devoteam

Podman Training

Devoteam Germany
Innovative Tech

Innovative technology consulting for business.

What you will learn.

- Containerisierungskonzepte verstehen
- Containerisierte Dienste verwalten
- Container und Images verwalten
- Eigene Container-Images erstellen
- Containerisierte Anwendungen erstellen und ausführen
- Mehrere Container-Anwendungen erstellen und ausführen (Pods)
- Fehlerbehebung und Troubleshooting von Container-Problemen
- Podman Compose verwenden
- Podman Play Kube verwenden
- Podman Desktop
- Buildah und Skopeo

1

Containerisierungskonzepte verstehen

Die Welt der Container verstehen:

Themenübersicht:

1. **Das Konzept der Containerisierung:** Was bedeutet das eigentlich?
2. **Container vs. Virtuelle Maschinen (VMs):** Ein wichtiger Unterschied.
3. **Die Stärken der Containerisierung:** Warum dieser Ansatz so beliebt ist.
4. **Podman als Werkzeug:** Eine kurze Vorstellung.

Was sind eigentlich Container?

- **Stellen Sie sich vor:** Sie möchten Software (z.B. eine Webanwendung) einfach und zuverlässig von einem Computer zum anderen bringen.
- **Die Idee des Containers:** Eine Art "Software-Transportbox".

- **Was ist drin?**
  - Die eigentliche Anwendung (Ihr Code).
  - Alle notwendigen "Zutaten", damit die Anwendung läuft:
    - Laufzeitumgebungen (z.B. Java, Python).
    - Systembibliotheken und andere Abhängigkeiten.

- **Der Clou:** Diese "Box" läuft überall gleich, egal ob auf dem Laptop des Entwicklers, einem Testserver oder in der Cloud. Keine bösen Überraschungen mehr durch unterschiedliche Umgebungen!

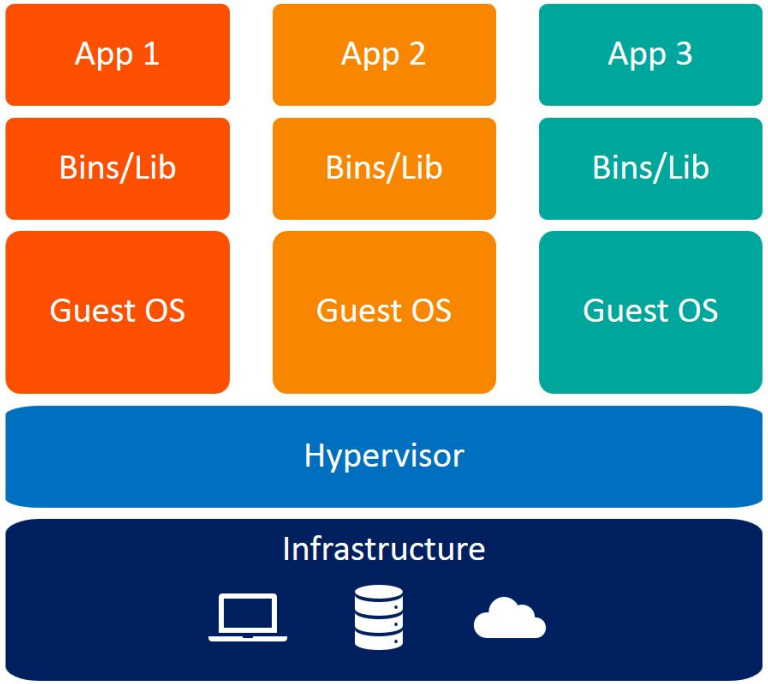
Container oder Virtuelle Maschine (VM)? Ein Vergleich.

Um Container besser zu verstehen, schauen wir uns den Unterschied zu Virtuellen Maschinen an:

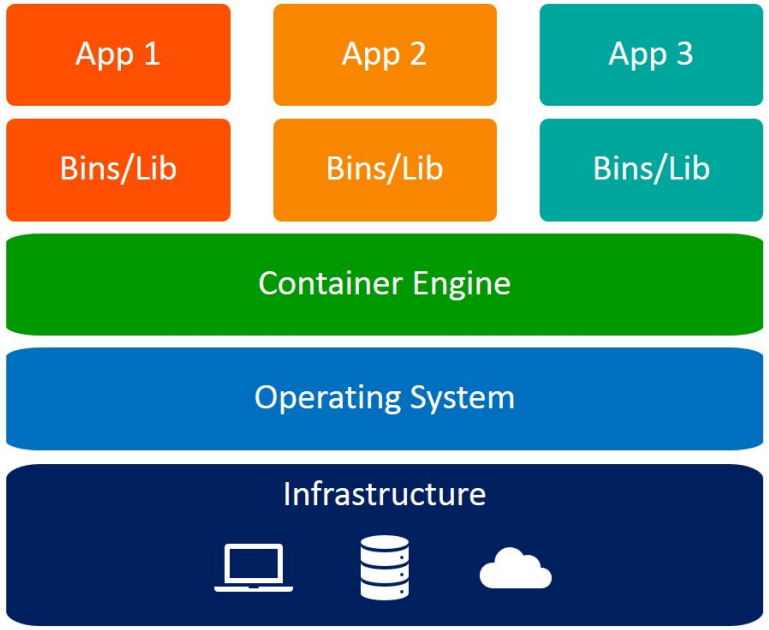
- **Virtuelle Maschinen (VMs):**
  - Stellen einen **kompletten Computer** digital nach (inklusive eigenem Betriebssystem-Kern / Kernel).
  - Laufen auf einer Software namens "Hypervisor".
  - Sind eher "schwergewichtig": brauchen mehr Ressourcen (Speicher, CPU) und starten langsamer.
  - Bieten eine sehr starke Trennung (Isolation) voneinander.

- **Container:**
  - Teilen sich den **Betriebssystem-Kern des Host-Computers**.
  - Enthalten nur die Anwendung und ihre direkten Abhängigkeiten.
  - Sind "leichtgewichtig": starten schnell und verbrauchen weniger Ressourcen.
  - Bieten eine gute Trennung auf Prozessebene.

Der Unterschied auf einen Blick



Virtual Machines



Containers

Kurz gesagt: Container sind oft die effizientere Lösung, wenn es darum geht, Anwendungen schnell und ressourcenschonend bereitzustellen und zu betreiben.
VMs haben ihre Berechtigung, wenn eine vollständige Betriebssystem-Isolation erforderlich ist.

Die Vorteile der Containerisierung

- **Konsistenz:** Eine Anwendung verhält sich überall gleich – von der Entwicklung bis zum Live-Betrieb. Das erspart viele "Es hat aber auf meinem Rechner funktioniert!"-Momente.
- **Isolation:** Verschiedene Anwendungen (oder verschiedene Versionen derselben Anwendung) können problemlos nebeneinander auf demselben System laufen, ohne sich gegenseitig zu stören.
- **Skalierbarkeit:** Benötigt eine Anwendung mehr Leistung, können einfach mehr identische Container gestartet werden.
- **Portabilität:** Ein Container läuft auf jedem System, das eine kompatible Container-Software installiert hat.
- **Effizienz:** Weniger Ressourcenverbrauch und schnellere Startzeiten im Vergleich zu VMs.

Podman: Ein Werkzeug zur Container-Verwaltung

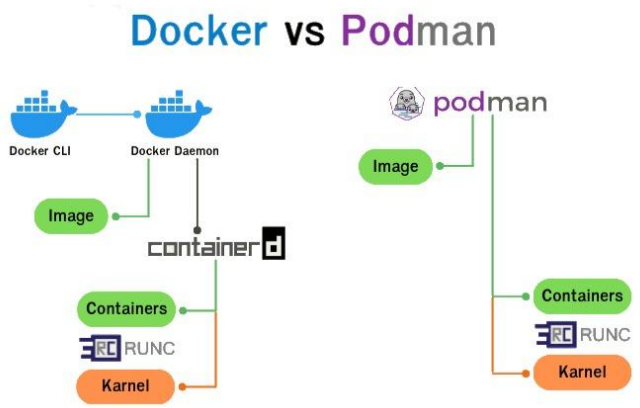
- **Podman ist...** eine Software (eine "Container Engine"), mit der Sie Container auf Linux-Systemen erstellen, verwalten und ausführen können.

- **Besondere Merkmale von Podman:**
  - **Ohne zentralen Dienst (Daemonless):**
    - Im Gegensatz zu anderen Werkzeugen (wie Docker) benötigt Podman keinen ständig im Hintergrund laufenden Prozess.
    - Jeder Podman-Befehl ist ein eigenständiger Prozess.
    - Das kann die Sicherheit erhöhen und die Systemintegration vereinfachen.
  - **Fokus auf Sicherheit (Rootless):**
    - Podman ist darauf ausgelegt, Container standardmäßig ohne Administratorrechte (root) auszuführen. Ein großes Plus für die Sicherheit!
  - **Kompatibilität:**
    - Die Befehle für Podman sind denen von Docker sehr ähnlich, was den Umstieg erleichtert.
  - **Pods:**
    - Podman unterstützt "Pods" – eine Möglichkeit, mehrere eng verwandte Container als eine Einheit zu gruppieren und zu verwalten (ähnlich wie in Kubernetes).

Podman: Wie funktioniert "Daemonless"?

- **Traditionelle Container-Tools (mit Daemon):**
  - Man tippt einen Befehl ein (z.B. docker start mein_container).
  - Dieser Befehl spricht mit einem zentralen "Manager"-Programm (dem Daemon), das ständig im Hintergrund läuft.
  - Der Daemon kümmert sich dann um den Container.
  - ***Nachteil:*** Der Daemon ist ein zentraler Punkt; wenn er Probleme hat oder angegriffen wird, betrifft das alles.

- **Podman (Daemonless):**
  - Man tippt einen Befehl ein (z.B. podman start mein_container).
  - Podman startet direkt die notwendigen Prozesse, um den Container zu verwalten, ohne einen Umweg über einen ständig laufenden Hintergrunddienst.
  - ***Vorteile:*** Weniger komplex, potenziell sicherer, verbraucht im Ruhezustand keine Ressourcen für einen Daemon.



Zusammenfassend: Wichtige Konzepte zum Mitnehmen

- **Container:** Sind wie standardisierte "Software-Pakete", die eine Anwendung und alles, was sie zum Laufen braucht, enthalten. Sie sorgen für Konsistenz und Portabilität.
- **Container vs. VMs:** Container sind leichtgewichtiger und teilen sich den Kern des Host-Betriebssystems, während VMs komplette Systeme emulieren.
- **Vorteile:** Konsistenz, Isolation, Skalierbarkeit, Portabilität und Effizienz sind die Hauptgründe für den Einsatz von Containern.
- **Podman:** Ein modernes, sicheres Werkzeug zur Verwaltung von Containern unter Linux, das ohne einen zentralen Daemon auskommt und den Betrieb ohne Root-Rechte favorisiert.

Exercises:
1 - Understanding Containerization Concepts

2

Containerisierte Dienste verwalten

Agenda des Themas:

Themenübersicht:

1. **Was sind "Containerized Services"?** Eine kurze Wiederholung.
2. **Der Lebenszyklus eines Dienstes:** Was gehört alles dazu?
3. **Podman als Management-Werkzeug:**
    - Status überwachen.
    - Details einsehen.
    - Protokolle (Logs) lesen.
    - Dienste steuern (Start, Stopp, Neustart).
    - Was passiert bei Fehlern? Neustart-Strategien.
4. **Ein Blick ins Innere:** Prozesse im Container.

Was sind "Containerized Services"?

- **Erinnerung aus Thema 1:** Container bündeln Anwendungen und ihre Abhängigkeiten.

- **"Containerized Services"** sind einfach diese Anwendungen, die in Containern laufen und einen bestimmten Dienst anbieten.
  - **Beispiele:**
    - Ein Webserver, der Webseiten ausliefert.
    - Eine Datenbank, die Daten speichert und bereitstellt. *(Daten selbst nicht im Container!)*
    - Eine Programmierschnittstelle (API), die Anfragen entgegennimmt und beantwortet.
    - Jede andere Art von Anwendung, die als Dienst konzipiert ist.

- **Die Vorteile bleiben:**
  - **Isolation:** Der Dienst läuft getrennt von anderen Anwendungen.
  - **Portabilität:** Der Dienst kann leicht auf andere Systeme verschoben werden.
  - **Konsistenz:** Der Dienst verhält sich überall gleich.

Der Lebenszyklus eines Container-Dienstes

Wenn wir eine Anwendung als Dienst in einem Container betreiben, müssen wir uns um ihren gesamten "Lebenszyklus" kümmern:

- **Starten:** Den Container mit der Anwendung initiieren.
- **Überwachen:**
  - Läuft der Dienst noch? (Status)
  - Was macht der Dienst gerade? (Protokolle/Logs)
  - Gibt es Probleme?
- **Steuern:**
  - Den Dienst anhalten (Stoppen).
  - Einen gestoppten Dienst wieder starten.
  - Den Dienst neu starten (z.B. nach Konfigurationsänderungen).
- **Warten/Aktualisieren:** (Wird in späteren Themen genauer behandelt)
  - Eine neue Version des Dienstes einspielen.
- **Entfernen:** Den Container (und ggf. seine Daten) beseitigen, wenn er nicht mehr benötigt wird.

Podman bietet Werkzeuge für all diese Phasen.

Podman: Werkzeuge zur Dienst-Verwaltung – Ein Überblick - 1

Podman stellt eine Reihe von Befehlen bereit, um den Zustand und das Verhalten Ihrer laufenden Container-Dienste zu kontrollieren:

- **Status überprüfen (`podman ps`):**
  - Zeigt an, welche Container gerade laufen.
  - Listet auch gestoppte Container (mit einer Option).
  - Gibt wichtige Infos wie Container-ID, Name, verwendetes Image und Port-Weiterleitungen.

- **Details einsehen (`podman inspect`):**
  - Liefert eine Fülle an technischen Detailinformationen zu einem Container.
  - Sehr nützlich für die Fehlersuche oder um genaue Konfigurationseinstellungen (Netzwerk, Speicher, Umgebungsvariablen) zu verstehen.
  - Die Ausgabe ist oft im JSON-Format, was maschinell gut verarbeitbar ist.

Podman: Werkzeuge zur Dienst-Verwaltung – Ein Überblick - 2

- **Protokolle/Logs einsehen (`podman logs`):**
  - Zeigt die Standardausgabe und Fehlermeldungen, die von der Anwendung im Container erzeugt werden.
  - Essentiell für die Fehlersuche und um zu verstehen, was im Inneren des Dienstes passiert.
  - Logs können auch "live" mitverfolgt werden.

- **Dienste steuern:**
  - `podman stop`: Hält einen laufenden Container an (sendet erst ein "freundliches" Signal, dann ein "hartes").
  - `podman start`: Startet einen zuvor gestoppten Container wieder.
  - `podman restart`: Hält einen Container an und startet ihn sofort neu.
  - `podman rm`: Entfernt einen **gestoppten** Container.

Mehrere Instanzen eines Dienstes

Kann ich denselben Dienst mehrfach starten?

- **Ja!** Das ist ein häufiger Anwendungsfall.
  - Zum Beispiel, um verschiedene Versionen parallel zu testen.
  - Oder um die Last auf mehrere Instanzen zu verteilen (obwohl das oft fortgeschrittenere Techniken erfordert).

- **Wichtige Überlegungen:**
  - **Eindeutige Namen:** Jeder Container sollte einen eigenen Namen haben (z.B. mein-webserver-v1, mein-webserver-v2).
  - **Port-Konflikte vermeiden:** Wenn Dienste über das Netzwerk erreichbar sein sollen (z.B. ein Webserver), muss jede Instanz einen eigenen, freien Port auf dem Host-Computer verwenden.
    - **Beispiel:** Instanz 1 auf Host-Port 8080, Instanz 2 auf Host-Port 8081.

Ein Blick ins Innere: Prozesse im Container

- Manchmal möchte man schnell sehen, welche Programme (Prozesse) *direkt im Container* gerade laufen.
- Ist der Hauptdienst gestartet? Laufen vielleicht unerwartete Prozesse?
- **`podman top` ermöglicht genau das:**
  - Zeigt eine Liste der aktiven Prozesse innerhalb eines bestimmten Containers.
  - Ähnlich dem bekannten Linux-Befehl top oder ps, aber auf den Container beschränkt.
  - Nützlich für eine schnelle Diagnose, ohne sich erst per Shell in den Container verbinden zu müssen.

Was, wenn ein Dienst ausfällt? Neustart-Strategien (`--restart`)

Für wichtige Dienste, die möglichst immer verfügbar sein sollen, ist es entscheidend, ihr Verhalten bei Fehlern oder Neustarts zu definieren.

- **Die Frage:** Was soll Podman tun, wenn ein Container unerwartet stoppt oder der gesamte Rechner neu startet?
- **Die Lösung: Neustart-Richtlinien (`--restart` Option beim Starten)**
  - `no` (Standard): Keine automatischen Neustarts.
  - `on-failure`: Container wird neu gestartet, wenn er mit einem Fehler beendet wurde. Man kann auch eine maximale Anzahl an Neustartversuchen festlegen.
  - `always`: Container wird immer neu gestartet (nach Fehlern, manuellem Stopp durch Podman, oder Systemneustart).
  - `unless-stopped`: Ähnlich wie always, aber der Container wird nicht neu gestartet, wenn er explizit vom Benutzer gestoppt wurde.
- **Warum ist das wichtig?** Es erhöht die Ausfallsicherheit und Verfügbarkeit Ihrer Dienste.

Wichtige Konzepte zum Mitnehmen

- **Container-Dienste managen** bedeutet, ihren gesamten Lebenszyklus von Start bis Entfernung zu kontrollieren.
- **Podman bietet Kernbefehle** dafür:
  - `ps`: Status laufender (und aller) Container.
  - `inspect`: Detaillierte Konfigurationsinformationen.
  - `logs`: Ausgabe und Fehler der Anwendung im Container.
  - `stop`, `start`, `restart`, `rm`: Steuerung des Lebenszyklus.
- `podman top` gibt einen schnellen Einblick in die Prozesse, die innerhalb eines Containers laufen.
- **Neustart-Richtlinien (`--restart`)** sind entscheidend, um die Verfügbarkeit von Diensten bei Fehlern oder Systemneustarts sicherzustellen.

Exercises:

2 - Managing Containerized Services with Podman

3

Container und Images verwalten

Agenda des Themas:

Themenübersicht:

1. **Erweitertes Container-Management:**
   - Den Zustand und Details laufender/gestoppter Container präzise erfassen (`ps, inspect`).
   - Einblick in die "Gedanken" eines Containers: Log-Analyse (`logs`).
   - Direkter Draht: Befehle in laufenden Containern ausführen (`exec`).
2. **Professionelles Image-Management:**
   - Images gezielt beziehen (`pull`) und deren Herkunft verstehen.
   - Die lokale Image-Sammlung überblicken (`images`) und organisieren.
   - Images versionieren und umbenennen mit Tags (`tag`).
   - Ordnung halten: Nicht mehr benötigte Images sicher entfernen (`rmi, image prune`).
3. **Grundlegende Diagnosestrategien:**
   - Ressourcennutzung im Blick behalten (`stats`).
   - Systematische Fehlersuche bei Container-Problemen.

Container-Status und Detailanalyse

- **Den Überblick behalten mit `podman ps`:**
  - `podman ps`: Zeigt Ihnen die aktuell **laufenden** Container – Ihr "Live-Dashboard".
  - `podman ps -a (--all)` : Enthüllt **alle** Container, auch die bereits **beendeten**.
  - Warum ist das wichtig? Beendete Container können Fehlerinformationen (Exit-Codes) enthalten oder für spätere Analysen relevant sein.

- **Tiefeninspektion mit `podman inspect`:**
  - `podman inspect <Container-Name oder ID>` : Liefert eine detaillierte JSON-Ausgabe mit **allen Konfigurationsdetails** eines Containers.
  - Netzwerkeinstellungen, gemountete Volumes, Umgebungsvariablen, der genaue Befehl, mit dem der Container gestartet wurde, und vieles mehr.
  - Anwendung: Unverzichtbar für fortgeschrittene Fehlersuche oder um die genaue Konfiguration eines laufenden Systems zu dokumentieren.
  - (**Grafikidee, falls nötig**: Eine stilisierte Lupe über einem Container-Symbol, die zu einem Ausschnitt eines JSON-Dokuments führt, um die Detailtiefe zu visualisieren.)

Kommunikation des Containers – Logs verstehen

- Jeder Container schreibt Ausgaben auf die Standardausgabe (stdout) und Standardfehlerausgabe (stderr).

- `podman logs <Container-Name oder ID>` : Zeigt genau diese Ausgaben an.
  - Standardmäßig werden alle bisherigen Logs angezeigt. Optionen wie `-f (--follow)` erlauben das Live-Verfolgen neuer Log-Einträge (ähnlich `tail -f`).
  - `--since, --until`: Filtern nach Zeitstempeln.

- **Warum sind Logs so entscheidend?**
  - **Fehlerdiagnose**: Die häufigste Ursache für Probleme steht oft in den Logs!
  - **Monitoring**: Verfolgen, was die Anwendung im Container gerade tut.
  - **Verständnis**: Nachvollziehen, wie eine Anwendung startet und arbeitet.

Interaktion im laufenden Betrieb: `podman exec`

- Manchmal müssen Sie einen Befehl innerhalb eines bereits laufenden Containers ausführen, ohne diesen zu stoppen oder neu zu starten.

- **Syntax:** `podman exec [OPTIONEN] <Container-Name oder ID> BEFEHL [ARGUMENTE...]`
  - **Beispiel:** `podman exec mein-webserver httpd -v` (um die Version des Apache-Servers im Container mein-webserver abzufragen).
  - **Beispiel:** `podman exec -it mein-datenbank-container bash` (um eine interaktive Shell im Datenbank-Container zu starten). Die Optionen -i (interaktiv) und -t (TTY) sind hier oft nützlich.

- **Anwendungsfälle für `podman exec`:**
  - **Live-Diagnose:** Überprüfen von Prozesslisten, Netzwerkverbindungen oder Dateiinhalten direkt im Container.
  - **Konfigurationsmanagement:** Kleine Änderungen an Konfigurationsdateien vornehmen (mit Vorsicht!).
  - **Troubleshooting:** Manuelles Starten von Diensten oder Skripten im Container.

Container-Images: Die "Baupläne" gezielt beschaffen

- Images sind die Vorlagen für Ihre Container und kommen meist von **Registries** (z.B. Docker Hub, Quay.io, interne Firmen-Registries).

- `podman pull [OPTIONEN] [REGISTRY/]REPOSITORY[:TAG|@DIGEST]`
  - **[OPTIONS]:** wie `--quiet` (weniger output) oder `--authfile` (private registry login), passen das Verhalten an.
  - **[REGISTRY/]:** Image host - z.B. docker.io (default) oder quay.io.
  - **REPOSITORY:** Image name - z.B. nginx oder redis.
  - **:TAG:** gibt die Version an—verwende spezifische (z.B. :1.21) oder **@Digest** (z.B. @sha256:abc123)

- **Beispiele:**
  - `podman pull redis:alpine` (Holt das redis Image mit dem alpine Tag).
  - `podman pull nginx:stable-alpine`

- `podman run` führt implizit einen `pull` aus, wenn das Image lokal nicht vorhanden ist. Ein expliziter `podman pull` gibt Ihnen jedoch mehr Kontrolle und ist oft der erste Schritt.

Lokale Image-Verwaltung: Überblick und Organisation

Die eigene Image-Bibliothek: `podman images` und `podman tag`

- **Bestandsaufnahme mit podman images:**
  - Zeigt alle lokal verfügbaren Images mit Repository, Tag, Image ID, Erstellungsdatum und Größe.
  - Hilft, den Überblick über die heruntergeladenen und selbst erstellten Images zu behalten.

- **Images tagging und versionieren mit `podman tag`:**
  - Tags sind Aliase oder menschenlesbare Namen für eine spezifische Image ID. Ein Image kann mehrere Tags haben.
  - **Syntax:** `podman tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]`
  - **Beispiel:** `podman tag httpd:alpine` mein-projekt/webserver:v1.0
  - **Wichtig:** `podman tag` erstellt keine Kopie der Image-Daten! Es fügt nur einen weiteren Verweis auf dieselben zugrundeliegenden Layer hinzu. Das ist extrem speichereffizient.
  - **Nutzen:** Versionierung (v1.0, v1.1, v2.0-beta), Kennzeichnung für Umgebungen (dev, staging, prod), Umbenennung für eigene Projekte.

Images clean-up: Platz schaffen und Ordnung halten

- Images können signifikanten Speicherplatz belegen.

- **Einzelne Images (oder Tags) entfernen mit `podman rmi`:**
  - `podman rmi IMAGE_NAME_ODER_ID[:TAG]`
  - Entfernt das angegebene Image oder nur den spezifischen Tag, falls das Image noch andere Tags hat.
  - **Achtung:** Funktioniert nur, wenn kein Container (auch kein gestoppter!) das Image bzw. den Tag verwendet. Mit der Option `-f (--force)` kann dies umgangen werden, ist aber mit Vorsicht zu genießen.

- **Automatisiertes Aufräumen mit podman image prune:**
  - `podman image prune`: Entfernt "dangling images" – das sind unreferenzierte Image-Layer, die oft nach dem Bau neuer Image-Versionen zurückbleiben und keinen Tag mehr haben.
  - `podman image prune -a (--all)` : Entfernt **alle ungenutzten Images**, d.h. Images, die von keinem Container (laufend oder gestoppt) verwendet werden.
    - Sehr nützlich, um regelmäßig Speicherplatz freizugeben. Aber seien Sie sicher, dass Sie die Images nicht doch noch benötigen!

- **Best Practices:** Regelmäßig podman image prune ausführen. `podman image prune -a` gezielt einsetzen.

Basis-Diagnose: Ressourcennutzung prüfen

- `podman stats [OPTIONEN] [CONTAINER...]`
  - Zeigt eine **Live-Ansicht** der Ressourcennutzung (CPU, Speicher, Netzwerk I/O, Block I/O) Ihrer laufenden Container.
  - Ohne Angabe eines Containers werden alle laufenden Container angezeigt.
  - `Ctrl+C` zum Beenden der Live-Ansicht.

- **Warum `podman stats` nutzen?**
  - Schnelle Überprüfung, ob ein Container "durchdreht" (z.B. 100% CPU).
  - Identifizieren von ressourcenintensiven Containern.
  - Grundlegendes Performance-Monitoring.

- **Hinweis zu Rootless & cgroups v2:**
  - Für die volle Funktionalität von `podman stats` (insbesondere die genaue Speichermessung) im Rootless-Modus ist ein Linux-System mit cgroups v2 erforderlich. Bei älteren Systemen (cgroups v1) können die angezeigten Werte unvollständig sein oder Warnungen erscheinen. Dies ist ein System-Setup-Thema, das die Funktionalität von Podman beeinflussen kann.

Debugging von Container-Startproblemen: Ein systematischer Ansatz

Wenn ein Container nicht wie erwartet startet oder sich seltsam verhält, gehen Sie methodisch vor:

- **Status und Exit-Code prüfen:** `podman ps -a`
  - Welchen Status hat der Container (z.B. Exited)?
  - Welchen Exit-Code hat er geliefert (ein Code ungleich 0 deutet meist auf einen Fehler hin)?

- **Logs konsultieren:** `podman logs <Container-Name>`
  - Was war die letzte Meldung des Containers? Oft steht die Ursache hier.

- **Konfiguration überprüfen:** `podman inspect <Container-Name>`
  - Sind Ports korrekt gemappt? Sind Volumes richtig eingebunden? Stimmen die Umgebungsvariablen?

- **Interaktiver Testlauf:**
  - Versuchen Sie, den Container interaktiv mit einer Shell zu starten, um die Umgebung zu untersuchen: `podman run -it --rm --entrypoint sh <Image-Name>`
  - Dies überschreibt den normalen Startbefehl und gibt Ihnen direkten Zugriff auf das Innere des Images, um Befehle zu testen oder Dateisysteme zu prüfen.

Zusammenfassung

- Wir können den **Lebenszyklus und die Details von Containern** mit `ps`, `inspect`, `logs` und `exec` präzise managen und analysieren.
- Das **Management von Images** umfasst das gezielte Holen (`pull`), das Auflisten (`images`), das sinnvolle Versionieren mittels Tags (`tag`) und das wichtige Aufräumen (`rmi`, `image prune`).
- Für die **grundlegende Diagnose** können wir die Ressourcennutzung mit `stats` überwachen und einen strukturierten Workflow zur Fehlersuche anwenden.
- **Wichtige Prinzipien:** Spezifische Image-Tags für Reproduzierbarkeit, regelmäßige Bereinigung zur Ressourcenschonung und das Verständnis der Log-Ausgaben als Schlüssel zur Problemlösung.

Exercises:

3 - Managing Containers and Images via Basic Diagnostics

4

Eigene Container-Images erstellen

Agenda des Themas:

Themenübersicht:

1. **Warum & Wie:** Die Motivation für eigene Images und das Containerfile als Bauplan.
2. **Kern-Anweisungen:** Die wichtigsten Werkzeuge im Containerfile.
3. **Der Build-Vorgang:** Vom Containerfile zum Image mit podman build.
4. **Optimierung & Sicherheit:** Kleine, schnelle und sichere Images bauen (Caching, Multi-Stage, Non-Root).
5. **Teilen & Versionieren:** Images mit Tags versehen und in Registries bereitstellen.

Warum eigene Images? Und wie?

- **Warum?**
  - Für **Ihre eigene Software** (Web-Apps, Skripte) benötigen Sie eine maßgeschneiderte Verpackung.
  - **Vorteile:** Konsistenz, Reproduzierbarkeit, einfache Verteilung, Optimierung.

- **Wie? Das `Containerfile` (oder `Dockerfile`)**
  - Ein **textbasierter Bauplan**, der Schritt für Schritt Ihr Image definiert.
  - Jede Anweisung erzeugt einen **Layer** (eine Schicht).
  - `# Basis festlegen`
  - `FROM alpine:latest`
  - `# Software installieren`
  - `RUN apk add --no-cache nginx`
  - `# Eigene Dateien kopieren`
  - `COPY meine-webseite.html /var/www/localhost/htdocs/`
  - `# Standard-Startbefehl`
  - `CMD ["nginx", "-g", "daemon off;"]`

Die wichtigsten Werkzeuge im Baukasten

- **FROM <basis-image>:** Der Startpunkt für Ihr Image.
- **RUN <befehl>:** Führt Befehle während des Builds aus (z.B. Software installieren).
- **COPY <quelle> <ziel>:** Kopiert Dateien von Ihrem Rechner ins Image.
- **WORKDIR /pfad:** Setzt das Arbeitsverzeichnis für nachfolgende Befehle.
- **ENV <variable>=<wert>:** Setzt Umgebungsvariablen für die Laufzeit.
- **USER <benutzername>:** Legt den ausführenden Benutzer fest (Sicherheit!).
- **CMD ["befehl"] / ENTRYPOINT ["befehl"]:** Definieren den Startbefehl des Containers.
- **EXPOSE <port>:** Dokumentiert den Port der Anwendung im Container.
- **LABEL <schlüssel>="<wert>":** Fügt Metadaten zum Image hinzu.

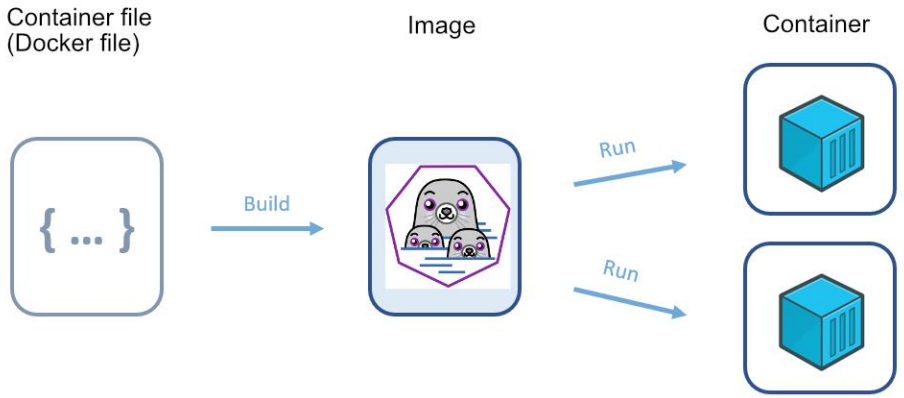
(Hinweis: Diese Auswahl dient dem schnellen Überblick. Details zu jeder Anweisung und weiteren Optionen folgen in der Praxis.)

Der Build-Prozess: Vom Text zum fertigen Image

- **Mit podman build -t mein-image:version . erstellen Sie Ihr Image.**
  - `-t`: Weist dem Image einen Namen und Tag (Version) zu.
  - . (der Punkt): Die **Build Context Directory** – alle Dateien hier stehen dem Build zur Verfügung.

- **.containerignore:**
  - Ähnlich wie .gitignore – schließt Dateien/Verzeichnisse vom Build Context aus.
  - **Wichtig für:** Kleine Build-Kontexte, Vermeidung unnötiger/sensibler Daten im Image.

- **Build Cache & Layering:**
  - Podman wiederverwendet Layer aus dem Cache, wenn sich Schritte nicht geändert haben -> schnelle Builds.
  - Ändert sich ein Schritt, werden dieser und alle folgenden Schritte neu gebaut.
  - **Optimierung:** Selten ändernde Befehle (z.B. Paketinstallationen) früh platzieren; häufig ändernde (z.B. COPY des App-Codes) spät.



Fortgeschrittene Konzepte für bessere Images

- **Multi-Stage Builds:** Der Königsweg für kleine, sichere Images.
  - **Builder-Stufe (`FROM ... AS builder`):** Kompilieren/Bauen der App mit allen Werkzeugen.
  - **Finale Stufe (`FROM ...`):** Start mit minimalem Basis-Image, dann `COPY --from=builder` **... nur die fertigen Artefakte** übernehmen**.**

- **`RUN`-Befehle optimieren:**
  - Befehle mit && verketten und im selben Schritt aufräumen (z.B. Paket-Caches), um Layer und Größe zu reduzieren.

- **Build-Argumente (`ARG <name>=<default>`):**
  - Anpassung des Build-Prozesses von außen über `--build-arg`.

- **Container Health Checks (`HEALTHCHECK CMD ...`):**
  - Periodische Prüfung, ob die Anwendung im Container noch gesund ist.

Versionierung und Verteilung

- **Tagging ist entscheidend** für das Management verschiedener Versionen:
  - z.B. mein-app:1.0, mein-app:latest.

- **Images in eine Registry pushen** (z.B. Docker Hub, Quay.io):
  - **Vollständiger Name: <registry>/<benutzer_oder_org>/<image-name>:<tag>**
  - **Workflow (vereinfacht):**
    - `podman login <registry>`
    - Image lokal bauen und mit vollständigem Registry-Pfad taggen: `podman build -t <registry>/<user>/mein-app:1.0 .` (oder `podman tag lokal-name:tag <registry>/<user>/mein-app:1.0`)
    - `podman push <registry>/<user>/mein-app:1.0`

Zusammenfassung

- Eigene Images (**`Containerfile`**) geben Kontrolle über Anwendungs-Deployments.
- Verstehen Sie Kernanweisungen und den podman build-Prozess.
- Optimieren Sie mit Cache-Nutzung, Multi-Stage Builds und sauberen **`RUN`**-Befehlen.
- Denken Sie an **`.containerignore`** und Non-Root-User für Sicherheit und Effizienz.
- Tags und Registries sind für Versionierung und Teilen unerlässlich.

Exercises:

5 - Building and Running Containerized Applications with Podman

5

Containerisierte Anwendungen erstellen und ausführen

Agenda des Themas:

Themenübersicht:

1. **Der typische Workflow:** Vom Code zum laufenden Container.
2. **Datenmanagement im Fokus:**
   - Host-Daten teilen: Bind Mounts.
   - Anwendungsdaten persistent speichern: **Named Volumes**.
3. **Konfiguration & Secrets:**
   - **Umgebungsvariablen**: Direkt (`-e`) und über Dateien (`--env-file`).
   - Sichere Geheimnisverwaltung mit **Podman Secrets.**
4. **Ressourcen & Sicherheit:**
   - Limits für CPU und Speicher (`--memory, --cpus`).
   - Schreibgeschütztes Dateisystem (`--read-only`) und temporärer Speicher (`--tmpfs`).

Vom Code zur laufenden Anwendung

Der Prozess, um eine eigene Anwendung in einem Container zum Laufen zu bringen, folgt oft diesem Muster:

- **Code schreiben:** Die eigentliche Anwendungslogik.
- **`Containerfile` erstellen:** Der Bauplan für Ihr Image (wie in Thema 4).
- **Image bauen:** `podman build -t mein-app:version .`
- **(Vorbereitung für Daten):** Ggf. Volumes (podman volume create) oder Secrets (podman secret create) erstellen.
- **Container starten:** `podman run [OPTIONEN] mein-app:version`
  - Hier kommen viele wichtige Optionen ins Spiel, um Ports zu mappen (`-p`), Daten einzubinden (`-v, --mount`), Umgebungsvariablen zu setzen (`-e`), Secrets zu injizieren (`--secret`) und Ressourcen zu limitieren.
- **Testen und Verwalten:** Überprüfen, ob alles wie erwartet läuft, Logs ansehen etc.

Persistente Daten: Host-Zugriff vs. Verwalteter Speicher

Wenn Container Daten speichern oder benötigen, die über ihre eigene Lebenszeit hinausgehen oder vom Host kommen, brauchen wir Speicherstrategien.

- **Option 1: Bind Mounts – "Der direkte Draht zum Host"**
  - **Was?** Ein existierendes Verzeichnis/Datei vom Host wird direkt in den Container gespiegelt. Änderungen sind sofort beidseitig sichtbar.
  - **Wann?**
    - **Entwicklung:** Lokalen Quellcode im Container nutzen.
    - **Host-Konfigurationen** in den Container geben.
  - **Syntax-Beispiele:** `-v ./host-code:/app` oder `--mount type=bind,source=./config,target=/etc/app/config,readonly`

- **Option 2: Named Volumes – "Die sichere Bank für Anwendungsdaten"**
  - **Was?** Von Podman verwalteter Speicherplatz, dessen Lebenszyklus unabhängig vom Container ist. Podman kümmert sich um den genauen Speicherort auf dem Host.
  - **Wann?** Für Daten, die vom Container erzeugt werden und persistent bleiben sollen (Datenbanken, Zähler, Uploads).
  - **Syntax-Beispiele:** `-v app-daten:/data` oder `--mount type=volume,source=meine-db-daten,target=/var/lib/mysql`
  - **Management:** `podman volume create/ls/inspect/rm/prune`
- **Generell:** Die `--mount` Syntax ist expliziter und oft bevorzugt gegenüber der kürzeren -v Syntax.

Konfiguration und Secrets managen

Anwendungen flexibel und sicher konfigurieren.

- **Umgebungsvariablen:** Einfacher Weg, Konfigurationen zu übergeben.
  - Direkt: `-e SCHLUESSEL=wert`
  - Aus Datei: `--env-file ./meine-app.env` (enthält SCHLUESSEL=wert Paare).

- **Secrets (Passwörter, API-Keys):** Sensible Daten erfordern spezielle Behandlung.
  - **Niemals direkt in Images oder Containerfile-ENV!**
  - **Podman-verwaltete Secrets:**
    - Secret erstellen: `echo "meinPasswort" | podman secret create mein-geheimnis -`
    - Im Container sicher bereitstellen mit `--secret ...`:
      - **Als Datei (Standard, sicher):** `source=mein-geheimnis,target=dateiname` (gemountet unter `/run/secrets/dateiname`). Die App liest die Datei.
      - **Als Umgebungsvariable:** `source=mein-geheimnis,type=env,target=MEINE_ENV_VAR` (Wert wird in podman inspect maskiert). Nützlich, wenn die App eine ENV-Variable erwartet.

Ressourcen kontrollieren und Sicherheit erhöhen

- **Ressourcenlimits: Verhindern Überlastung des Host-Systems.**
  - Speicher: `--memory 100m`
  - CPU: `--cpus 0.5`
  - *Hinweis:* Effektive Durchsetzung kann von cgroups-Version abhängen.

- **Schreibgeschütztes Root-Dateisystem (`--read-only`):**
  - **Starke Sicherheitsmaßnahme!** Das gesamte Container-Dateisystem wird schreibgeschützt.
  - Alle Pfade, in die geschrieben werden muss (z.B. für Logs, Daten), benötigen explizite beschreibbare Volumes oder tmpfs-Mounts.
  - Unterscheidet sich vom Mount-spezifischen `:ro` (das nur einen einzelnen Mount betrifft).

- **Temporärer Speicher (`--tmpfs /pfad-im-container` oder `--mount type=tmpfs,...`):**
  - Erstellt ein Dateisystem im Arbeitsspeicher; schnell, aber nicht persistent.

Zusammenfassung

- Containerisierte Anwendungen erfordern ein durchdachtes Management von **Daten (Persistenz & Austausch), Konfiguration** und **Secrets**.
- **Bind Mounts** für direkten Host-Zugriff (Entwicklung, Host-Konfigs); **named Volumes** für von Podman verwaltete, persistente Anwendungsdaten. Die `--mount` Syntax ist klarer.
- Konfiguration über **Umgebungsvariablen**; sensible Daten sicher mit **Podman Secrets**.
- **Ressourcenlimits** und ein `--read-only` Dateisystem steigern Stabilität und Sicherheit.

Exercises:

5 - Managing Containers and Images via Basic Diagnostics

6

Mehrere Container-Anwendungen erstellen und ausführen (Pods)

Agenda des Themas:

Themenübersicht:

1. **Die Herausforderung:** Warum benötigen wir mehr als einzelne Container?
2. **Grundlagen Podman Networking:** Wie Container (isoliert) kommunizieren.
    - Netzwerk-Namespaces, Default-Bridge.
    - **Benutzerdefinierte Netzwerke** für verbesserte Kommunikation via DNS.
3. **Einführung in Pods:**
    - Das Konzept: Gemeinsam genutzte Ressourcen, insbesondere das Netzwerk.
    - Der "Infra-Container".
    - Optionales Teilen anderer Namespaces (`pid`, `ipc`, `uts`).
4. **Arbeiten mit Pods:** Erstellen, Container hinzufügen, Lifecycle-Management.
5. **Wichtige Aspekte:** Healthchecks in Pods, Ressourcenlimits auf Pod-Ebene.
6. **Ausblick:** Deklaratives Management.

Von einzelnen Containern zu Multi-Container-Anwendungen

- Einfache Anwendungen mögen in einem einzigen Container laufen.
- Moderne Architekturen (z.B. Microservices) bestehen jedoch oft aus **mehreren, spezialisierten Diensten**:
  - Web-Frontend (z.B. Nginx, das statische Dateien ausliefert)
  - Backend-API (z.B. eine Python Flask-App)
  - Datenbank (z.B. MySQL, PostgreSQL)
  - Caching-Dienst, Message Queue etc.
- Diese Dienste müssen **miteinander kommunizieren** und oft **gemeinsam verwaltet** werden (Start, Stopp, Updates).
- **Herausforderung:** Wie koordiniert man diese einzelnen Container effizient und sicher?

Podman Networking Grundlagen (vor Pods)

- **Isolation durch Netzwerk-Namespaces:** Standardmäßig erhält jeder Container seinen eigenen, isolierten Netzwerkstack (eigene IP-Adresse, Routing-Tabelle etc.). Er ist vom Host und anderen Containern getrennt.

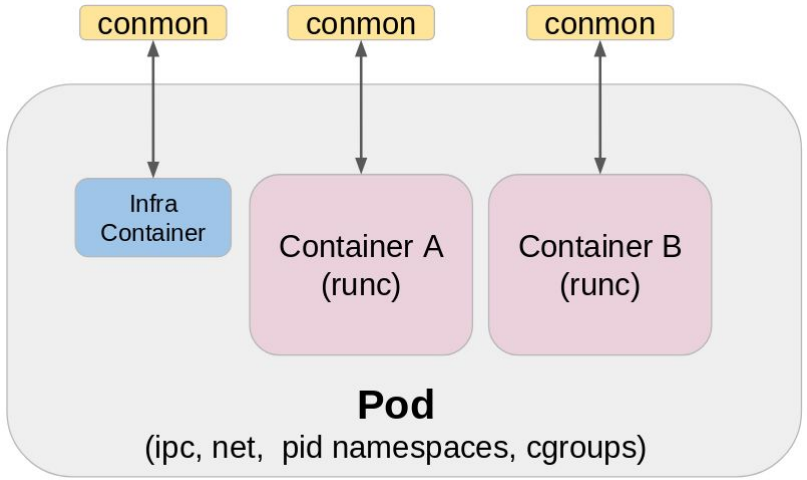
- **Default Bridge-Netzwerk (`podman`):**
  - Wenn keine spezielle Netzwerkkonfiguration erfolgt, verbindet Podman Container mit diesem Standardnetz.
  - Container erhalten eine IP, können ausgehende Verbindungen herstellen.
  - **Direkte Kommunikation über Namen ist hier standardmäßig nicht möglich.**

- **Port Mapping (`-p HOST_PORT:CONTAINER_PORT`):**
  - Macht einen Dienst, der im Container auf CONTAINER_PORT lauscht, außerhalb des Containers über HOST_PORT auf dem Host-System erreichbar.

- **Benutzerdefinierte Netzwerke (`podman network create mein-netz`):**
  - Für eine verbesserte Kommunikation zwischen Containern, die nicht in einem Pod laufen.
  - **Hauptvorteil: Eingebautes DNS.** Container im selben benutzerdefinierten Netzwerk können sich über ihren **Containernamen** als Hostnamen erreichen.
  - Beispiel: `podman run --network mein-netz --name server ...` und `podman run --network mein-netz --name client ...` -> Client kann http://server aufrufen.

Pods: Mehrere Container, eine gemeinsame Umgebung

- Ein **Pod** (abgeleitet von Kubernetes) ist eine Gruppe von einem oder mehreren Containern, die **wichtige Ressourcen teilen** und als eine Einheit verwaltet werden.
- **Wichtigste geteilte Ressource: Netzwerk-Namespace**
  - Alle Container in einem Pod teilen sich dieselbe IP-Adresse und denselben Port-Raum.
  - Sie können untereinander über `localhost` kommunizieren (z.B. Web-App auf `localhost:5000` spricht mit DB auf `localhost:3306`).
  - Port-Mappings (`-p`) werden auf **Pod-Ebene** definiert, nicht für einzelne Container im Pod.
- **Der "Infra-Container":**
  - Ein kleiner, meist unsichtbarer Container, der im Pod läuft. Seine Hauptaufgabe ist es, den Netzwerk-Namespace und andere geteilte Namespaces "offen zu halten", auch wenn keine anderen Anwendungscontainer im Pod laufen.
- **Struktur (vereinfacht):**



Pods erstellen und Container hinzufügen

Praktische Arbeit mit Pods

- **Pod erstellen:**
  - `podman pod create --name mein-pod -p HOST_PORT:POD_PORT`
  - Hiermit definieren Sie den Pod selbst und das Port-Mapping für den gesamten Pod.
- **Container zum Pod hinzufügen:**
  - `podman run -d --pod mein-pod --name container-a mein-image-a`
  - `podman run -d --pod mein-pod --name container-b mein-image-b`
  - Die Container werden innerhalb des Netzwerk-Namespace des Pods gestartet.
- **Lifecycle Management (Pod-Ebene):**
  - `podman pod ps` (Pods auflisten)
  - `podman pod inspect mein-pod` (Details ansehen)
  - `podman pod start/stop/restart/rm mein-pod` (gesamten Pod verwalten)
  - `podman ps --pod` (Container innerhalb von Pods auflisten)

Wichtige Aspekte bei Multi-Container-Anwendungen in Pods

- **Health Checks:**
  - Definieren Sie HEALTHCHECK-Anweisungen in Ihren Containerfiles (benötigt --format docker beim podman build).
  - Podman kann den Gesundheitszustand der einzelnen Container im Pod überwachen. `podman ps --pod` zeigt den Status (z.B. (healthy)).
  - Manuelle Prüfung mit `podman healthcheck run <container_name>` .

- **Ressourcenlimits auf Pod-Ebene:**
  - `podman pod create --memory 200m --cpus 1.0 ...`
  - Limitiert die Gesamtressourcen für alle Container im Pod.

- **Optionales Teilen anderer Namespaces (`--share <namespace>` bei `podman pod create`):**
  - `--share pid`: Container im Pod sehen dieselben Prozesse und können sich Signale senden.
  - `--share ipc`: Gemeinsame Inter-Process Communication Ressourcen.
  - `--share uts`: Gleicher Hostname.

Deklaratives Management

Für komplexere Anwendungen

- Während `podman pod create` und `podman run --pod` für einfache Setups gut funktionieren, werden für komplexere Multi-Container-Anwendungen oft **deklarative Ansätze** bevorzugt.
- Dabei beschreiben Sie den gewünschten Zustand Ihrer Anwendung (welche Images, Ports, Volumes, Netzwerke, Pods etc.) in einer Konfigurationsdatei.
- Podman unterstützt dies primär über **Kubernetes YAML-Dateien:**
- Andere Werkzeuge wie **podman-compose** (ähnlich docker-compose) existieren ebenfalls

Diese Ansätze erleichtern die Verwaltung, Versionierung und Reproduzierbarkeit komplexer Setups.

Zusammenfassung

- Moderne Anwendungen bestehen oft aus mehreren zusammenarbeitenden Containern.
- **Benutzerdefinierte Netzwerke** erlauben DNS-basierte Kommunikation zwischen einzelnen Containern.
- **Pods** gruppieren Container, die sich primär den Netzwerk-Namespace teilen (localhost-Kommunikation) und als Einheit verwaltet werden.
- **Ports** werden auf **Pod-Ebene** gemappt; Container mit `podman run --pod ...` hinzugefügt.
- Achten Sie auf **Healthchecks** und **Ressourcenlimits** (cgroup v2 für Pod-Level-Limits!).
- Für komplexe Anwendungen sind **deklarative Ansätze** (Kubernetes YAML) oft die bessere Wahl.

Exercises:
6 - Building and Running Multi-Container Applications (Pods)

7

Fehlerbehebung und Troubleshooting von Container-Problemen

Agenda des Themas:

Themenübersicht:

1. **Strukturierter Troubleshooting-Prozess:** Ein allgemeiner Plan zur Fehlersuche.
2. **Häufige Container-Probleme und Lösungsansätze:**
   - Container startet nicht.
   - Netzwerkprobleme.
   - Berechtigungsfehler (Volumes, Ports).
3. **Wichtige Diagnosewerkzeuge in Podman:**
   - Status, Logs, Konfigurationen prüfen.
   - Interaktive Fehlerdiagnose.
   - Ereignisse der Podman Engine überwachen (podman events).
4. **Spezifische Optionen für Berechtigungen:** :U, :z, :Z bei Volumes.

Ein strukturierter Plan für Troubleshooting

Wenn etwas nicht wie erwartet funktioniert, hilft ein strukturierter Prozess:

- **Problem beobachten & beschreiben:**
  - Was genau funktioniert nicht? (z.B. "Container startet nicht", "Webseite nicht erreichbar").
  - Was war das erwartete Verhalten?
  - Gibt es Fehlermeldungen?

- **Informationen sammeln:**
  - **Status:** Läuft der Container? Welchen Exit-Code hat er? (`podman ps -a`)
  - **Logs:** Was meldet die Anwendung im Container? (`podman logs <container>`)
  - **Konfiguration:** Ist der Container richtig konfiguriert? (`podman inspect <container/pod/network/volume>`)
  - **Ereignisse:** Was hat die Podman Engine zuletzt gemacht? (`podman events`)

- **Hypothesen bilden:** Was könnte die Ursache des Problems sein? (z.B. falscher Befehl im `Containerfile`, Port-Konflikt, fehlende Berechtigung).

- **Hypothesen testen:** Gezielte Änderungen vornehmen oder spezifische Diagnosetools einsetzen.

- **Lösung implementieren & verifizieren:** Problem beheben und erneut testen.

Typisches Problem: Container startet nicht

- **Symptome:** `podman ps -a` zeigt den Container mit Status "Exited" und einem Exit-Code ungleich Null (z.B. 1, 127). Der Container erscheint nicht in `podman ps` (da er nicht läuft).

- **Diagnose-Workflow:**
  - **Exit Code prüfen:** `podman ps -a` (Spalte STATUS oder EXIT CODE).
    - Ein Exit-Code > 0 signalisiert meist einen Fehler im Hauptprozess des Containers.
  - **Logs ansehen:** `podman logs <container_name_oder_id>`
    - Hier steht oft die direkte Fehlermeldung der Anwendung (z.B. "command not found", "file not found", "permission denied").
  - **Konfiguration prüfen:** `podman inspect <container_name_oder_id>`
    - Ist der Cmd oder Entrypoint korrekt? Sind Umgebungsvariablen richtig gesetzt?
  - **Image interaktiv testen:** `podman run -it --rm --entrypoint sh <image_name>`
    - Startet eine Shell im Image. Hier können Sie den fehlgeschlagenen Befehl manuell ausführen, um das Problem genauer zu untersuchen.
  - **Berechtigungen prüfen:** Hat der Benutzer, als der der Prozess im Container läuft (siehe USER im Containerfile oder `podman exec <container> id`), die nötigen Rechte für Dateien/Volumes?

Typisches Problem: Netzwerk Troubleshooting

Wenn Container nicht miteinander oder mit der Außenwelt sprechen:

- **Symptome:** "Connection refused", "Connection timed out", DNS-Fehler ("Could not resolve host"), Port-Mapping funktioniert nicht wie erwartet.

- **Diagnose-Workflow:**
  - **Port-Mapping prüfen:** `podman ps` (zeigt gemappte Ports), `podman inspect <container>` (Details zu NetworkSettings.Ports).
    - Lauscht der Prozess im Container wirklich auf dem erwarteten Port? `podman exec <container> ss -tulnp` (oder `netstat`).
  - **Host-Firewall prüfen:** Blockiert die Firewall des Host-Systems den Zugriff auf den gemappten Port? (z.B. `sudo firewall-cmd --list-all` oder `sudo ufw status`).
  - **Konnektivität testen:**
    - **Vom Host zum Container:** `curl http://localhost:<host_port>`
    - Innerhalb des Containers/Pods (zu sich selbst): podman exec <container> curl http://localhost:<container_port>
    - **Zwischen Containern (im selben Custom Network/Pod):** `podman exec <client_container> curl http://<server_container_name>:<server_port>`
  - **Netzwerkkonfiguration prüfen:** podman network ls, podman network inspect <network_name>. Sind die Container im richtigen Netzwerk?

Typisches Problem: Berechtigungsfehler

Wenn der Zugriff verweigert wird

- **Symptome:** "Permission denied" in den Logs, Container kann nicht auf gemountete Volumes schreiben, Fehler beim Starten von Diensten auf privilegierten Ports (<1024) im rootless Modus.

- **Diagnose-Workflow (Volumes):**
  - **Host-Berechtigungen:** Hat der Host-Benutzer, der podman run ausführt, Zugriff auf den Host-Pfad, der gemountet wird? (`ls -ld ./host-pfad`)
  - **UID/GID-Konflikt:** Der Benutzer im Container (z.B. UID 1000) hat andere Rechte als der Besitzer der Dateien auf dem Host (oft UID des Host-Users).
    - **Vergleichen:** podman exec <container> id vs. ls -ln ./host-pfad (zeigt numerische IDs).
    - **Lösung für Rootless & non-root Container User:** Die Mount-Option `:U` verwenden (z.B. `-v ./host-daten:/daten:U`). Dies passt die Besitzrechte des Mounts innerhalb des Containers an die UID des Container-Prozesses an, ohne die Host-Rechte zu ändern.
  - `SELinux-Probleme:` Auf Systemen wie Fedora/RHEL kann SELinux den Zugriff blockieren, auch wenn Dateisystemrechte passen.
    - **Prüfen:** `sudo ausearch -m avc -ts recent` (zeigt SELinux-Verweigerungen).
    - **Lösung:** Mount-Optionen `:z` (shared content) oder `:Z` (private content) verwenden (z.B. `-v ./host-daten:/daten:z`). ***Nur bei SELinux-Fehlern einsetzen!***

- **Diagnose-Workflow (Rootless Ports <1024):**
  - Rootless Container dürfen standardmäßig keine Ports unter 1024 auf dem Host binden.
  - **Lösung:** Einen höheren Host-Port auf den niedrigen Container-Port mappen (z.B. `-p 8080:80`) oder Systemeinstellungen anpassen (erfordert mehr Konfiguration).

Ereignisse in Echtzeit und rückblickend analysieren

- `podman events` zeigt Ihnen ein Protokoll der Aktionen und Ereignisse, die von der Podman Engine verarbeitet werden.

- **Nützlich für:**
  - Diagnose unerwarteter Container-Stopps (z.B. durch OOM-Killer – Out Of Memory).
  - Verfolgen von Image-Pulls, Container-Erstellungen, Starts, Stopps, Entfernungen etc.

- **Beispiele:**
  - Live-Stream aller Ereignisse: `podman events --stream`
  - Ereignisse der letzten Stunde filtern, die auf das "Sterben" eines Containers oder OOM hinweisen: `podman events --filter 'event=die' --filter 'event=oom' --since 1h`

Wichtige Troubleshooting-Befehle im Überblick

Ihre Werkzeugkiste für die Fehlersuche:

- `podman ps -a`: Status, Exit-Codes aller Container.
- `podman logs <container>`: Anwendungslogs aus dem Container.
- `podman inspect <resource>`: Detaillierte Konfigurationsinformationen (Container, Pod, Netzwerk, Volume, Image).
- `podman exec -it <container> sh (oder bash)`: Interaktive Shell im laufenden Container für direkte Untersuchungen.
- `podman stats [container]`: Live-Ressourcennutzung.
- `podman events`: Engine-Ereignisse (OOM, Start/Stop etc.).
- `podman network inspect <netzwerk>`: Details zur Netzwerkkonfiguration.
- `podman volume inspect <volume>`: Details zu Volumes.

Zusammenfassung

- Ein strukturierter Ansatz ist entscheidend für effizientes Troubleshooting.
- Die häufigsten Probleme betreffen **Container-Starts**, **Netzwerkkommunikation** und **Berechtigungen**.
- `podman logs`, `podman ps -a` und `podman inspect` sind Ihre wichtigsten ersten Anlaufstellen.
- Verstehen Sie die Unterschiede und Lösungen für Berechtigungsprobleme im Rootless-Modus (Mount-Option `:U`) und bei SELinux (Mount-Optionen `:z`/`:Z`).
- `podman events` hilft, tieferliegende Probleme der Engine oder unerwartete Stopps zu erkennen.

Exercises:
7 - Troubleshooting and Resolving Container Issues

8

Podman Compose verwenden

Agenda des Themas:

Themenübersicht:

1. **Die Herausforderung:** Komplexität von Multi-Container-Anwendungen.
2. **Lösungsansatz podman-compose:** Die Idee der deklarativen Anwendungsdefinition.
3. **Der Bauplan compose.yaml:** Struktur und Schlüsselkonzepte anhand eines Beispiels.
4. **Grundlegende Interaktion:** Wie man mit podman-compose Anwendungen zum Leben erweckt.
5. **Konfigurationsmanagement:** Die Rolle von .env-Dateien.
6. **Einordnung:** Wann und warum podman-compose?

Die Herausforderung: Komplexität managen

- Wir haben gesehen, dass moderne Anwendungen oft nicht nur aus einem, sondern aus **mehreren spezialisierten Containern** bestehen (z.B. Webserver, API-Backend, Datenbank).

- **Diese Container müssen:**
  - Korrekt konfiguriert werden.
  - Miteinander kommunizieren können.
  - Gemeinsam gestartet, gestoppt und verwaltet werden.
  - Ihre Daten persistent speichern.

- Das manuelle Starten und Verwalten jedes einzelnen Containers mit langen `podman run`-Befehlen wird schnell **komplex, fehleranfällig und schwer zu reproduzieren.**

- **Gesucht:** Eine Methode, um diese Anwendungslandschaft einfach, deklarativ und wiederholbar zu beschreiben und zu managen.

Lösungsansatz podman-compose

Die Idee: "Beschreibe deine Anwendung, nicht die Befehle"

- podman-compose ist ein Werkzeug, das hier ansetzt. Es erlaubt Ihnen, Ihre Multi-Container-Anwendung in einer einzigen Datei zu **beschreiben**, anstatt eine Serie von imperativen podman-Befehlen auszuführen.

- Es orientiert sich stark an der populären **Docker Compose-Syntax**.

- **Das Prinzip:**
  - Sie erstellen eine compose.yaml-Datei, die alle Ihre Dienste (Container), deren Images, Ports, Volumes, Netzwerke und Abhängigkeiten definiert.
  - podman-compose liest diese Datei und übersetzt Ihre Beschreibung in die notwendigen Podman-Aktionen (oftmals durch das Erstellen eines Pods, der Ihre Dienste gruppiert).

- **Vorteil:**
  - Deklarativ: Sie definieren den gewünschten Zustand.
  - Lesbarkeit & Wartbarkeit: Die gesamte Anwendungsdefinition an einem Ort.
  - Reproduzierbarkeit: Einfaches Aufsetzen der Anwendung auf verschiedenen Systemen.

- **Wichtig:** podman-compose ist ein separates, von der Community betriebenes Werkzeug, das Podman ergänzt.

Der Bauplan: `compose.yaml` – Ein Beispiel

```yaml
version: '3.8'  # Version der Compose-Spezifikation

services:       # Hier definieren wir unsere einzelnen Container-Dienste

  frontend_web:                # Name unseres ersten Dienstes (z.B. ein Webserver)
    image: nginx:alpine          # Verwendet ein fertiges Nginx-Image aus einer Registry
    container_name: mein_webserver # Ein spezifischer Name für den Container
    ports:
      - "8080:80"               # Mappt Port 8080 des Hosts auf Port 80 im Nginx-Container
    volumes:
      - ./meine_webseite:/usr/share/nginx/html:ro # Koppelt den lokalen Ordner 'meine_webseite'
                                                  # mit den Webinhalten des Nginx (nur lesend)
  api_service:                 # Name unseres zweiten Dienstes (z.B. ein Backend)
    build: ./mein_backend_code    # Weist Compose an, ein Image zu bauen:
                                  # Sucht ein 'Containerfile' im lokalen Ordner './mein_backend_code'
                                  # und baut daraus das Image für diesen Backend-Dienst.
    container_name: mein_api
    ports:                        # Optional: Mappt einen Port, falls das API direkt erreichbar sein soll
      - "5001:5000"               # Host-Port 5001 auf Container-Port 5000 des APIs
    environment:
      - API_KEY=geheim123
      - DATENBANK_HOST=datenbank_dienst # (Beispiel, wie es auf einen anderen Dienst verweisen könnte)

# (Optional könnten hier noch weitere Dienste wie eine Datenbank
#  oder benannte Volumes für persistente Daten deklariert werden)
#
#  datenbank_dienst:
#    image: postgres:latest
#    volumes:
#      - db_daten:/var/lib/postgresql/data
#
# volumes:
#   db_daten:
```

Schlüsselkonzepte hier:

- **services:** Die einzelnen Bausteine/Container Ihrer Anwendung (hier frontend_web und api_service).
- **image:** <name>:<tag>: Einen fertigen Bauplan (Image) aus einer Registry verwenden (wie bei nginx).
- **build:** <pfad/zum/ordner>: Einen eigenen Bauplan (Containerfile) im angegebenen lokalen Ordner verwenden, um ein neues Image zu erstellen (wie bei api_service).
- **ports:** Wie die Dienste von außen erreichbar gemacht werden.
- **volumes:** Wie lokale Ordner oder verwaltete Datenbereiche mit dem Container verbunden werden.
- **environment:** Wie Konfigurationswerte an die Dienste übergeben werden.

Grundlegende Interaktion mit podman-compose

- **Anwendung starten:**
  - `podman-compose up -d`
  - Dieser Befehl liest die `compose.yaml`, erstellt (falls nötig) Netzwerke, Volumes, Images (via build:) und Container und startet alles. Die Option `-d` lässt es im Hintergrund laufen.

- **Anwendung stoppen und entfernen:**
  - `podman-compose down [-v]`
  - **Beendet und löscht** die Container und das zugehörige (implizit erstellte) Netzwerk.
  - Mit der Option `-v` werden auch die in der compose.yaml deklarierten benannten Volumes gelöscht (Achtung: Datenverlust!).

- **Status der Dienste einsehen:**
  - `podman-compose ps`

- **Logs der Dienste anzeigen:**
  - `podman-compose logs` [dienstname] (optional mit `-f` für Live-Verfolgung).

Diese Befehle vereinfachen das Management erheblich im Vergleich zu einzelnen podman-Aufrufen.

Flexibilität durch Konfiguration mit .env-Dateien

Anpassungen ohne die compose.yaml zu ändern

- Für Werte, die sich zwischen Umgebungen (Entwicklung, Test, Produktion) ändern können oder nicht direkt in die compose.yaml gehören (z.B. spezifische Ports, Image-Tags), gibt es .env-Dateien.

- podman-compose lädt automatisch Variablen aus einer Datei namens `.env` im selben Verzeichnis wie die compose.yaml.

- **Prinzip:**
  - **In .env:** VARIABLE_NAME=wert
  - **In compose.yaml:** Verwendung mit ${VARIABLE_NAME}

- **Beispiel:**

```
# In compose.yaml
ports:
  - "${HOST_PORT_WEB}:80"
—----------------------------
# In .env
HOST_PORT_WEB=8081
```

- Dies erhöht die Flexibilität und hält die compose.yaml generischer.

- **Wichtig:** `.env-Dateien` sollten keine hochsensiblen Secrets enthalten, wenn die `.env-Datei` versioniert wird. Für echte Secrets sind Podman Secrets (Topic 5) oder andere spezialisierte Lösungen besser.

Einordnung: Wann und warum podman-compose?

- **Stärken:**
  - **Vertraute Syntax:** Ideal für Entwickler, die von Docker Compose kommen.
  - **Lokale Entwicklung:** Sehr gut geeignet, um komplexe Entwicklungsumgebungen schnell und einfach aufzusetzen und zu teilen.
  - **Vereinfachung:** Abstrahiert viele Podman-Details und ermöglicht die Steuerung einer ganzen Anwendungslandschaft mit wenigen Befehlen.
  - **Implizite Pod-Erstellung:** Erleichtert die Netzwerkkommunikation zwischen Diensten via localhost, da podman-compose die Dienste oft in einem gemeinsamen Pod startet.

- **Zu beachten:**
  - Es ist ein **Community-Tool**, dessen Feature-Set und Verhalten sich von Docker Compose und der reinen Podman-Nutzung unterscheiden können.
  - Für **Produktionsumgebungen oder sehr komplexe Szenarien**, die eng an Kubernetes-Konzepte angelehnt sind, bietet podman play kube (das wir im nächsten Topic behandeln) oft eine robustere und standardkonformere Lösung.

- **Fazit:** podman-compose ist ein nützliches Werkzeug im Werkzeugkasten, besonders für Entwicklung und wenn eine Compose-Datei bereits existiert oder bevorzugt wird.

Zusammenfassung

- podman-compose ermöglicht die **deklarative Definition von Multi-Container-Anwendungen** mittels einer compose.yaml-Datei, ähnlich der Docker Compose-Syntax.
- Es vereinfacht das Management durch Befehle wie `up` und `down` und nutzt Konzepte wie Services, Images, Ports und Volumes.
- .env-Dateien bieten eine flexible Möglichkeit zur Konfiguration.
- Ein wertvolles Werkzeug, insbesondere für die **lokale Entwicklung**, mit podman play kube als Alternative für Kubernetes-nahe Szenarien.

Exercises:
8 - Using Podman Compose

9

Podman Play Kube verwenden

Agenda des Themas:

Themenübersicht:

1. **Die Idee hinter `podman play kube`:** Warum ein deklarativer Ansatz?
2. **Der Bauplan: Kubernetes YAML für Podman**
   - Grundstruktur und wie Podman gängige Kubernetes-Objekte (Pod, PersistentVolumeClaim, ConfigMap, Secret) interpretiert.
3. **Der Workflow mit podman play kube:**
   - Anwendungen starten/erstellen (podman play kube <datei.yaml>).
   - Aufräumen (podman play kube --down <datei.yaml>).
4. **YAML generieren mit podman generate kube:** Ein Helfer mit Vorbehalten.
5. **Wichtige Aspekte:** Port-Mapping, Updates, Netzwerke in diesem Kontext.
6. **Einordnung:** Die Rolle von podman play kube in der Podman-Welt.

Die Idee: Was ist podman play kube?

Kubernetes-ähnliche Workflows für lokale Entwicklung

- Bisher haben wir Pods und Container oft **imperativ** erstellt (mit podman pod create, podman run).
- In größeren Umgebungen, insbesondere mit Kubernetes, ist ein **deklarativer Ansatz** Standard: Sie beschreiben den gewünschten Zustand Ihrer Anwendung in Konfigurationsdateien.
- `podman play kube` ermöglicht genau das für Podman:
  - Es liest Kubernetes YAML-Manifeste (Textdateien, die Ihre Anwendung beschreiben).
  - Es interpretiert diese Manifeste und erstellt die entsprechenden Podman-Ressourcen (Pods, Container, Volumes etc.) lokal auf Ihrem System.
- **Vorteile:**
  - **Deklarativ:** Sie definieren, was laufen soll, nicht wie es Schritt für Schritt erstellt wird.
  - **Kubernetes-Kompatibilität:** Nutzt die Standard-Syntax von Kubernetes. Das erleichtert das Verständnis und den späteren Übergang zu einem echten Kubernetes-Cluster.
  - **Reproduzierbar & Versionierbar:** YAML-Dateien können in Git verwaltet werden.
  - **Mächtiger als Compose (für K8s-Konstrukte):** Unterstützt mehr Kubernetes-spezifische Objekte.

Kubernetes YAML für Podman – Der Bauplan

```yaml
# Beispiel: meine-app.yaml
apiVersion: v1
kind: Pod               # Podman erstellt hieraus einen Podman Pod
metadata:
  name: mein-web-pod
spec:
  containers:
  - name: mein-nginx-container
    image: nginx:alpine
    ports:
    - containerPort: 80  # Port, den der Container im Pod bereitstellt
    volumeMounts:        # Wie Volumes im Container eingebunden werden
    - name: web-daten     # Interner Name des Volume-Mounts
      mountPath: /usr/share/nginx/html
  volumes:               # Definition der Volumes für diesen Pod
  - name: web-daten         # Muss zum 'volumeMounts.name' passen
    persistentVolumeClaim:
      claimName: mein-nginx-pvc # Verweist auf einen PersistentVolumeClaim
---
apiVersion: v1
kind: PersistentVolumeClaim # Podman interpretiert dies als Anforderung für ein Podman Volume
metadata:
  name: mein-nginx-pvc     # Der Name, auf den oben verwiesen wird
spec:
  # ... (Kubernetes-spezifische storageClass, accessModes etc. sind für
  #      Podman oft weniger relevant; der 'claimName' ist entscheidend)
  resources:
    requests:
      storage: 1Gi        # Größe oft informativ für Podman
```

Wie Podman die kind-Typen interpretiert:

- **kind:** Pod: Wird direkt zu einem Podman Pod. Enthält Container-Definitionen (image, ports, volumeMounts, env, resources, probes).
- **kind:** PersistentVolumeClaim (PVC):
  - Podman erwartet, dass ein Podman Volume mit dem Namen existiert, der im PVC unter metadata.name (oder genauer dem referenzierten claimName in der Pod-Spezifikation) angegeben ist. Z.B. podman volume create mein-nginx-pvc.
- **kind:** ConfigMap / kind: Secret:
  - Sie definieren diese in einer YAML-Datei (oder separaten Dateien).
  - podman play kube <ihre-configmap-oder-secret.yaml> erstellt diese Ressourcen innerhalb von Podmans Verständnis.
  - Ihr Pod-YAML kann dann via configMap.name oder secret.secretName in der volumes-Sektion darauf verweisen, um sie als Dateien in den Pod zu mounten.

Der Workflow mit podman play kube

Vom YAML-Manifest zum laufenden Pod

- **YAML-Manifest erstellen (oder generieren & stark anpassen):**
  - Sie schreiben eine .yaml-Datei (oder mehrere), die Ihre Pods und ggf. PersistentVolumeClaims, ConfigMaps, Secrets definiert.

- **Abhängige Podman-Ressourcen sicherstellen (falls nötig):**
  - Wenn Ihre YAML einen PersistentVolumeClaim referenziert (z.B. claimName: mein-daten-volumen), stellen Sie sicher, dass ein Podman Volume dieses Namens existiert: podman volume create mein-daten-volumen.

- **Anwendung starten/erstellen:**
  - `podman play kube meine-app.yaml`
  - Podman liest die Datei(en) und erstellt die entsprechenden lokalen Ressourcen.
  - Port-Mapping: Für den Zugriff von außen erfolgt dies am besten explizit mit der Option --publish HOST_PORT:CONTAINER_PORT beim podman play kube-Aufruf.

- **Anwendung aufräumen:**
  - `podman play kube --down meine-app.yaml`
  - Stoppt und entfernt die Pods und Container, die durch diese YAML-Datei erstellt wurden.
  - Wichtig: Podman-verwaltete Volumes bleiben standardmäßig erhalten!

YAML generieren mit podman generate kube – Ein Helfer

- Wenn Sie bereits laufende Pods oder Container mit Podman haben, können Sie podman generate kube <pod_oder_container_name> verwenden, um eine Kubernetes YAML-Repräsentation davon zu erstellen.

- **Das Ergebnis ist ein Roh-Entwurf und muss fast immer stark überarbeitet werden!**

- **Typische Anpassungen nach der Generierung:**
  - **Entfernen von Laufzeit-Details:** Zeitstempel, interne IDs, etc.
  - **Namen anpassen:** Für Pods, Container, Volumes/PVCs.
  - **persistentVolumeClaim.claimName korrigieren:** Sicherstellen, dass es auf das gewünschte Podman Volume zeigt.
  - **hostPort entfernen:** Besser --publish bei play kube verwenden.
  - Ggf. Ressourcenlimits, Health Probes hinzufügen, die nicht automatisch generiert wurden.

- **Sinn:** generate kube kann als Ausgangspunkt dienen, um die Grundstruktur einer YAML zu erhalten, die man dann für eine saubere, deklarative Definition verfeinert.

Wichtige Aspekte und Optionen für podman play kube

Feinsteuerung und typische Anwendungsfälle

- **Port-Mapping:**
  - Der containerPort (Port im Pod) wird in der YAML definiert.
  - Das Mapping zum Host-Port (Zugriff von außen) idealerweise mit `--publish HOST:CONTAINER` beim podman play kube-Aufruf.

- **Updates von Deployments:**
  - Ändern Sie Ihre YAML-Datei.
  - Verwenden Sie `podman play kube --replace meine-app.yaml`. Dies entfernt den alten Pod und erstellt ihn neu basierend auf der aktualisierten YAML.

- **Netzwerke:**
  - Mit `--network <podman-netzwerkname>` können Sie den durch die YAML erstellten Pod explizit einem bestehenden Podman-Netzwerk zuweisen.

- **Reihenfolge bei Abhängigkeiten:**
  - Wenn ein Pod ein ConfigMap oder Secret benötigt: `podman play kube` zuerst für die ConfigMap/Secret-YAML ausführen, dann für die Pod-YAML.

- **Ressourcenlimits & Health Probes in YAML:**
  - Sie können resources.limits (CPU/Memory) und livenessProbe/readinessProbe direkt in der Pod-Spezifikation definieren. Podman versucht, diese umzusetzen (Effektivität der Limits hängt von cgroups ab).

Einordnung: podman play kube in der Podman-Welt

Die Rolle des deklarativen Ansatzes

- **Stärken:**
  - **Deklarativ:** Sie beschreiben den gewünschten Zustand, was zu reproduzierbaren und versionierbaren Setups führt.
  - **Kubernetes-Syntax:** Nutzt einen etablierten Standard, was das Verständnis und die Portabilität von Konzepten fördert.
  - **Mächtiger als podman-compose für K8s-Konstrukte:** Bessere Unterstützung für Pods, PVCs, ConfigMaps, Secrets im Kubernetes-Stil.
  - **Ideal für lokale Entwicklung/Tests von Kubernetes-nahen Anwendungen.**

- **Abgrenzung:**
  - `podman play kube` ist **kein vollständiger Kubernetes-Cluster**. Es fehlen komplexe K8s-Controller (z.B. für Rolling Updates, Skalierung) und fortgeschrittene Netzwerk-Services. Es führt primär die Pod-Definitionen aus.

- **Empfehlung:**
  - Für Podman ist `podman play kube` der **bevorzugte deklarative Ansatz**, wenn es um die Verwaltung von Pods mit mehreren Ressourcen geht oder wenn eine Nähe zu Kubernetes-Workflows gewünscht ist.

Zusammenfassung

- podman play kube ermöglicht die **deklarative Verwaltung von Pods** und verwandten Ressourcen (Volumes, ConfigMaps, Secrets) unter Verwendung von **Kubernetes YAML-Manifesten.**
- Es interpretiert Kubernetes-Objekte wie Pod, PersistentVolumeClaim (wird auf Podman Volume gemappt), ConfigMap und Secret.
- podman generate kube kann als Ausgangspunkt dienen, erfordert aber **stets eine signifikante Bereinigung** der generierten YAML.
- Wichtige Optionen wie --publish, --down, --replace, --network steuern das Verhalten.
- Ein mächtiges Werkzeug für die lokale Entwicklung, das einen standardisierten, deklarativen Weg zur Definition von Podman-Anwendungen bietet.

Exercises:

9 - Using podman play kube

10 Podman Desktop

Agenda des Themas:

Themenübersicht:

1. **Was ist Podman Desktop?** Die Idee und Zielgruppe.
2. **Podman Machine:** Das Fundament verstehen.
3. **Kernfunktionen im Überblick:** Was kann Podman Desktop?
4. **Warum Podman Desktop nutzen?** Die Vorteile.
5. **Erste Schritte:** Installation und Start (Kurzüberblick).
6. **Einordnung:** GUI als Ergänzung zur CLI.

Was ist Podman Desktop?

- Podman Desktop ist eine **alternative, grafische Benutzeroberfläche (GUI)**. Sie ermöglicht es Ihnen, Ihre Container, Images, Pods und die gesamte Container-Umgebung visuell zu verwalten und zu inspizieren.

- **Gedacht für:**
  - Nutzer, die eine **visuelle Interaktion** bevorzugen oder benötigen.
  - **Einsteiger**, denen eine grafische Darstellung hilft, die Konzepte besser zu verstehen.
  - **Entwickler**, die eine schnelle Übersicht und einfache Aktionen für ihre lokale Umgebung wünschen.

- Podman Desktop arbeitet mit Ihrer bereits installierten Podman Engine zusammen und macht viele CLI-Funktionen grafisch zugänglich.

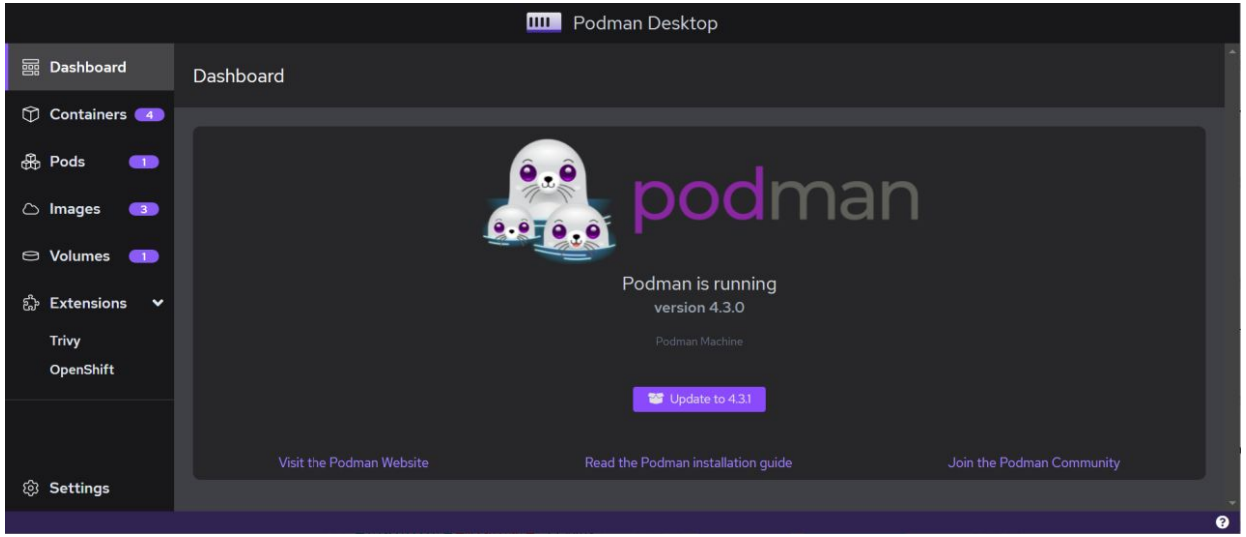
Podman Machine (Wichtig für Windows/macOS)

- Container-Technologie ist **nativ unter Linux** zu Hause.
- Um Podman auf **Windows oder macOS** nutzen zu können, wird eine Linux-Umgebung benötigt.
- **Podman Desktop verwaltet hierfür oft eine virtuelle Maschine (VM)**, die als **Podman Machine** bezeichnet wird.
  - Diese VM (z.B. basierend auf WSL2 unter Windows oder QEMU/Lima unter macOS) hostet die eigentliche Podman Engine.
  - Podman Desktop kümmert sich um die Einrichtung, das Starten/Stoppen und die Kommunikation mit dieser VM.
- **Unter Linux:** Läuft die Podman Engine in der Regel direkt auf Ihrem Host, eine Podman Machine ist dann meist nicht erforderlich.
- In den Einstellungen von Podman Desktop können Sie den Status dieser Podman Machine einsehen.

Kernfunktionen von Podman Desktop im Überblick

Podman Desktop bietet eine grafische Schnittstelle für viele gängige Podman-Aufgaben:

- **Dashboard:** Zentrale Übersicht (laufende Container, Pods, Images, Volumes).
- **Container Management:** Auflisten, Starten, Stoppen, Details inspizieren, Logs ansehen, Terminal öffnen.
- **Pod Management:** Erstellen (auch aus bestehenden Containern), Auflisten, Starten, Stoppen, Inspizieren.
- **Image Management:** Auflisten, Pullen, eigene Images aus Containerfiles bauen, Taggen, Pushen, Löschen.
- **Volume Management:** Auflisten, Erstellen, Löschen.
- **Kubernetes-Integration:**
  - Kubernetes YAML-Dateien anwenden (podman play kube).
  - YAML aus bestehenden Pods generieren (podman generate kube).
- **Erweiterungen (Extensions):** Möglichkeit, zusätzliche Funktionen durch Plugins zu integrieren.



Warum Podman Desktop nutzen?

Die Vorteile einer grafischen Oberfläche

- **Benutzerfreundlichkeit:**
  - Einfacherer, visueller Einstieg, besonders ohne tiefe CLI-Kenntnisse.
  - Viele Aktionen sind intuitiv per Klick erreichbar.
- **Visuelle Einsicht & Verständnis:**
  - Grafische Darstellung von Ressourcen und deren Status macht Zusammenhänge oft klarer.
- **Entdeckbarkeit von Funktionen:**
  - Man sieht direkt, welche Optionen verfügbar sind, ohne alle CLI-Befehle und Flags kennen zu müssen.
- **Beschleunigung von Routineaufgaben:**
  - Für schnelle Überprüfungen oder Standardaktionen kann die GUI zeitsparend sein.
- **Brücke zu Kubernetes-Konzepten:**
  - Erleichtert das lokale Arbeiten mit Kubernetes-Manifesten und das Verständnis von Pod-Strukturen.

Inbetriebnahme und Verbindung (Konzeptionell)

Wie Podman Desktop mit Ihrer Engine interagiert

- **Podman Desktop starten:**
  - Nach dem Start der Anwendung wird diese versuchen, eine Verbindung zu einer laufenden Podman Engine aufzubauen.

- **Automatische Verbindung (Idealfall):**
  - Podman Desktop erkennt Ihre lokale Podman Installation (oder die laufende Podman Machine unter Windows/macOS) und verbindet sich automatisch.

- **Status und Engine-Details prüfen:**
  - Im Dashboard oder in den Einstellungen (oft unter "Resources" oder "Providers") können Sie sehen, mit welcher Engine Podman Desktop verbunden ist und ob die Verbindung aktiv ist.
  - Hier können Sie ggf. auch die Podman Machine (falls verwendet) starten oder stoppen.

- **Interaktion:** Sobald verbunden, können Sie über die GUI mit Ihren Containern, Images etc. interagieren. Podman Desktop sendet im Hintergrund die entsprechenden Befehle an die Podman Engine.

Podman Desktop – Ergänzung, nicht Ersatz

Das Beste aus beiden Welten nutzen

- Podman Desktop ist ein **nützliches Werkzeug zur Visualisierung und Vereinfachung** vieler Aufgaben im Container-Management.

- Es ist jedoch optimal als **Ergänzung zur mächtigen Kommandozeile (CLI)** zu sehen, nicht unbedingt als vollständiger Ersatz.

- **Stärken der CLI bleiben bestehen:**
  - Automatisierung durch Skripting.
  - Präzise Kontrolle über alle verfügbaren Optionen und Flags.
  - Tiefergehende Diagnosemöglichkeiten bei komplexen Problemen.
  - Effizienz für erfahrene Nutzer, die Befehle schnell tippen können.

- **Empfohlene Herangehensweise:**
  - Nutzen Sie Podman Desktop für einen schnellen Überblick, einfache Aktionen und um Konzepte visuell zu erfassen.
  - Verwenden Sie die CLI für komplexe Setups, Automatisierung und wenn Sie die volle Kontrolle benötigen.

Zusammenfassung

- **Podman Desktop** bietet eine grafische Oberfläche (GUI) zur komfortablen Verwaltung Ihrer Podman-Umgebung.
- Es vereinfacht viele gängige Aufgaben und kann besonders für Einsteiger oder für einen schnellen visuellen Überblick nützlich sein.
- Unter Windows und macOS ist oft die **Podman Machine** (eine Linux-VM) involviert, die von Podman Desktop verwaltet wird.
- Kernfunktionen umfassen das Management von Containern, Images, Pods sowie die Integration mit Kubernetes YAML.
- Eine wertvolle **Ergänzung zur Kommandozeile**, die verschiedene Arbeitsweisen unterstützt.

Exercises:
10 - Podman Desktop

11

Buildah und Skopeo

Vorstellung Buildah: Agenda

1. Einführung: Was ist Buildah und warum sollte ich das Tool verwenden?
2. Kernfunktionen von Buildah: Was Buildah alles kann im Detail
3. Buildah vs. Podman Build: Die entscheidenden Vorteile
4. Buildah im Ökosystem: Zusammenspiel mit Podman & Co.
5. Erste Schritte & Praktische Übungen (Hands-on)

Was ist Buildah?

buildah

Die Kernidee: Flexible und daemon-lose Container-Image-Erstellung.

- Ein einfaches Tool zum Erstellen von Container-Images – ohne Docker und ohne laufende Hintergrunddienste.
- **Buildah** ist ein Open-Source-Programm für Linux, mit dem man sogenannte Container-Images bauen kann. Diese Images sind die Grundlage für Container – also kleine, abgeschlossene Softwarepakete, in denen Anwendungen mit allem laufen, was sie brauchen.
- Im Gegensatz zu Docker braucht Buildah **keinen laufenden Hintergrunddienst** (Daemon). Das macht es sicherer, schneller und flexibler.
- **Buildah** ist speziell darauf ausgelegt, **OCI-kompatible (Open Container Initiative) Container-Images** zu erstellen, **die auch mit Docker und Kubernetes kompatibel sind**

Was kann Buildah?

buildah

- **Container-Images erstellen – ganz flexibel:** Sie können Buildah nutzen, um eigene Images zu bauen – entweder aus einem vorhandenen Basis-Image oder komplett von Null.
- **Mit oder ohne Dockerfile:** Ein Dockerfile ist eine Art Bauanleitung für ein Container-Image. Buildah kann solche Dateien verwenden, aber auch ohne sie arbeiten – zum Beispiel mit einfachen Befehlen in der Kommandozeile.
- **Leichte und sichere Images:** Die erstellten Images enthalten nur das Nötigste – keine zusätzlichen Werkzeuge, die im Image zurückbleiben. Das spart Platz und erhöht die Sicherheit.
- **Kompatibel mit Docker:** Wenn Sie bisher Docker verwendet haben, können Sie Buildah leicht in Ihren bisherigen Workflow einbinden.
- **Keine Spezialsoftware nötig:** Buildah arbeitet direkt auf dem Linux-System und kann mit anderen Tools kombiniert werden, die Sie schon kennen.

Kernfunktionen und Vorteile von Buildah

- **Daemon-los und sicher:** Im Gegensatz zu Docker benötigt Buildah keinen dauerhaft laufenden Daemon. Dadurch erhöht sich die Sicherheit, da keine Root-Rechte für den Betrieb erforderlich sind.
- **Flexible Image-Erstellung:** Buildah kann Container-Images sowohl aus bestehenden Basis-Images als auch komplett von Grund auf („from scratch“) erstellen. Dabei können Sie Ihre bevorzugten Tools und Skriptsprachen nutzen, was eine hohe Anpassungsfähigkeit bietet.
- **Mit oder ohne Dockerfile:** Sie können Images klassisch mit einem Dockerfile (bzw. Containerfile) bauen, aber auch ohne – Schritt für Schritt per Kommandozeile oder Skript. Dies erlaubt individuelle Build-Prozesse und die Integration anderer Skriptsprachen.
- **Schlanke Images:** Da Buildah keine Build-Tools in das Image selbst integriert, werden die Images kleiner, sicherer und benötigen weniger Ressourcen beim Transport.
- **Kompatibilität:** Buildah unterstützt Dockerfiles vollständig und erleichtert so den Umstieg von Docker. Die erzeugten Images können direkt in Container-Registries gespeichert und mit anderen Tools wie Podman oder Docker genutzt werden.
- **Nutzerspezifische Images:** Buildah ermöglicht es, Images nach dem Nutzer zu sortieren, der sie erstellt hat. Das erleichtert die Verwaltung in Multi-User-Umgebungen.
- **Integration in das Container-Ökosystem:** Buildah ist Teil eines modularen Toolsets (u.a. mit Podman und Skopeo) und kann mit bestehenden Container-Host-Tools kombiniert werden.

Warum sollte ich Buildah verwenden?

Buildah ist besonders dann hilfreich, wenn ihr:

- einfache und sichere Images bauen möchtet,
- mehr Kontrolle über den Build-Prozess wünscht,
- oder keinen Docker-Daemon (z. B. aus Sicherheitsgründen) einsetzen könnt oder wollt.

Buildah ist das ideale Tool, wenn ihr **schlanke, sichere und hochgradig anpassbare Container-Images** ohne den Overhead eines Daemons erstellen möchtet.

Zusätzliche Möglichkeiten und Unterschiede zu Podman

- **Zusätzliche Möglichkeiten von Buildah:**
  - Images prüfen, verifizieren und ändern
  - Container und Images zwischen lokalen Systemen und Registries verschieben
  - Lokale Images löschen oder das Root-Dateisystem eines Containers mounten und als neue Image-Schicht verwenden

- **Unterschied zu Podman:**
  - Während **Buildah** auf die **Erstellung von Images** spezialisiert ist, dient **Podman** vor allem dem **Verwalten und Ausführen** von Containern.
  - **Buildah** eignet sich für Nutzer, die **maximale Flexibilität, Kontrolle und Sicherheit** beim **Image-Bau** benötigen.
  - Beide Tools sind jedoch eng miteinander verzahnt und teilen sich die zugrunde liegende Technologie für Images.
  - Podman nutzt für den Build-Prozess intern Buildah. Die Container von Buildah und Podman sind voneinander getrennt, aber beide Tools greifen auf denselben Image Speicher zu.

Übersicht: Buildah vs. Podman.

Aspekt	Buildah	Podman
Hauptaufgabe	Erstellung und Modifikation von Images	Verwaltung und Ausführung von Containern
Image-Build	Feingranulare Steuerung, auch ohne Dockerfile, Skripting möglich	Nutzt Buildah-Code für podman build, weniger Kontrolle
Container-Konzept	Temporäre Container zum Image-Bau	Langfristige, "traditionelle" Container für den Betrieb
CLI-Kompatibilität	Eigene Kommandos, z.B. buildah from, buildah run	Docker-kompatible CLI, kann als Drop-in-Ersatz dienen
Sichtbarkeit	Container nur in Buildah sichtbar	Container nur in Podman sichtbar
Image Store	Gemeinsamer Image Speicher mit Podman	Gemeinsamer Image Speicher mit Buildah

Buildah im Ökosystem: Zusammenspiel mit Podman & Co.

- Buildah, Podman und Skopeo sind eng miteinander verzahnte Open-Source-Tools, die gemeinsam ein flexibles und sicheres Container-Ökosystem bilden. Jedes Werkzeug übernimmt dabei eine klar definierte Rolle:

Rollenverteilung und Zusammenspiel

- **Buildah:** Spezialisiert auf das Erstellen von OCI- und Docker-kompatiblen Container-Images. Buildah ermöglicht sowohl das klassische Bauen mit Dockerfile/Containerfile als auch feingranulare Builds ohne diese Dateien. Die erstellten Images sind sofort für andere Tools nutzbar und werden im gemeinsamen lokalen Imagespeicher abgelegt.
- **Podman:** Übernimmt das Management, die Ausführung und Wartung von Containern und Images. Podman kann Container aus beliebigen Images starten, steuern und verwalten. Für den Befehl podman build nutzt Podman intern die Buildah-Bibliothek, sodass beide Tools eng zusammenarbeiten und auf denselben Imagespeicher zugreifen.
- **Skopeo:** Ergänzt das Ökosystem um Funktionen zur Inspektion, zum Kopieren und Synchronisieren von Container-Images zwischen lokalen Speichern und entfernten Registries – ohne die Images lokal ausführen oder entpacken zu müssen

Exercises:
11.1 - Buildah

Erstellen eines Images mit Buildah (ohne Dockerfile, mit Ubuntu).

Ziel: Verstehen, wie Buildah Images ohne Containerfile erzeugt.

```bash
# Basis-Container erstellen
buildah from ubuntu:22.04

# Container-Namen zwischenspeichern
ctr=$(buildah from ubuntu:22.04)

# curl installieren
buildah run $ctr apt-get update
buildah run $ctr apt-get install -y curl

# Metadaten setzen
buildah config --author "Dein Name" --label version=1.0 $ctr

# Image erstellen
buildah commit $ctr ubuntu-curl:1.0

# Testen
podman run ubuntu-curl:1.0 curl --version
```

Multistage-Build mit Buildah (Go-App auf Ubuntu).

Schritte:

- Stage 1: Kompilierung einer Go-Anwendung mit `golang:1.20`
- Stage 2: Nur Binärdatei in Ubuntu-Image übernehmen
- Ergebnis: Kleines, produktionsreifes Image

Ziel: Erstellung eines optimierten Images mit Go-Build und Ubuntu-Stage.
Containerfile (im gleichen Verzeichnis wie main.go):

Dockerfile

```
FROM golang:1.20 as builder
WORKDIR /app
COPY main.go .
RUN go build -o myapp main.go

FROM ubuntu:22.04
RUN apt-get update && apt-get install -y ca-certificates && apt-get clean
COPY --from=builder /app/myapp /usr/local/bin/myapp
CMD ["myapp"]
```

Build und Test:

bash

```
buildah bud -t go-app-ubuntu:latest -f Containerfile .
podman run go-app-ubuntu:latest
```

Vermittelt effiziente Image-Erstellung und praktiziert Trennung von Build- und Laufzeitumgebung

Benutzerdefiniertes Basis Image mit Ubuntu-Minimum.

Schritte:

- buildah from scratch
- Manuelles Hinzufügen von Binaries und Configs mit buildah copy
- Einstiegspunkt mit buildah config setzen
- Finales Commit mit buildah commit

Ziel: Minimales, sicheres Image selbst aufbauen.
Containerfile (im gleichen Verzeichnis wie main.go):

bash

```
# Neuen Container von Ubuntu starten
ctr=$(buildah from ubuntu:22.04)

# Nur benötigte Tools installieren (z. B. ping)
buildah run $ctr apt-get update
buildah run $ctr apt-get install -y iputils-ping

# Einstiegspunkt setzen
buildah config --entrypoint '["ping", "8.8.8.8"]' $ctr

# Commit
buildah commit $ctr ping-ubuntu:latest

# Testen
podman run ping-ubuntu:latest
```

Multi-Architektur-Image mit Ubuntu.

Schritte:

- Bau einzelner Architekturen mit --arch
- Zusammenführung via buildah manifest create/add
- Push zu Registry mit Manifest-Support

Ziel: Bereitstellung eines Images für mehrere Architekturen (z. B. amd64, arm64).
Voraussetzung: Du brauchst eine Umgebung mit QEMU-Emulation, z. B. über binfmt-support.)

```
bash
    # Build Images für amd64 und arm64
    buildah bud --arch amd64 -t ubuntu-app:amd64 -f Containerfile .
    buildah bud --arch arm64 -t ubuntu-app:arm64 -f Containerfile .

    # Manifest erstellen und Images hinzufügen
    buildah manifest create ubuntu-app:multi
    buildah manifest add ubuntu-app:multi containers-storage:ubuntu-app:amd64
    buildah manifest add ubuntu-app:multi containers-storage:ubuntu-app:arm64

    # Optional: Push zu einer Registry
    # buildah manifest push ubuntu-app:multi docker://your-registry/ubuntu-app:multi
```

Nutzen: Sehr relevant für Cloud-native Deployments, Edge-Devices oder GitOps-Workflows.

Hinweis: ZIP-Datei mit angepassten Dateien für die vier Buildah-Übungen mit Ubuntu als Basis-Image:

ubuntu-buildah-übungen.zip

Enthalten in der ZIP -Datei sind:

Datei	Zweck
1-erstellen-ohne-dockerfile.sh	Ubuntu-Image mit curl ohne Containerfile
2-multistage-build-containerfile	Multistage-Build (Go → Ubuntu)
2-multistage-build.sh	Bauskript zum Containerfile
3-basisimage-ubuntu.sh	Minimal-Image mit ping als Einstiegspunkt
4-multiarch-containerfile	Einfaches Image zur Multiarch-Demo
4-multiarch-manifest.sh	Manifest-Erstellung für amd64 + arm64

Nach dem Entpacken ggf. Ausführbarkeit setzen: chmod +x *.sh

HINWEIS: Red Hat Buildah Cheat Sheet, Red Hat Podman Cheat Sheet

Vorstellung Skopeo: Agenda

1. Skopeo: Das Schweizer Taschenmesser für Container-Images
2. Kernfunktionen im Detail: Skopeo's vielseitige Fähigkeiten
3. Die unschlagbaren Vorteile von Skopeo
4. Skopeo im Podman/Buildah-Ökosystem
5. Erste Schritte & Praktische Übungen (Hands-on)

Skopeo: Das Schweizer Taschenmesser für Container-Images

Was ist Skopeo?

- **Skopeo** ist ein vielseitiges Open-Source-CLI-Tool, das speziell für das **Management von Container-Images und Image-Repositories** entwickelt wurde.
- Es ermöglicht das **Inspektieren, Kopieren, Signieren und Übertragen** von Container-Images zwischen unterschiedlichen Speichern und Registries – und das ohne laufenden Daemon oder Root-Rechte.
- Skopeo arbeitet plattformübergreifend auf Linux, Windows und macOS und unterstützt sowohl OCI- als auch Docker-Image-Formate

Skopeo ist das „Schweizer Taschenmesser“ für Container-Images: leichtgewichtig, flexibel und spezialisiert auf das Inspektieren, Kopieren, Synchronisieren und Signieren von Images – lokal wie remote, ohne Root und ohne Daemon. Damit ist es ein unverzichtbares Werkzeug für moderne, sichere und effiziente Container-Workflows

Skopeo: Das Schweizer Taschenmesser für Container-Images

skopeo

Ein kompakter Überblick über Skopeo:

- **Inspektion ohne Download:** Mit Skopeo können Sie Metadaten und Eigenschaften von Images direkt in einer Registry anzeigen lassen, ohne das komplette Image herunterladen zu müssen. Das spart Zeit und Speicherplatz, etwa bei der Prüfung von Image-Tags, Labels oder Layer-Strukturen.
- **Kopieren zwischen Registries:** Skopeo erlaubt das direkte Kopieren von Images zwischen verschiedenen Registries oder Speichertypen – ohne Zwischenspeicherung auf dem lokalen System. So lassen sich Images effizient zwischen Docker Hub, Quay, internen Registries oder lokalen Verzeichnissen verschieben.
- **Synchronisation und Air-Gapped Deployments:** Für abgeschottete (air-gapped) Umgebungen kann Skopeo komplette Repositories synchronisieren. Das ist besonders nützlich, um externe Images in interne, abgesicherte Registries zu übertragen.
- **Signieren und Verifizieren:** Images können mit Skopeo signiert und deren Authentizität überprüft werden – ein wichtiger Beitrag zur Supply-Chain-Sicherheit.
- **Löschen von Images:** Skopeo kann Images auch aus Remote-Repositories entfernen

Die unschlagbaren Vorteile von Skopeo. Essenzielle Anwendungsfälle und Mehrwert.

- **Effiziente Inspektion und Verwaltung:** Ihr könnt **Images in entfernten Registries inspizieren, ohne sie zu ziehen**. Das erleichtert die Analyse und Qualitätssicherung, vor allem bei großen Images oder limitierten Speicher.
- **Flexibles Kopieren und Synchronisieren:** Skopeo ermöglicht das **Kopieren von Images** zwischen beliebigen Registries, lokalen Verzeichnissen oder OCI-Layouts – **ideal für Multi-Cloud- oder Hybrid-Umgebungen und CI/CD-Pipelines**. Besonders in **air-gapped Szenarien** ist Skopeo ein unverzichtbares Werkzeug, um Images sicher und automatisiert in interne Netze zu bringe.
- **Sicherheit und Compliance:** Durch das **Signieren und Verifizieren von Images** wird die Integrität und Authentizität der Images sichergestellt.
- **Rootlos und Daemonlos:** Skopeo **benötigt keine erhöhten Rechte und läuft ohne Hintergrunddienst**, was die Angriffsfläche verringert und die Sicherheit erhöht.
- **Modularität und Integration:** Skopeo ist Teil eines modularen Toolsets (mit Podman und Buildah), das gemeinsam ein flexibles, sicheres und modernes Container-Ökosystem bildet. Jedes Tool übernimmt eine klar definierte Aufgabe und kann unabhängig oder in Kombination genutzt werden.
- **Automatisierung und CI/CD:** Skopeo ist ideal für automatisierte Workflows in CI/CD-Systemen, etwa um Images zwischen Build- und Produktionsumgebungen zu verschieben oder regelmäßig interne Registries zu synchronisieren

Skopeo im Podman/Buildah-Ökosystem.

- Skopeo ist ein zentrales Werkzeug im modernen Container-Ökosystem, das eng mit Podman und Buildah zusammenarbeitet. Während Buildah für das Erstellen und Podman für das Ausführen und Verwalten von Containern zuständig ist, übernimmt Skopeo alle Aufgaben rund um das Übertragen, Inspektieren und Synchronisieren von Container-Images zwischen unterschiedlichen Speichern und Registries.

Zusammenspiel in typischen Workflows:

1. **Image bauen:** Mit Buildah wird ein neues Container-Image erstellt und im lokalen Speicher abgelegt.
2. **Image prüfen:** Skopeo kann das Image inspizieren – sowohl lokal als auch remote, ohne es komplett herunterzuladen.
3. **Image übertragen:** Skopeo kopiert das Image direkt in eine Registry, auf einen anderen Host oder in ein anderes Format (z.B. für Air-Gapped-Umgebungen).
4. **Container starten:** Podman nutzt das Image, um Container zu starten und zu verwalten.
5. **Synchronisation:** Skopeo kann komplette Repositories synchronisieren, etwa zur Spiegelung von Images in interne Registries für abgesicherte Netzwerke.

Skopeo im Podman/Buildah-Ökosystem.

Vorteile des gemeinsamen Einsatzes

- **Gemeinsamer Image Speicher:** Buildah, Podman und Skopeo greifen auf **denselben lokalen Imagespeicher** zu, was nahtlose Übergänge zwischen Build, Verwaltung und Transfer ermöglicht.
- **Keine Daemons, rootlos:** Alle Tools arbeiten **ohne Hintergrunddienste** und **unterstützen den rootlosen Betrieb** – das erhöht Sicherheit und Flexibilität.
- **Formatvielfalt:** Skopeo unterstützt **zahlreiche Speicher- und Registry-Formate** (Docker, OCI, Archive, Directory, etc.) und kann Images zwischen diesen Formaten konvertieren.
- **Automatisierung:** Die Tools lassen sich **hervorragend in CI/CD-Pipelines und automatisierte Deployments integrieren**, z.B. für Air-Gapped-Deployments oder Multi-Registry-Workflows.

Skopeo ergänzt Buildah und Podman als „Transport- und Inspektionswerkzeug“ für Container-Images. Im Zusammenspiel ermöglichen die drei Tools einen vollständigen, sicheren und flexiblen Container-Lifecycle – vom Bau über das Management bis hin zum Transfer und zur Synchronisation von Images, sowohl lokal als auch über verschiedene Registries hinweg.

Exercises:
11.2 - Skopeo

Remote-Image inspizieren ohne Download.

Ziel: Ein öffentliches Container-Image analysieren, ohne es lokal zu speichern.

```bash
skopeo inspect docker://docker.io/library/ubuntu:22.04
```

Du erhältst Details wie:

- Digest (SHA)
- Architektur
- Labels
- Layerstruktur

Nutzen: Sehr schneller Einstieg zur Image-Verifikation vor dem Einsatz.

Image von Docker Hub in eine Private Registry kopieren.

Ziel: Ein ubuntu:22.04-Image in ein internes Repository übertragen.

```
bash
    skopeo copy \
      docker://docker.io/library/ubuntu:22.04 \
      docker://myregistry.example.com/myproject/ubuntu:22.04
```

Erfordert ggf. Authentifizierung mit --src-creds / --dest-creds.

Nutzen: Ideal für Offline-Cluster, CI/CD oder abgesicherte Produktionspipelines.

Ubuntu-Image signieren und signierte Quelle verifizieren.

Ziel: Supply Chain Security – sicherstellen, dass ein Image authentisch ist.

```
bash
        # Image signieren (GPG-Key muss konfiguriert sein)
        skopeo copy --sign-by my-gpg-id \
          docker://docker.io/library/ubuntu:22.04 \
          dir:/tmp/signed-images

        # Verifikation (nur mit unterstütztem Setup)
        skopeo inspect --verify-signature docker://docker.io/library/ubuntu:22.04
```

Nutzen: Schutz gegen manipulierte Images, z. B. in regulierten Branchen.

Repository-Mirroring: Ubuntu-Images lokal synchronisieren

Ziel: Ein ganzes Ubuntu-Image-Repository lokal spiegeln (z. B. für Air-Gapped-Umgebungen).

```
bash
        skopeo sync \
          --src docker \
          --dest dir \
          docker.io/library/ubuntu \
          /opt/repo-mirror/ubuntu
```

Nutzen: Auch für geplante Snapshots oder lokale CI/CD-Systeme nutzbar.

Hinweis: ZIP-Datei mit angepassten Dateien für die vier Skopeo-Übungen mit Ubuntu als Basis-Image:

ubuntu-skopeo-exercises

Enthalten in der ZIP -Datei sind:

1. 1-inspect-ubuntu-image.sh – Metadaten von ubuntu:22.04 ohne Download anzeigen
2. 2-copy-ubuntu-to-private-registry.sh – Image in eine private Registry kopieren
3. 3-sign-and-verify-ubuntu.sh – Image signieren & verifizieren (mit GPG)
4. 4-sync-ubuntu-repo.sh – Ubuntu-Repository lokal spiegeln

Nach dem Entpacken ggf. Ausführbarkeit setzen: chmod +x *.sh

HINWEIS: Zusätzlich enthalten ist eine Gitlab CI/CD Pipeline Konfiguration: .gitlab-ci.yml. Diese Datei definiert eine vierstufige Pipeline:

Stage	**Zweck**
inspect	Prüft Metadaten des ubuntu:22.04-Images
copy	Kopiert das Image in eine Registry ($DEST_REGISTRY)
sign	Platzhalter für Signierung (mit Secret Mgmt erweiterbar)
sync	Spiegelt das Repository lokal (nur exemplarisch)


thank you.
#TechforPeople.
devoteam

contact us.

Dirk Radde
dirk.radde@devoteam.com
+49 170 7822288