offen
- ✓ Verständnis für Defense in Depth

Tagesstand jetzt



Fundament steht.
Jetzt dürfen wir Container darauf loslassen.

Ausblick nächster Abschnitt

➡️ Docker verstehen, ohne Magie

- Images vs Container
- docker run
- Ports bewusst öffnen (noch!)
- Container sehen & stoppen

👉 weiter mit Abschnitt 5: Docker Basics

Abschnitt 5

Docker Basics – Container ohne Magie

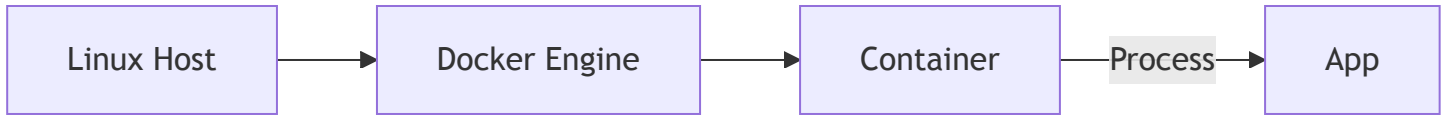
> Ziel:
> Du verstehst **was Docker wirklich tut**,
> kannst Container starten, stoppen, beobachten
> und erkennst **warum Port-Exposure gefährlich ist**.

Warum wir Docker einsetzen

- reproduzierbare Umgebungen
- gleiche Software überall
- schneller Start / schneller Stop
- saubere Trennung von Diensten

Docker ist **kein Ersatz für Security**,
sondern ein **Werkzeug zur Ordnung**.

Mental Model: Was ist ein Container?



Container = Prozess
läuft auf dem Host-Kernel
kein eigenes Betriebssystem

Das nennt man Kernal-Namespaces!

Image vs Container

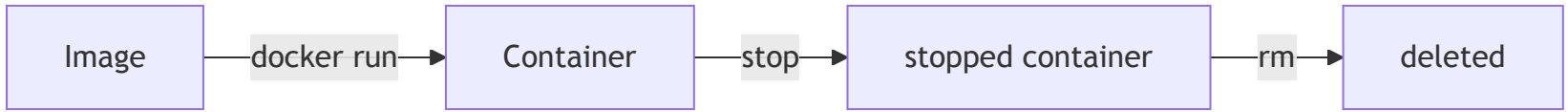


Image = Bauplan
Container = laufende Instanz
Image bleibt unverändert

Docker installieren (falls noch nicht passiert)

```
curl -fsSL https://get.docker.com | sh
sudo usermod -aG docker dihuser
```

👉 Danach neu einloggen

Test:

```
docker version
docker info
```

Dein erster Container (sichtbar & bewusst)

```
docker run -d -p 8080:80 nginx
```

Was passiert hier?

- `-d` → detached
- `-p 8080:80` → Port Mapping
- `nginx` → Image

Container live beobachten

```
docker ps
```

Browser:

```
http://SERVER-IP:8080
```

Funktioniert das bei euch?

Wie schaut es bei unsere ufw firewall aus? 🔐

💡 mit allow 8080 funktioniert es jetzt.

Was jedoch noch fehlt ist verschlüsselte Kommunikation zum Webserver.

Container Logs & Prozesse

```
docker logs <container-id>
docker top <container-id>
```

Vergleich:

```
ps aux | grep nginx
```

Container = Prozess mit Grenzen.

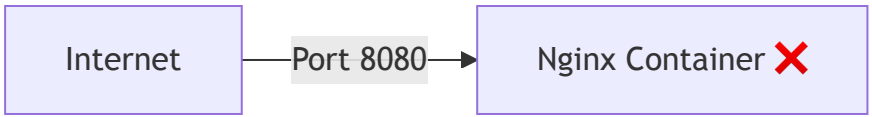
Container stoppen & löschen

```
docker stop <container-id>
docker rm <container-id>
```

Image bleibt:

```
docker images
```

Warum offene Ports ein Problem sind



Probleme:

- kein TLS
- kein Auth
- kein Rate Limit
- kein Überblick

Merksatz:

> **Jeder offene Port ist ein Versprechen an Angreifer.**

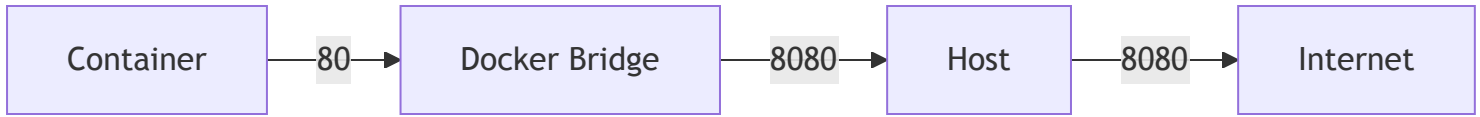
Port Mapping bewusst variieren

```
docker run -d -p 127.0.0.1:8081:80 nginx
```

Test:

- von Server: funktioniert
- von außen: ❌ blockiert

Wichtiger Aha-Moment



Port Exposure = Host-Entscheidung,
nicht Container-Entscheidung.

Container ohne Ports starten

```
docker run -d --name internal-nginx nginx
```

```
docker ps
```

➡️ Läuft, aber nicht erreichbar

Container inspizieren (Gold!)

```
docker inspect internal-nginx | less
```

Achte auf:

- NetworkSettings
- Mounts
- State

Typische Anfängerfehler

❌ alles mit Ports starten
❌ Container als VM behandeln
❌ Configs im Container ändern ❌ Logs ignorieren

👉 Alles fixen wir strukturiert.

Checkpoint ✅ Abschnitt 5

- ✓ Image vs Container verstanden
- ✓ Container starten & stoppen
- ✓ Logs lesen
- ✓ Port-Exposure bewusst eingesetzt
- ✓ Container ≠ VM

Vorbereitung auf den nächsten Schritt

Was jetzt noch weh tut:

- viele docker run Befehle
- nichts versioniert
- kein Überblick

➡️ Lösung:
Docker Compose

👉 weiter mit Abschnitt 6: Docker Networking & Compose

Abschnitt 6
Docker Networking & Docker Compose

> Ziel:
> Du verstehst **wie Container miteinander sprechen**,
> warum **Namen wichtiger sind als IPs**,
> und warum **Docker Compose Pflicht** ist.

Warum Networking jetzt kommt

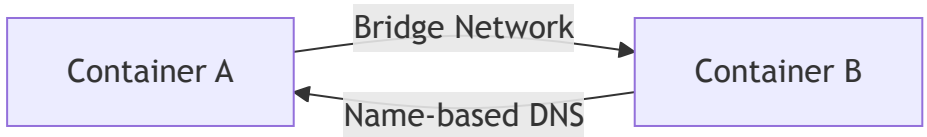
Bis jetzt:

- Container einzeln
- Ports manuell
- Chaos wächst

Ab jetzt:

- strukturierte Netze
- Services sprechen intern
- Host bleibt sauber

Docker Networking – das Grundprinzip



Wichtig:

- Docker bringt internes DNS
- Container finden sich über Namen
- IPs sind egal

Docker Default Bridge (kurz)

```
docker network ls
docker network inspect bridge
```

Probleme:

- alles in einem Netz
- keine Trennung
- schlecht für Übersicht

➡️ Eigene Netze sind besser.

Eigenes Netzwerk erstellen

```
docker network create lab-net
```

Check:

```
docker network ls
docker network inspect lab-net
```

Container im selben Netzwerk starten

```
docker run -d --name web --network lab-net nginx
docker run --rm --network lab-net alpine ping web
```

Aha-Moment:

- web ist DNS-Name
- keine IP nötig
- kein Port-Mapping

Warum das wichtig ist (ohne Grafik)

Was hier gerade passiert ist:

- Docker stellt **internes DNS** bereit
- Container finden sich über **Namen**
- Kommunikation bleibt **innerhalb des Netzwerks**
- Nichts davon ist von außen erreichbar

Merksatz:

> **Interne Kommunikation ≠ Exponierung**

Typischer Anfängerfehler (bitte merken)

❌ IP-Adressen hardcoden
❌ localhost im Container nutzen
❌ Ports für interne Kommunikation öffnen

Merksatz:

> **Container reden über Namen, nicht über Ports.**

Warum docker run jetzt an seine Grenzen kommt

Beispiel:

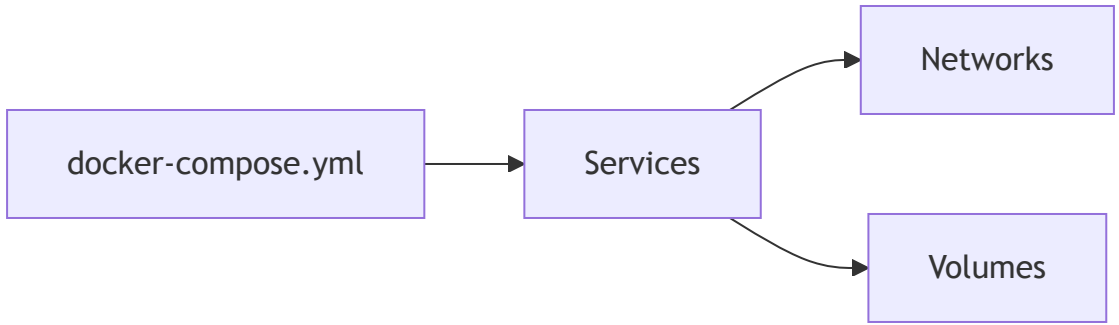
```
docker run ...
docker run ...
docker run ...
```

Probleme:

- nicht reproduzierbar
- nicht versionierbar
- nicht teamfähig

➡️ Compose löst das.

Docker Compose = Infrastruktur-Code



Ein File beschreibt:

- was läuft
- wie es vernetzt ist
- wie es gestartet wird

Erste docker-compose.yml (bewusst simpel)

```yaml
services:
  web:
    image: nginx
    container_name: web
    networks:
      - lab-net

networks:
  lab-net:
    external: true
```

Starten:

```
docker compose up -d
```

Was Compose automatisch macht

- Container-Namen
- internes DNS
- Start-Reihenfolge
- sauberes Stoppen

```
docker compose ps
docker compose logs
```

Interne Kommunikation testen

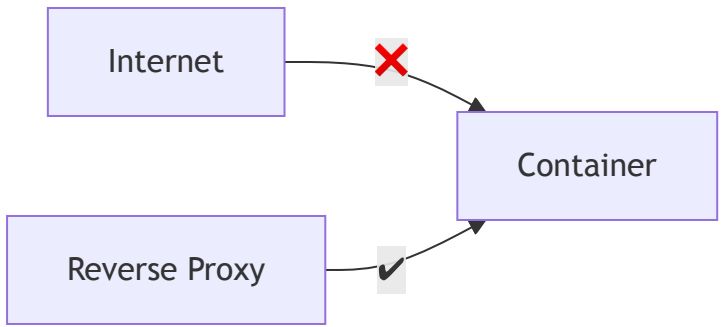
```
docker compose exec web nginx -v
```

Oder:

```
docker run --rm --network lab-net alpine ping web
```

Ports jetzt bewusst NICHT setzen

Warum?



➡️ Ports kommen zentral, nicht pro App
➡️ Das ist die Brücke zu Traefik (Tag 2)

Volumes kurz angerissen (Preview)

```
volumes:
  data:
```

Daten leben außerhalb des Containers
Container darf sterben
Daten bleiben

👉 Morgen wichtig für Apps & Backups

Typische Fehler & Debugging

❌ Container sehen sich nicht
→ gleiches Network?

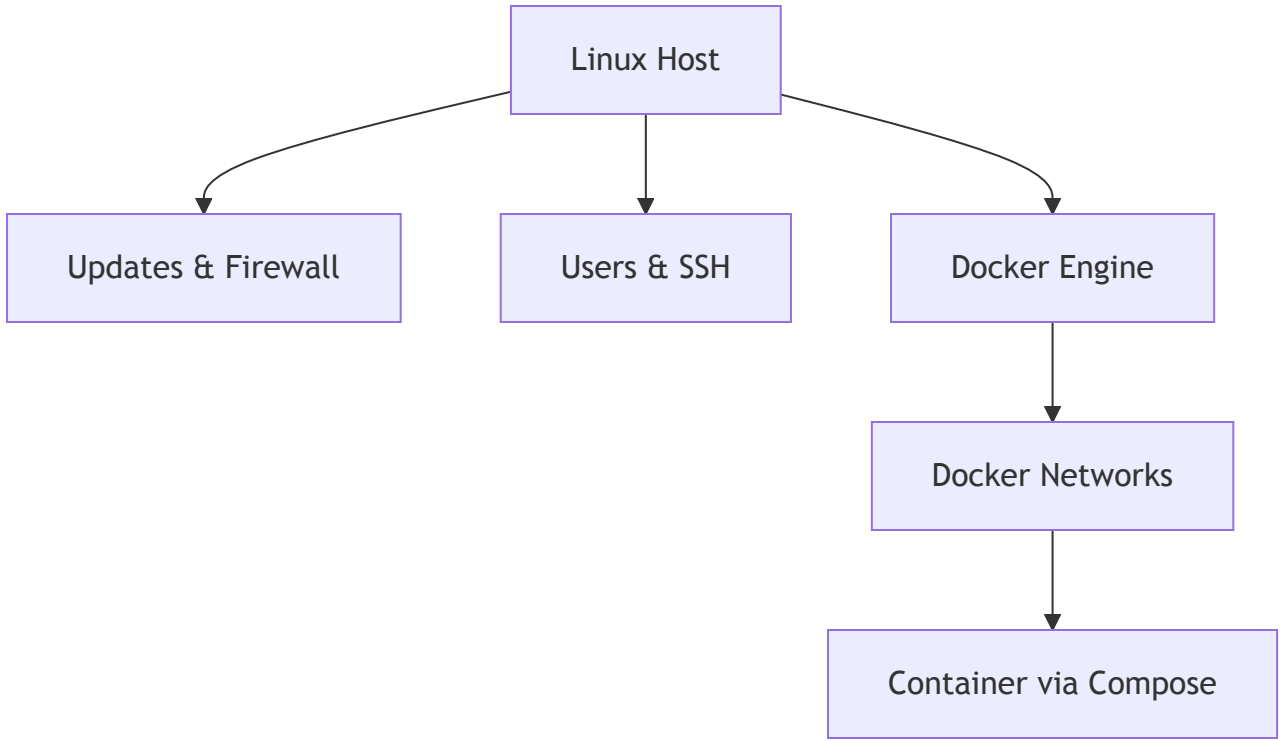
❌ Name funktioniert nicht
→ docker inspect

❌ Compose startet nicht
→ docker compose config

Checkpoint ✅ Abschnitt 6

- ✓ Docker Networking verstanden
- ✓ Container-Namen als DNS
- ✓ Compose als Standard
- ✓ Keine unnötigen Ports offen

Endzustand Tag 1 – Gesamtbild



👉 Fundament steht.

Abschluss Tag 1

Heute gelernt:

- Linux sicher bedienen
- Zugriffe kontrollieren
- Docker verstehen
- Infrastruktur strukturieren

Morgen: ➡️ Reverse Proxy
➡️ HTTPS & Cloudflare
➡️ Applikationen
➡️ Monitoring & Backup

💡 Bitte Server nicht löschen 🙂

Tag 2 – Applikationen & Betrieb

Secure Infrastructure Lab · Lakeside Park · CSAW × HoliSec

Heute machen wir den Server produktiv

- Services erreichbar machen
- Angriffsfläche klein halten
- Struktur statt Port-Chaos

Rückblick: Wo stehen wir?

Gestern aufgebaut:

- ✓ sicherer Linux-Host
- ✓ SSH Hardening & Firewall
- ✓ Docker & Docker Compose
- ✓ interne Docker-Netzwerke
- ✓ **keine offenen Ports außer SSH**

👉 Genau das brauchen wir jetzt.

Zielbild für heute

Am Ende von **Tag 2**:

- Ein **zentraler Einstiegspunkt** (Reverse Proxy)
- HTTPS überall
- Applikationen **nur intern**
- Monitoring & Backups als Basis
- Klarheit: *Was ist produktionsreif, was nicht?*

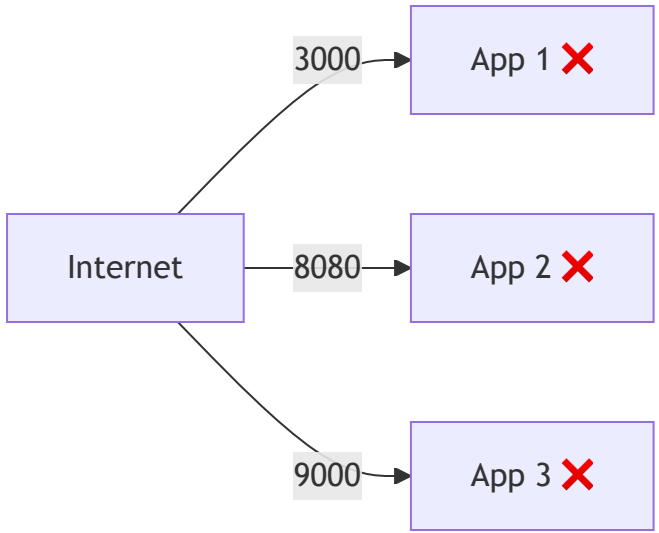
Warum wir heute nichts „einfach öffnen“

Typischer Anfänger-Reflex:

```
docker run -p 3000:3000 app
docker run -p 8080:8080 app
docker run -p 9000:9000 app
```

Fühlt sich schnell an – ist aber technische Schulden.

Das Port-Chaos-Problem



Probleme:

- keine Übersicht
- kein TLS
- jede App selbst verantwortlich
- Firewall-Regeln explodieren

Security-Sicht: Jeder Port ist Risiko

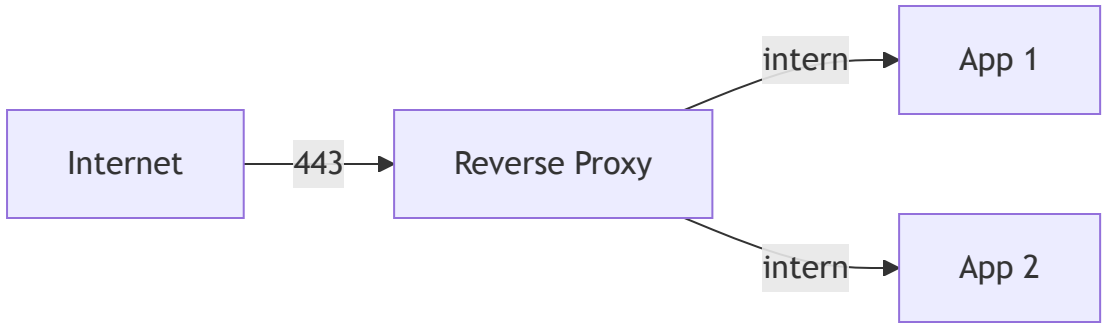
Merksatz:

> **Ein offener Port ist ein Versprechen an das Internet.**

Jeder Port bedeutet:

- neue Angriffsfläche
- neue CVEs
- neue Logs
- neue Wartung

Die Lösung: Reverse Proxy (Grundidee)



Nur eine Stelle:

- spricht mit dem Internet
- macht TLS
- macht Routing

Was ein Reverse Proxy NICHT ist

❌ keine Firewall
❌ kein Auth-System
❌ kein IDS/IPS

Er ist:

- Verkehrsverteiler
- TLS-Endpunkt
- Kontrollpunkt

Warum Traefik (und nicht Nginx)

Kurzer Reality-Check:

Nginx:

- statische Configs
- Reloads
- viel Handarbeit

Traefik:

- Docker-native
- liest Labels
- dynamisch
- perfekt für Compose

👉 Für Self-Hosting & KMUs klarer Sieger.

Mental Model: Traefik hört Docker zu

Was Traefik macht:

- beobachtet laufende Container
- liest deren Labels
- erstellt daraus automatisch Routing-Regeln

Wichtig:

> **Kein Reload.**
> **Keine statischen Configs.**
> **Labels = Wahrheit.**

Wichtige Design-Entscheidung (bitte merken)

> **Apps haben keine Ports nach außen.**

- keine `ports:` bei Apps
- keine Firewall-Regeln pro App
- alles geht über Traefik

Das ist der Grund, warum Tag 1 so wichtig war.

Hands-on-Vorbereitung (noch nichts starten!)

Bevor wir irgendwas deployen, prüfen wir:

```
docker ps
docker network ls
```

Wir erwarten:

- Docker läuft
- kein Traefik
- keine Apps exposed

Checkpoint ✅ Abschnitt 1

Wenn das passt, bist du bereit:

- ✓ du verstehst das Port-Problem
- ✓ du kennst die Reverse-Proxy-Idee
- ✓ du weißt, warum wir Traefik einsetzen
- ✓ dein System ist noch „sauber“

Nächster Schritt

Jetzt bauen wir Traefik minimal & bewusst:

- eigenes Docker-Netzwerk
- nur Port 80/443
- noch kein HTTPS
- Dashboard nur lokal

👉 weiter mit Abschnitt 2: Traefik Core Setup

Abschnitt 2
Traefik Core Setup (ohne TLS)

> Ziel:
> Traefik läuft, hört Docker zu
> und ist **der einzige Einstiegspunkt** nach außen.

Noch:

- ❌ kein HTTPS
- ❌ keine echten Apps
- ❌ kein Cloudflare

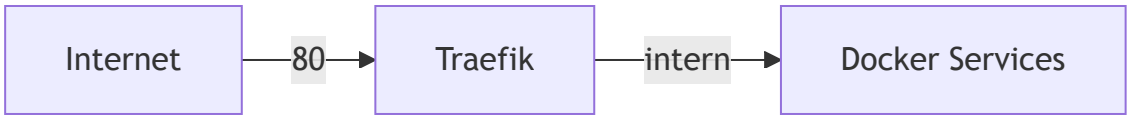
Design-Entscheidung (bitte merken)

> **Traefik ist Infrastruktur, keine App**

Das heißt:

- eigener Compose-Stack
- eigenes Docker-Netzwerk
- möglichst wenig Abhängigkeiten
- stabil & boring

Mental Model: Traefik als Torwächter



Alles, was nicht Traefik ist:

- bleibt intern
- bekommt keine Ports
- spricht nur über Docker-Netzwerke

Vorbereitung: Eigenes Proxy-Netzwerk

Warum ein eigenes Netzwerk?

- saubere Trennung
- Apps können mehrere Netze haben
- später wichtig für Security

```
docker network create proxy
```

Check:

```
docker network ls
```

Projektstruktur für Traefik

Wir arbeiten bewusst strukturiert:

```
mkdir -p ~/workshop/compose/traefik
cd ~/workshop/compose/traefik
```

```
touch docker-compose.yml
```

Traefik Minimal-Konfiguration (Teil 1)

```yaml
services:
  traefik:
    image: traefik:v3.0
    container_name: traefik
    command:
      - "--providers.docker=true"
      - "--providers.docker.exposedbydefault=false"
      - "--entrypoints.web.address=:80"
    ports:
      - "80:80"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
    networks:
      - proxy

networks:
  proxy:
    external: true
```

Wichtige Flags – warum sie existieren

```
--providers.docker=true
```

👉 Traefik liest Docker

```
--providers.docker.exposedbydefault=false
```

👉 Sicherheitskritisch! Container sind nicht automatisch öffentlich

```
--entrypoints.web.address=:80
```

👉 Ein Einstiegspunkt, kein Chaos

Docker Socket – kritisch, aber nötig

```
- /var/run/docker.sock:/var/run/docker.sock:ro
```

Was bedeutet das?

- Traefik kann Container sehen
- aber nicht verändern (read-only)

Merksatz:

> **Wer Docker Socket liest, vertraut Traefik.**

Traefik starten

```
docker compose up -d
```

Check:

```
docker ps
```

Du solltest sehen:

- traefik läuft
- kein anderer Service exposed

Test: Tut Traefik irgendwas?

```
curl http://localhost
```

Erwartung:

- 404 oder leere Antwort

👉 Das ist korrekt.
Traefik routet erst, wenn wir es sagen.

Traefik Dashboard – bewusst nur lokal

Jetzt nur zu Lernzwecken:

```
command:
  - "--api.insecure=true"
  - "--api.dashboard=true"
```

Port hinzufügen:

```
ports:
  - "80:80"
  - "8080:8080"
```

Restart:

```
docker compose up -d
```

Dashboard prüfen

Browser:

```
http://SERVER-IP:8080
```

Was sehen wir?

- EntryPoints
- Router (leer)
- Services (leer)

👉 Genau so soll es jetzt sein.

Security-Warnung

> **Traefik Dashboard darf NIE öffentlich sein**

Im Workshop:

- ok
- zum Lernen
- temporär

Produktiv:

- Dashboard nur intern
- oder gar nicht

Typische Fehler an dieser Stelle

❌ Docker Socket nicht gemountet
❌ falsches Netzwerk
❌ exposedbydefault=true
❌ Ports bei Apps öffnen

Debug:

```
docker logs traefik
```

Checkpoint ✅ Abschnitt 2

Wenn das passt:

- ✓ Traefik läuft stabil
- ✓ hört Docker zu
- ✓ nur Port 80 offen
- ✓ Dashboard sichtbar
- ✓ noch keine Apps exposed

Mentale Pause (wichtig!)

Bis hierher haben wir:

- keine App
- kein HTTPS
- kein DNS

Und trotzdem: ➡️ **das wichtigste Infrastruktur-Element steht**

Nächster Schritt

Jetzt wird Traefik nützlich:

➡️ Routing zu einer Test-App
➡️ Labels verstehen
➡️ „Wie kommt Traffic zur App?“

👉 weiter mit Abschnitt 3: Erste App über Traefik routen

Abschnitt 3
Erste App über Traefik routen

> Ziel:
> Eine **interne App** ist über Traefik erreichbar –
> ohne Ports, ohne TLS, nur mit Labels.

Was wir jetzt bauen

App läuft nur intern
Traefik entscheidet
Routing ist explizit

Wichtige Regel (nochmal!)

> **Apps bekommen keine ports:**

Wenn du `ports:` bei Apps siehst:

- falsches Setup
- sofort stoppen
- neu denken

Wahl der Test-App

Wir nehmen:

- klein
- stateless
- sofort sichtbar

👉 **traefik/whoami**

Warum?

- zeigt Request-Infos
- perfekt zum Lernen
- kein Setup nötig

Projektstruktur für Test-App

```
mkdir -p ~/workshop/compose/apps
cd ~/workshop/compose/apps
```

```
touch whoami.yml
```

whoami – Minimal Compose File

```yaml
services:
  whoami:
    image: traefik/whoami:latest
    container_name: whoami
    networks:
      - proxy
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.whoami.rule=Host(`whoami.localhost`)"
      - "traefik.http.routers.whoami.entrypoints=web"

networks:
  proxy:
    external: true
```

Labels – langsam & bewusst erklärt

```
traefik.enable=true
```

👉 Traefik darf diesen Container sehen

```
routers.whoami.rule=Host(`whoami.localhost`)
```

👉 Routing-Regel (Host-Header)

```
entrypoints=web
```

👉 Port 80

App starten

```
docker compose -f whoami.yml up -d
```

Check:

```
docker ps
```

DNS vorbereiten (lokal)

Da wir noch kein echtes DNS nutzen:

👉 auf deinem Laptop:

```
SERVER-IP   whoami.localhost
```

Datei:

- Linux/macOS: `/etc/hosts`
- Windows: `C:\Windows\System32\drivers\etc\hosts`

Test im Browser

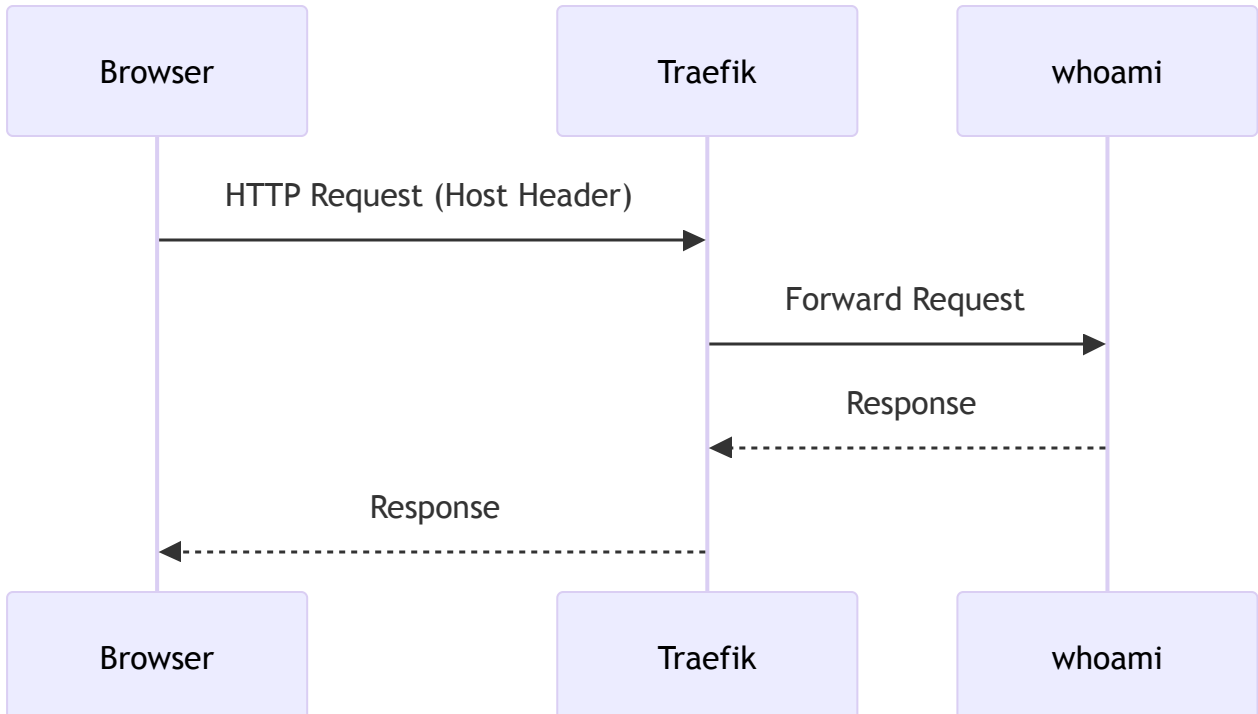
```
http://whoami.localhost
```

Erwartung:

- Seite mit Request-Infos
- Hostname sichtbar
- Container-Name sichtbar

🎉 **Das ist der Aha-Moment - Es geht um die http header!**

Was gerade passiert ist



Keine Ports - Keine IPs - Nur Namen & Regeln

Traefik Dashboard prüfen

```
http://SERVER-IP:8080
```

Was ist neu?

- Router: whoami
- Service: whoami
- EntryPoint: web

👉 Dashboard ist Debug-Tool, kein Feature

Typische Fehler & Fixes

❌ 404

- Hostname stimmt nicht

❌ App nicht sichtbar

- richtiges Network?

❌ Traefik reagiert nicht

- Labels korrekt?

Debug:

```
docker logs traefik
```

Wichtiges Learning (bitte merken)

> **Routing passiert über Labels, nicht Ports.**

Alles Weitere (TLS, Auth, Rate Limits):

- sind Erweiterungen
- bauen darauf auf

Cleanup (Disziplin!)

```
docker compose -f whoami.yml down
```

Warum?

- sauberes System
- keine Altlasten
- produktionsnahes Arbeiten

Checkpoint ✅ Abschnitt 3

- ✓ App läuft ohne Ports
- ✓ Routing per Hostname
- ✓ Traefik Dashboard verstanden
- ✓ Traffic-Flow klar

Nächster Schritt

Jetzt kommt die nächste Schicht:

➡️ HTTPS
➡️ echte Domains
➡️ Cloudflare
➡️ Zertifikate automatisch

👉 weiter mit Abschnitt 4: HTTPS & Cloudflare (DNS Challenge)

Warum wir jetzt umbauen

Bis jetzt:

- einzelne Compose-Files
- Lern-Setups
- schnelle Experimente

Ab jetzt:

- produktionsnahe Struktur
- klare Verantwortlichkeiten
- wiederauffindbar in 6 Monaten

Merksatz:

> **Ordnerstruktur ist Teil der Security.**

Mental Model: Ordner = Verantwortung

- jede Komponente hat ihren Platz
- Infrastruktur ≠ Applikationen
- Daten leben getrennt von Code

Ziel:

> **Ich weiß sofort, wo ich etwas ändern oder sichern muss.**

Umbau der Grundstruktur

```
/opt|home/docker
├── traefik/
│   ├── docker-compose.yml
│   ├── config/
│   │   ├── conf.d/*extra konfigs*
│   │   └── traefik.yml
│   └── data/
│       └── *später für Zertifikate*
├── apps/
│   ├── whoami/
│   │   ├── docker-compose.yml
│   │   └── data/
│   │
│   ├── application1/
│   │   ├── docker-compose.yml
│   │   └── data/
│   │
│   └── application2/
│       ├── docker-compose.yml
│       └── data/
└── backups/
    └── data/
    └── databases/
```

✅ Slide 5 – Erklärung: Was liegt wo (kurz & klar)

```
# Was liegt wo - und warum

traefik/
- zentrale Infrastruktur
- einziger Einstiegspunkt
- Zertifikate & Routing

apps/
- jede App für sich
- eigenes Compose
- eigene Daten

backups/
- bewusst sichtbar
- kein „vergessenes Feature“
```

Warum sich das später auszahlt

- Backups klar definierbar
- einfache Migration auf neue Server
- weniger Angst vor Updates
- bessere Übergabe an andere Admins

Das ist:

- sauber
- professionell
- replizierbar

👉 genau das, was Infrastruktur sein soll

Neues Docker Compose

```yaml
services:
  traefik:
    image: traefik:latest
    container_name: traefik
    ports:
      - "80:80" #expose http
      - "8080:8080" #expose dashboard
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./config/traefik.yaml:/etc/traefik/traefik.yaml:ro #read access to traefik config
      - ./config/conf.d/:/etc/traefik/conf.d/:ro # additional traefik configurations
    networks:
      - proxy
    restart: unless-stopped
networks:
  proxy:
    external: true
```

Traefik Config

```yaml
global:
  checkNewVersion: false
  sendAnonymousUsage: false

log:
  level: DEBUG

api:
  dashboard: true
  insecure: true

entryPoints:
  web:
    address: :80

providers:
  docker:
    endpoint: "unix:///var/run/docker.sock"
    network: proxy
    exposedByDefault: false
```

https://doc.traefik.io/traefik/

Abschnitt 4
HTTPS & Cloudflare – sauber und reproduzierbar

Ziel:

Traefik stellt **automatisch gültige TLS-Zertifikate** aus

– ohne offene Challenge-Ports

– ohne App-spezifische TLS-Configs.

Warum HTTPS nicht optional ist

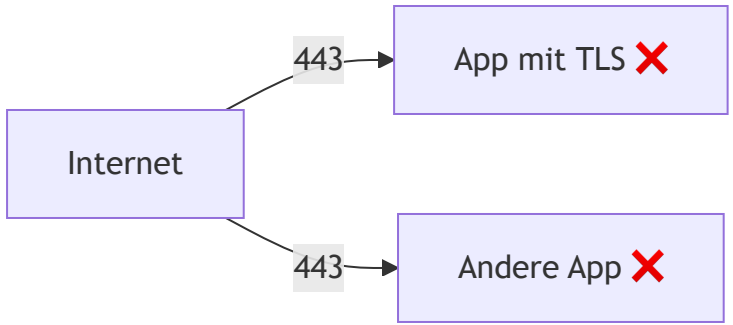
Ohne HTTPS:

- Passwörter im Klartext
- Session Hijacking
- Browser-Warnungen
- kein Vertrauen

Merksatz:

> **Alles, was über das Internet geht, ist TLS-terminiert.**

Klassischer HTTPS-Fail (bitte nicht!)



Probleme:

- Zertifikate pro App
- unklare Zuständigkeiten
- Ablauf vergessen
- Downtime

Zielbild: Zentrales TLS

Nur Traefik spricht TLS
Apps bleiben simpel
Zertifikate automatisiert

Rolle von Cloudflare (klar abgrenzen)

Cloudflare ist:

- DNS
- DDoS-Schutz
- Proxy (optional)

Cloudflare ist nicht:

- deine Firewall
- dein TLS-Endpunkt (bei Full Strict)

👉 Wir nutzen Cloudflare für DNS & API-Zugriff

Warum DNS-Challenge?

Vorteile:

- kein Port 80 nötig
- Firewall-freundlich
- Wildcard-Zertifikate möglich
- perfekt für Server ohne Public HTTP

Vorbereitung: Domain & DNS

Voraussetzung:

- Domain bei Cloudflare
- Nameserver zeigen auf Cloudflare
- Zugriff auf Zone

👉 Noch keine Records nötig

Cloudflare API Token erstellen

Benötigte Rechte:

- Zone - DNS - Edit
- Zone - Zone - Read

Wichtig:

- Scoped Token
- kein Global API Key

💡 Token niemals committen

Token sicher speichern (Server)

```
vim ./traefik/.env
```

```
CF_DNS_API_TOKEN=xxxxxxxxxxxxxxx
```

```
# optional permissions einschränken
chmod 600 ./traefik/.env
```

Wichtig! Secrets sollen nicht geteilt werden und auch nicht in einem Source Code Management aufpoppen.

Warum wir Secrets trennen

Merksatz:

> **Secrets gehören nicht ins Compose-File.**

Vorteile:

- kein Leak im Git
- klarer Ort
- einfacher Wechsel

Traefik für TLS vorbereiten

```
entryPoints:
  web:
    address: :80
    http:
      redirections:
        entryPoint:
          to: websecure
          scheme: https
  websecure:
    address: :443

certificatesResolvers:
  cloudflare:
    acme:
      email: "christian.gubesch@gmail.com"
      storage: /var/traefik/certs/cloudflare-acme.json
      caServer: "https://acme-v02.api.letsencrypt.org/directory"
      dnsChallenge:
        provider: cloudflare
        resolvers:
          - "1.1.1.1:53"
          - "8.8.8.8:53"
```

Optional TLS Config

```
vim .traefik/config/conf.d/tls.yaml
```

```
tls:
  options:
    default:
      minVersion: VersionTLS12
      sniStrict: true
      curvePreferences:
        - CurveP256
        - CurveP384
        - CurveP521
      cipherSuites:
        - TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
        - TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
        - TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
        - TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
        - TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
        - TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305
```

Docker Compose anpassen

```yaml
services:
  traefik:
    image: traefik:latest
    container_name: traefik
    ports:
      - "80:80"
      - "443:443" #expose https
      - "8080:8080"
    environment:
      - CF_DNS_API_TOKEN=${CF_DNS_API_TOKEN} #Cloudflare API Token with access to edit DNS entries for certain domains
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./config/traefik.yaml:/etc/traefik/traefik.yaml:ro
      - ./config/conf.d/:/etc/traefik/conf.d/:ro
      - ./data/certs:/var/traefik/certs:rw #certificate store -> this should be backuped
    networks:
      - proxy
    restart: unless-stopped
networks:
  proxy:
    external: true
```

Traefik neu starten

```
docker compose down
docker compose up -d
```

Logs beobachten:

```
docker logs -f traefik
```

👉 Noch keine Zertifikate – das ist korrekt.

HTTPS passiert erst bei Routing!

Wichtig:

> **Traefik holt Zertifikate nur, wenn ein Router TLS verlangt.**

➡️ Das machen wir gleich mit einer Test-App

Test-App erneut – jetzt mit TLS

Wir nehmen wieder whoami, aber HTTPS.

```
labels:
  - "traefik.enable=true"
  - "traefik.http.routers.whoami.rule=Host(`whoami.example.com`)"
  - "traefik.http.routers.whoami.entrypoints=websecure"
  - "traefik.http.routers.whoami.tls.certresolver=cloudflare"
```

DNS Record setzen

In Cloudflare:

- Type: A
- Name: whoami
- IP: SERVER-IP
- Proxy: DNS only (graue Wolke)

💡 Für den Workshop absichtlich kein Proxy

Optional wäre es auch möglich einen wildcard DNS record zu setzen

Cooles Zusatzfeature

Wenn man in Cloudflare keine manuellen DNS Records setzten mag kann man:

- Lokalen DNS Server verwenden und Record setzten
- In Hosts File DNS Eintrag hinzufügen
- DNS Challange funktioniert auch mit Sub-Sub Domainen

💡 Wichtig ist eigentlich nur, dass der HTTP Header stimmt!

Für uns heißt das eigentlich jede Applikation bekommt eigene Subdomain.

Es wäre auch möglich /Pfade zu verwenden ohne Subdomainen

Test im Browser

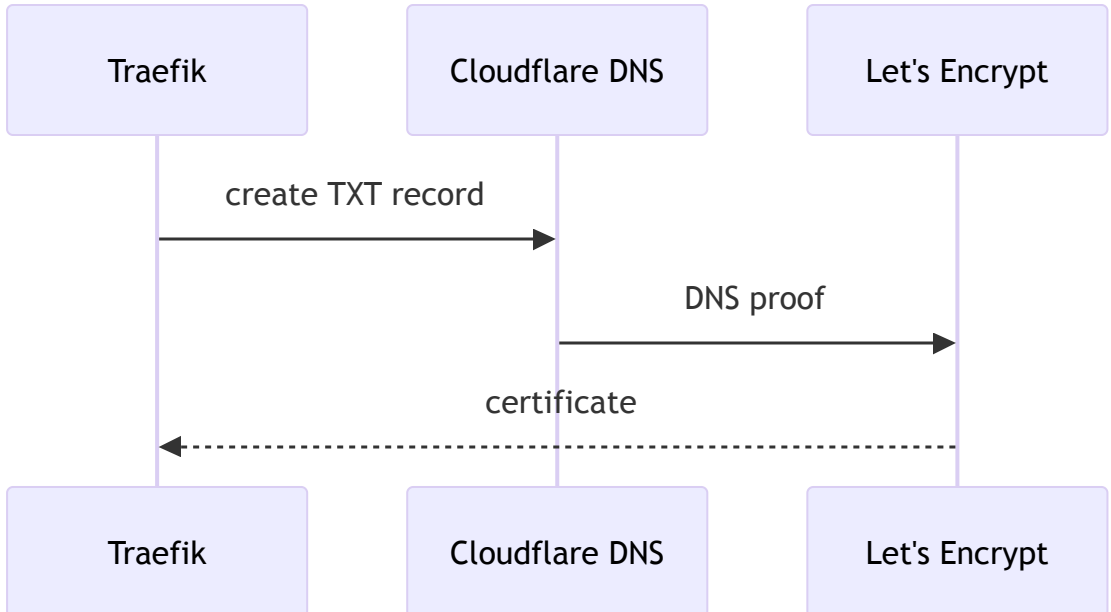
```
https://whoami.example.com
```

Erwartung:

- gültiges Zertifikat
- Let’s Encrypt als Issuer
- Seite lädt ohne Warnung

🎉 **HTTPS steht.**

Was gerade passiert ist



Kein HTTP-Challenge.
Kein Port-Freigaben.

Typische Fehler & Debugging

❌ Zertifikat kommt nicht

- DNS korrekt?
- Token-Rechte?

Logs:

```
docker logs traefik | grep acme
```

❌ Permission denied auf acme.json

- chmod 600

Security-Hinweis (wichtig!)

Zertifikate sind Secrets:

- acme.json niemals öffentlich
- niemals ins Git

Checkpoint ✅ Abschnitt 4

- ✓ Cloudflare DNS verstanden
- ✓ DNS-Challenge erklärt
- ✓ Traefik stellt Zertifikate aus
- ✓ HTTPS funktioniert
- ✓ Apps bleiben TLS-frei

Nächster Schritt

Jetzt wird es real:

➡️ echte Applikationen
➡️ Volumes & Daten
➡️ produktionsnahe Compose-Files

👉 weiter mit Abschnitt 5: Erste produktive Apps

Abschnitt 5
Produktive Applikationen – sauber & wartbar

> Ziel:
> Reale Services laufen **stabil**, **HTTPS-gesichert**
> und sind **wartbar & backupfähig**.

Heute:

- ✓ Uptime Kuma (Monitoring)
- ✓ Vaultwarden (Passwortmanager)
- (optional) Stirling PDF als Demo

Vor dem Start: Grundregeln für Apps

Bitte merken:

- ❌ keine `ports:` bei Apps
- ✓ immer eigenes Volume
- ✓ Konfig über ENV, nicht im Container
- ✓ ein Service = ein Compose-Block
- ✓ klare Namen

Merksatz:

> **Container sind Wegwerfware – Daten nicht.**

Struktur für produktive Apps

Pro App:

```
apps/
├── uptime-kuma/
│   ├── docker-compose.yml
│   └── data/
├── vaultwarden/
│   ├── docker-compose.yml
│   └── data/
```

➡️ Ein Repo, viele kleine Stacks

Teil A

Uptime Kuma – Monitoring als erstes

> Warum wir mit Monitoring starten

Wenn Monitoring fehlt, merkst du Fehler zu spät
Monitoring ist kein Luxus
Erste App = einfach & sichtbar

Uptime Kuma – Compose File (1)

```
mkdir uptime-kuma
cd uptime-kuma
vim docker-compose.yml
```

```
services:
  uptime-kuma:
    image: louislam/uptime-kuma:latest
    container_name: uptime-kuma
    restart: unless-stopped
    volumes:
      - ./data:/app/data
    networks:
      - proxy
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.uptime.rule=Host(`uptime.example.com`)"
      - "traefik.http.routers.uptime.entrypoints=websecure"
      - "traefik.http.routers.uptime.tls=true"
      - "traefik.http.routers.uptime.tls.certresolver=cloudflare"

networks:
  proxy:
    external: true
```

Uptime Kuma – Compose File (2)

```yaml
services:
  uptime-kuma:
    image: louislam/uptime-kuma:latest
    container_name: uptime-kuma
    restart: unless-stopped
    volumes:
      - uptimekuma-data:/app/data #use docker volume
      - /var/run/docker.sock:/var/run/docker.sock # container monitoring
    networks:
      - proxy
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.uptime.rule=Host(`uptime.example.com`)"
      - "traefik.http.routers.uptime.entrypoints=websecure"
      - "traefik.http.routers.uptime.tls=true"
      - "traefik.http.routers.uptime.tls.certresolver=cloudflare"

volumes:
  uptimekuma-data:
    driver: local

networks:
  proxy:
    external: true
```

Warum diese Punkte wichtig sind

```
restart: unless-stopped
```

👉 Server reboot ≠ App down

```
volumes: # mount to file system
  - ./data:/app/data
volumes: # docker volumes
  - uptimekuma-data:/app/data
```

👉 Daten überleben Container - Optional kann man auch mit Docker Volumes arbeiten!

```
labels:
```

👉 Routing zentral, nicht im Code

App starten & prüfen

```
docker compose up -d
docker ps
```

Browser:

```
https://uptime.example.com
```

🎉 Erste produktive App steht.

Permissions-Check (wichtig!)

```
docker inspect uptime-kuma | grep -A5 Mounts
```

Volume gehört Docker:

- nicht chmod 777
- Docker regelt Ownership

Merksatz:

> **Volumes nicht manuell anfassen, außer du weißt warum.**

Erste sinnvolle Checks anlegen

In Uptime Kuma:

- HTTPS-Check auf eigene Domain
- Zertifikatsablauf prüfen
- Docker Container Überwachen
- später: externe Dienste

Checkpoint ✅ Uptime Kuma

- ✓ App läuft stabil
- ✓ HTTPS funktioniert
- ✓ Daten persistent
- ✓ kein Port offen

Teil B

Vaultwarden – Passwortmanager (Core Setup)

Warum Vaultwarden heikel ist

- speichert Secrets
- Internet-exponiert
- braucht saubere Defaults

👉 Wir machen nur Basis, kein Endausbau.

Vaultwarden – Compose File

```yaml
services:
  vaultwarden:
    image: vaultwarden/server:latest
    container_name: vaultwarden
    restart: unless-stopped
    environment:
      - SIGNUPS_ALLOWED = {$SIGNUPS_ALLOWED} #manual account creation
      - ADMIN_TOKEN={VAULTWARDEN_ADMIN_TOKEN} #access admin panel
      - DOMAIN={$DOMAIN} #domain of application
      - WEBSOCKET_ENABLED={$WEBSOCKET_ENABLED} #smoother application handling
    volumes:
      - ./vaultwarden:/data
    networks:
      - proxy
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.vault.rule=Host(`vault.example.com`)"
      - "traefik.http.routers.vault.entrypoints=websecure"
      - "traefik.http.routers.vault.tls=true"
      - "traefik.http.routers.vault.tls.certresolver=cloudflare"
networks:
  proxy:
    external: true
```

Vaultwarden starten

```
docker compose up -d
docker ps
```

Browser:

```
https://vault.example.com
```

➡️ Account manuell anlegen

Vaultwarden: Minimal-Hardening (Hinweis)

Nicht jetzt, aber merken:

- Admin Token
- 2FA
- Backup-Strategie
- regelmäßige Updates

Checkpoint ✅ Vaultwarden

- ✓ HTTPS aktiv
- ✓ Registrierungen kontrolliert
- ✓ Daten persistent
- ✓ keine offenen Ports

Typische Fehler bei Apps

❌ App startet nicht

- Logs prüfen:

```
docker compose logs
```

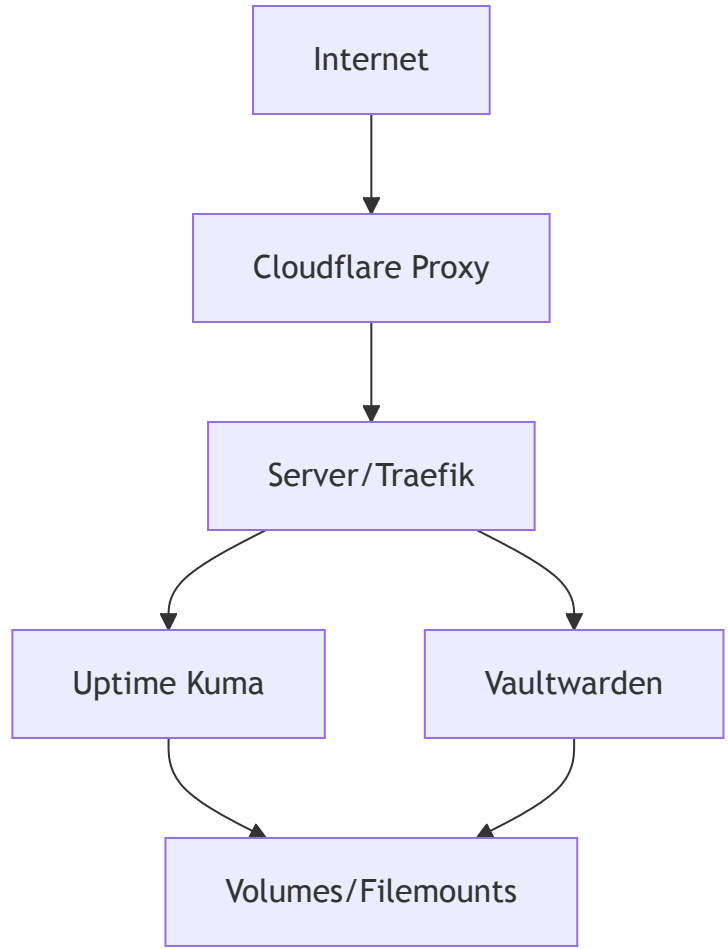
❌ HTTPS geht nicht

- Labels / DNS prüfen

❌ Daten weg

- Volume vergessen

Schematischer Aufbau



Checkpoint ✅ Abschnitt 5 (gesamt)

- ✓ mindestens eine produktive App
- ✓ sauberes Compose
- ✓ TLS zentral
- ✓ Volumes korrekt
- ✓ Struktur verständlich

Nächster Schritt

Jetzt kümmern wir uns um:

➡️ Backups (Duplicati)
➡️ Restore-Denken
➡️ Monitoring sinnvoll erweitern

👉 weiter mit Abschnitt 6: Backup & Betrieb (Basics)

Abschnitt 6

Backup & Betrieb – Basics, aber richtig

> Ziel:
> Daten sind gesichert, Ausfälle werden bemerkt
> und der Betrieb ist **vorhersehbar**, nicht reaktiv.

Die wichtigste Wahrheit zuerst

> **Backup ohne Restore-Test ist kein Backup.**

Viele Systeme:

- sichern fleißig
- testen nie
- fallen im Ernstfall durch

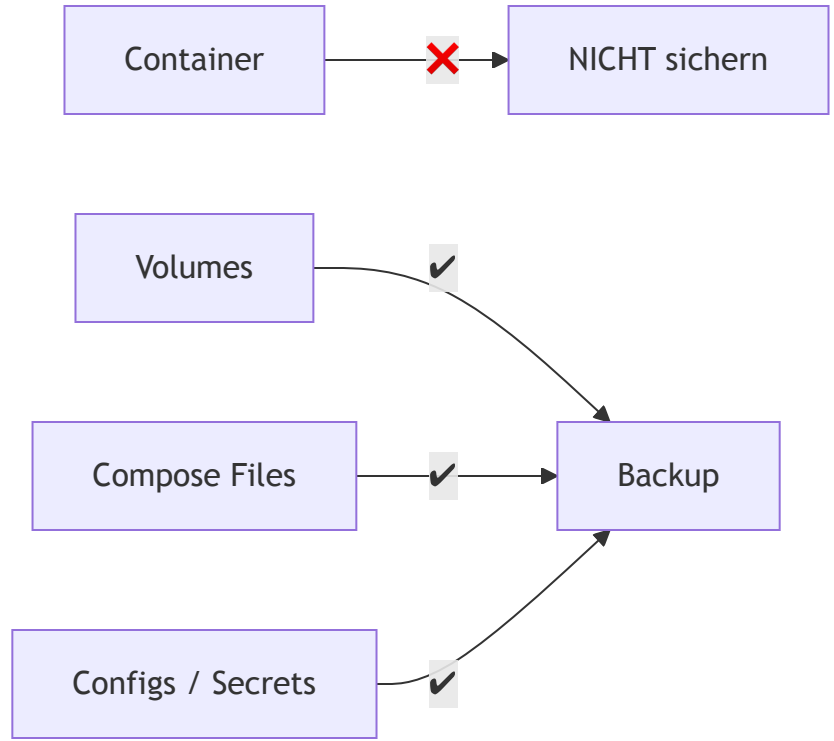
Nicht sichern:

- Images
- laufende Container

Sichern:

- Volumes
- Konfigs
- Compose-Files
- Secrets

Was wir wirklich sichern?



Backup-Ziel definieren (realistisch)

Fragen:

- Wie viel Datenverlust ist ok? (RPO)
- Wie lange darf Restore dauern? (RTO)
- Wo liegen die Backups?

👉 Für KMUs oft:

- täglich
- offsite
- automatisiert

Teil A

Duplicati – pragmatische Backup-Lösung

Warum Duplicati?

- Open Source
- Containerisiert
- Verschlüsselung
- viele Backends

Nicht perfekt – aber praxisnah.

Duplicati – Grundsetup

```yaml
services:
  duplicati:
    image: duplicati/duplicati:latest
    container_name: duplicati
    restart: unless-stopped
    volumes:
      - ./data:/data
      - /home/deploy/workshop:/source:ro
      - /home/deploy/backups:/backups
    networks:
      - proxy
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.duplicati.rule=Host(`backup.example.com`)"
      - "traefik.http.routers.duplicati.entrypoints=websecure"
      - "traefik.http.routers.duplicati.tls=true"
      - "traefik.http.routers.duplicati.tls.certresolver=cloudflare"

networks:
  proxy:
    external: true
```

Warum diese Mounts?

```
/source:ro
```

👉 Nur lesen, nie schreiben

```
/backups
```

👉 Lokales Ziel (Demo)

Produktiv:

- S3
- Storage Box
- Offsite Location

Duplicati starten

```
docker compose up -d
docker ps
```

Browser:

```
https://backup.example.com
```

Backup-Job anlegen (Demo)

In der UI:

- Source: /source
- Destination: /backups
- Verschlüsselung: immer
- Schedule: täglich
- Retention: z.B. 30 Tage

💡 Passwort sicher dokumentieren!

Restore-Denken (Pflicht!)

Gedankenexperiment:

- Volume gelöscht
- Server neu

Was brauchst du?

Antwort:

- Compose-Files
- Volumes
- Secrets

👉 Mindestens einmal Restore testen!

Teil B

Monitoring sinnvoll nutzen

Monitoring

Ziel:

- wissen, dass etwas kaputt ist
- nicht alles messen

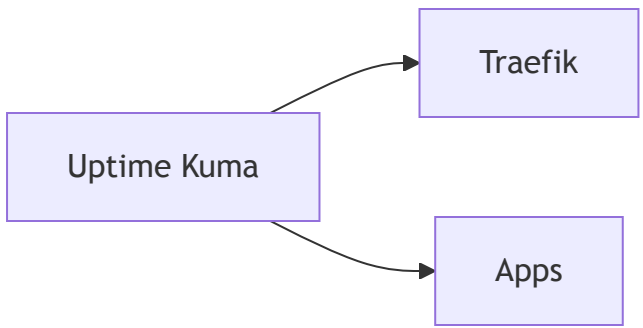
Was ist mit Security Monitoring?

- Büchse der Pandora
- Patches of Server
- Backups
- Applications/Images

Sinnvolle Checks mit Uptime Kuma

Checks:

- HTTPS Endpoint
- Zertifikatsablauf
- Traefik erreichbar
- App erreichbar



Typische Monitoring-Fehler

❌ zu viele Checks
❌ niemand schaut rein
❌ keine Alerts

Merksatz:

> **Ein Alarm ohne Reaktion ist Lärm.**

Betrieb: Updates & Lifecycle

Fragen, die ihr euch stellen müsst:

- Wer macht Updates?
- Wann?
- Wie teste ich vorher?
- Was passiert bei Fehlern?

Überlegung

Eigentlich muss Monitoring extern passieren nie am selben Server.

Idee: Eigener Server mit reinen Monitoring/Operations Task

Beispiel Stack:

- Uptime Kuma als Basis
- PatchMon als Patch Management
- Wazuh als Security Plattform

Demo PatchMon

https://patchmon.smesecurity.eu

Container-Updates – ehrlich betrachtet

Optionen:

- manuell (docker compose pull)
- halbautomatisch
- vollautomatisch (Watchtower)

👉 Watchtower = Diskussion, kein Muss

Optional: Security Monitoring mittels trivy https://trivy.dev/

Logs & Debugging im Alltag

```
docker compose logs
docker logs traefik
journalctl -u docker
```

👉 Logs sind erste Hilfe, nicht letzter Ausweg.

Ausblick für Security und Operations

- Authentik (SSO)
- CrowdSec
- Rate Limiting
- Read-only Container
- IDS / SIEM
- CI/CD
- Ansible/GitOps

Warum?

> **Erst Betrieb im Griff, dann Ausbau.**

Wann DIY endet

DIY gut für:

- kleine Teams
- Lernphasen
- überschaubare Risiken

Managed sinnvoll bei:

- SLA
- Compliance
- wenig Personal
- hoher Schaden bei Ausfall

Abschluss – Takeaways

- Infrastruktur ist Systemdenken
- Security ist Prozess, kein Tool
- Weniger ist oft mehr
- Verstehen schlägt Kopieren

Danke & offene Fragen 🙌

Feedback via Menti: https://menti.com

Was nehmt ihr mit?
Was würdet ihr produktiv so machen?
Wo würdet ihr Hilfe holen?

CSAW × HoliSec
Lakeside Science & Technology Park

Advanced Outlook

Wenn das Fundament steht

Diese Themen sind **bewusster Ausblick**

- 👉 keine Konfiguration
- 👉 keine Hands-on-Pflicht
- 👉 Fokus: **Verständnis & Entscheidungsfähigkeit**

Warum Advanced erst später kommt

Merksatz:

> **Komplexität ist kein Feature.**

Advanced-Setups machen nur Sinn, wenn:

- Betrieb stabil läuft
- Basics automatisiert sind
- Verantwortlichkeiten klar sind

Entscheidungsleitfaden

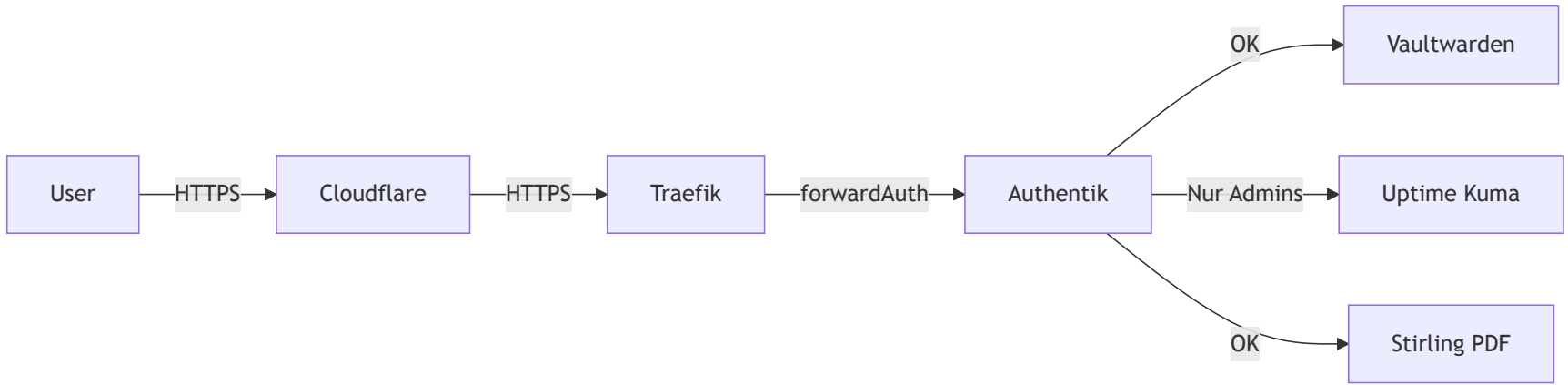
Frage dich immer:

- Welches Problem löst das?
- Für wen?
- Zu welchem Preis (Komplexität)?

Abschnitt A

Identity & Access – Zentrale Anmeldung

Use Case: SSO / Identity Gateway (Authentik)



Problem, das gelöst wird:

- viele Apps, viele Logins
- Offboarding schwierig
- keine zentrale Policy

Wann ist das sinnvoll?

- 5+ Apps
- mehrere User
- MFA / Rollen nötig
- Auditing

Was Authentik NICHT ist

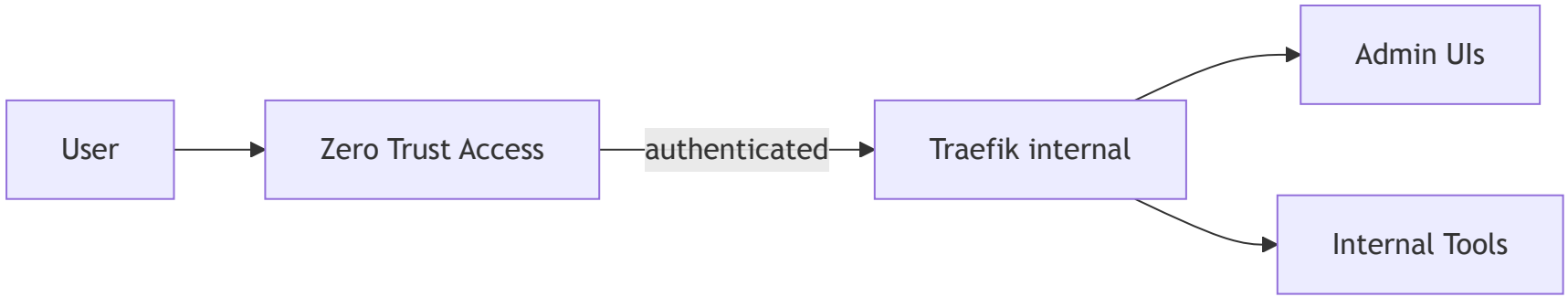
❌ kein Passwortmanager
❌ kein VPN
❌ kein Ersatz für App-Security

👉 Es ist ein Zugangs-Gateway (IAM).

Abschnitt B

Zero Trust – Keine Apps öffentlich

Use Case: Zero-Trust-Zugriff



Problem, das gelöst wird:

- Admin-UIs im Internet
- VPN unhandlich
- Zugriff schwer steuerbar

Wann ist das sinnvoll?

- Admin-Oberflächen
- kleine, definierte Teams
- minimaler Internet-Footprint

Zero Trust ≠ VPN

VPN:

- Netzwerkzugang
- oft „alles oder nichts“

Zero Trust:

- Applikationszugang
- Identitätsbasiert
- fein granuliert

Abschnitt C

CrowdSec – Automatische Reaktion auf Angriffe

Use Case: CrowdSec + Traefik + SSH

Problem, das gelöst wird:

- Bruteforce
- Scanner
- Credential Stuffing

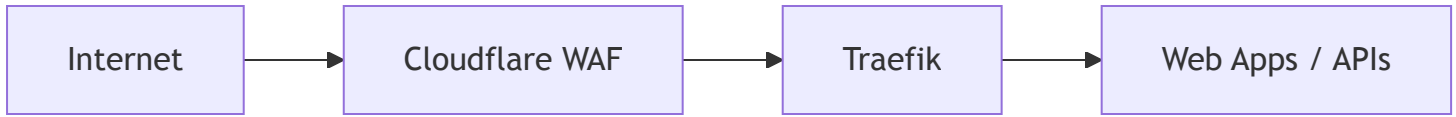
Wann sinnvoll?

- öffentliche Login-Seiten
- SSH öffentlich erreichbar
- wiederkehrender Angriffs-Traffic

Abschnitt D

WAF & Rate Limiting – Schutz vor Abuse

Use Case: WAF vor Applikationen



Problem, das gelöst wird:

- Bot-Traffic
- Formular-Abuse
- API-Missbrauch

Wann sinnvoll?

- öffentliche APIs
- Kontaktformulare
- Login-Endpunkte

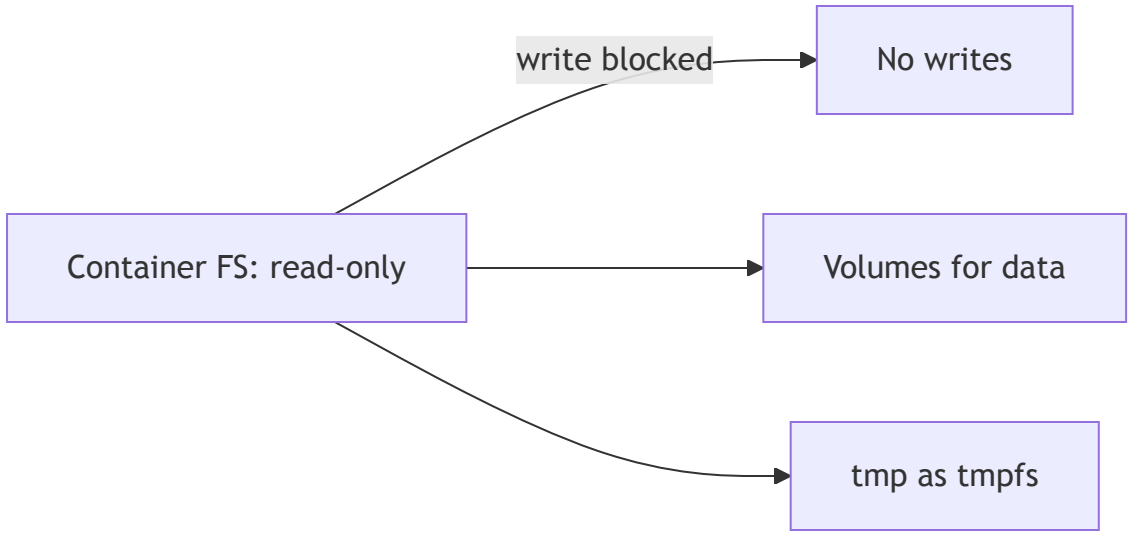
WAF ≠ Sicherheitsgarantie

- reduziert Noise
- verhindert triviale Angriffe
- ersetzt keine sichere App

Abschnitt E

Immutable / Read-only Container

Use Case: Read-only Container



Problem, das gelöst wird:

- Schadcode im Container
- unkontrollierte Writes

Wann sinnvoll?

- einfache Webapps
- APIs
- Compliance-Anforderungen

Realitätsscheck

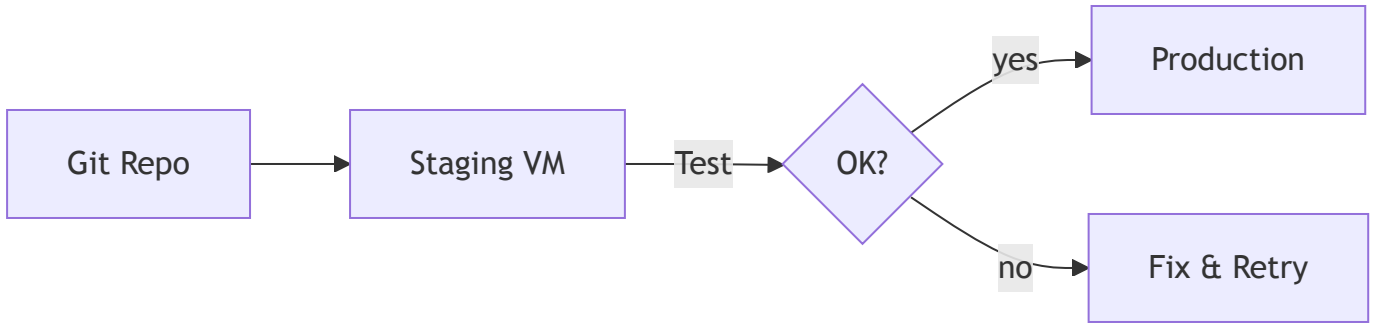
- nicht jede App unterstützt das
- erhöht Setup-Komplexität
- Debugging schwieriger

👉 Optionales Hardening

Abschnitt F

Update- & Release-Strategie

Use Case: Staging to Produktion



Problem, das gelöst wird:

- kaputte Updates
- Downtime
- Freitagabend-Deployments

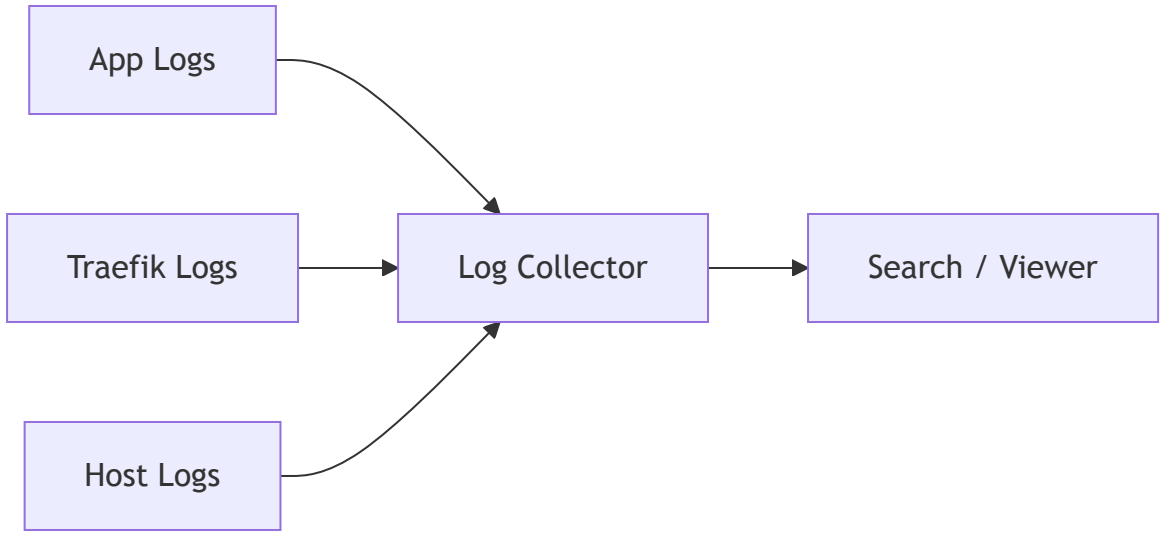
Wann sinnvoll?

- mehrere produktive Apps
- Business-kritische Systeme

Abschnitt G

Zentrales Logging

Use Case: Logs an einem Ort



Problem, das gelöst wird:

- Debugging über SSH
- Logs verstreut

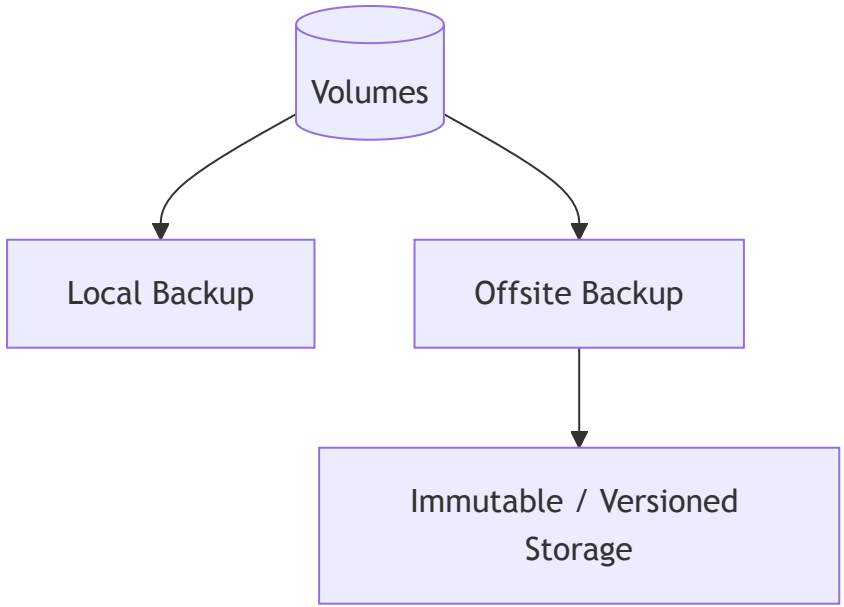
Wann sinnvoll?

- viele Services
- Audit-Anforderungen
- häufige Fehleranalyse

Abschnitt H

Backup-Strategie – 3-2-1

Use Case: 3-2-1 Backup (praxisnah)



Problem, das gelöst wird:

- Ransomware
- Datenverlust
- menschliche Fehler

Wann sinnvoll?

- kritische Daten
- rechtliche Anforderungen
- längere Aufbewahrung

Zusammenfassung – Advanced heißt nicht besser

Merksätze zum Mitnehmen:

- Stabil schlägt komplex
- Security ist ein Prozess
- Nicht jedes Tool passt zu jedem Team
- Know your limits

Nächste Schritte für euch

Fragen für den Transfer:

- Was brauchen wir wirklich?
- Was können wir betreiben?
- Was lagern wir aus?

Danke fürs Mitmachen 🙌

CSAW × HoliSec
Lakeside Science & Technology Park

Fragen? Diskussion? Austausch?