Einführung in Docker für Lernende Informatik

Zielgruppe: Lernende Informatik EFZ in den Fachrichtungen Applikationsentwicklung und Plattformentwicklung.

Als erstes startest du bitte mit dem learn Docker Kurs.

Erst danach beginnst du mit den Aufgaben!

Anschliessend unsere Aufgaben 1 - 10 lösen.

Ausgangslage

Docker ist ein Werkzeug, um Anwendungen und Dienste in Containern auszuführen. Container enthalten eine Anwendung mit ihrer Laufzeitumgebung, Konfiguration und Abhängigkeiten. Dadurch läuft eine Anwendung auf unterschiedlichen Systemen möglichst gleich.

Docker ist für beide Fachrichtungen relevant:

- Applikationsentwicklung: lokale Entwicklungsumgebungen, Datenbanken, Testumgebungen, reproduzierbare Builds, CI/CD und Deployment.
- Plattformentwicklung: Betrieb von Services, Netzwerke, Volumes, Images, Registries, Troubleshooting, Sicherheit und Automatisierung.

Diese Einführung soll nicht nur Befehle vermitteln, sondern ein grundlegendes Verständnis schaffen, wie Container, Images, Volumes, Netzwerke und Compose zusammenhängen.

Lernziele

Die Lernenden können nach der Einführung:

- erklären, was ein Container ist und wie er sich von einer VM unterscheidet.
- den Unterschied zwischen Image und Container beschreiben.
- Docker Desktop unter Windows installieren und prüfen.
- einfache Container mit `docker run` starten und beenden.
- Images mit `docker build` aus einem `Dockerfile` erstellen.
- Container-Ports auf den Host weiterleiten.
- Volumes für persistente Daten verwenden.
- Docker-Netzwerke für die Kommunikation zwischen Containern nutzen.
- mit `docker compose` mehrere Services gemeinsam starten.
- einen einfachen Stack mit Postgres und pgAdmin betreiben.
- Logs anzeigen und grundlegendes Troubleshooting durchführen.
- einschätzen, wann Docker sinnvoll ist und wo Grenzen liegen.

Voraussetzungen

Empfohlen:

- Grundkenntnisse im Umgang mit Windows.
- Grundkenntnisse in der Kommandozeile, zum Beispiel PowerShell oder Linux-Shell.
- Grundkenntnisse zu Dateien, Verzeichnissen und Pfaden.
- Grundkenntnisse in Git.
- Erste Erfahrung mit einer Programmiersprache, zum Beispiel JavaScript, Python, Java oder C#.
- Grundverständnis von TCP/IP, Ports und localhost.

Hilfreich, aber nicht zwingend:

- Grundkenntnisse in Linux.
- Erfahrung mit Paketmanagern wie `apt`, `npm`, `pip` oder `maven`.
- Grundkenntnisse zu Datenbanken.
- Grundkenntnisse zu YAML.

Wichtige Begriffe, die vor oder während der Einführung geklärt werden sollten:

- Datei und Verzeichnis
- Terminal / Shell
- Prozess
- Port
- IP-Adresse
- localhost
- Umgebungsvariable
- Datenbank
- Client und Server
- Git Repository

Muss-Vorwissen

Die Lernenden sollten vor Docker sicher können:

- sich in einer Shell orientieren.
- Dateien und Verzeichnisse erstellen, anzeigen, kopieren, verschieben und löschen.
- absolute und relative Pfade unterscheiden.
- einfache Textdateien mit einem Editor wie `nano` oder VS Code bearbeiten.
- Befehle mit Optionen und Argumenten lesen.
- Paketlisten aktualisieren und Pakete installieren.
- Prozesse anzeigen und beenden.
- Umgebungsvariablen anzeigen und setzen.
- grundlegende Netzwerkbegriffe wie IP-Adresse, Port, DNS und `localhost` erklären.
- einfache Netzwerkdiagnosen mit `ping`, `ip`, `ss` oder `curl` durchführen.
- Git-Repositories klonen und Projektdateien im Terminal öffnen.

Besonders wichtige Begriffe für Docker

Diese Begriffe sollten vor oder während der Docker-Einführung nochmals gezielt aufgegriffen werden:

- Prozess: Ein Container startet normalerweise einen Hauptprozess.
- Dateisystem: Images und Container enthalten eigene Dateisysteme.
- Pfad: Volumes und Bind Mounts verbinden Host-Pfade mit Container-Pfaden.
- Port: Mit Portweiterleitungen wird ein Dienst im Container vom Host erreichbar.
- Netzwerk: Container können über Docker-Netzwerke miteinander kommunizieren.
- DNS-Name: In Docker Compose können Services über ihren Servicenamen erreicht werden.
- Umgebungsvariable: Viele Images werden über Environment Variables konfiguriert.
- Paketmanager: Ein `Dockerfile` installiert Abhängigkeiten oft mit `apt`, `npm`, `pip` oder ähnlichen Werkzeugen.
- Benutzerrechte: Container laufen mit einem Benutzer; Root im Container ist nicht gleichbedeutend mit sicherem Betrieb.
- Logausgabe: Container-Logs sind für Fehlersuche zentral.

Sinnvolle Reihenfolge

Bevor Docker eingeführt wird, sollten mindestens diese Linux-Module behandelt sein:

1. Shell-Grundlagen
2. Dateisystem und Pfade
3. Dateien anzeigen und bearbeiten
4. Paketverwaltung
5. Prozesse
6. Netzwerkgrundlagen
7. Entwicklungswerkzeuge mit Git und Umgebungsvariablen

Docker kann danach als nächster Schritt eingeführt werden. Für Applikationsentwickler ist besonders die Verbindung von App, Datenbank, Ports und Umgebungsvariablen wichtig. Für Plattformentwickler sind Prozesse, Netzwerke, Volumes, Logs und Betriebsaspekte besonders wichtig.

Mini-Check vor Docker

Die Lernenden sollten diese Aufgaben ohne Schritt-für-Schritt-Anleitung lösen können:

```
pwd
mkdir docker-vorwissen
cd docker-vorwissen
echo "Hallo Docker" > index.html
cat index.html
env | head
ip addr
ss -tulpen
curl http://example.com
git --version
```

Wenn diese Aufgaben verstanden werden, ist der Einstieg in Docker deutlich einfacher.

Empfohlene Arbeitsumgebung

Einstieg: Docker Desktop unter Windows

Für den Einstieg wird Docker Desktop unter Windows empfohlen. Das ist für Lernende am einfachsten, weil Docker Engine, Docker CLI, Docker Compose und die Integration mit WSL 2 gemeinsam installiert werden.

Empfohlene Variante:

- Betriebssystem: Windows 10 oder Windows 11
- Docker: Docker Desktop
- Backend: WSL 2
- Terminal: PowerShell, Windows Terminal oder VS Code Terminal
- Editor: Visual Studio Code
- Später: Projektdateien in WSL bearbeiten und Docker Compose aus WSL ausführen

Warum Docker Desktop für den Einstieg:

- einfache Installation
- grafische Übersicht über Container, Images, Volumes und Logs
- Docker Compose ist direkt enthalten
- gute Integration mit WSL 2

- weniger Einstiegshürden als eine manülle Docker-Installation in Linux

Später: Arbeiten aus WSL

Nach den ersten Grundlagen kann die Arbeit schrittweise nach WSL verschoben werden. Das ist besonders sinnvoll, wenn die Lernenden bereits mit Linux, Git und VS Code Remote WSL arbeiten.

Empfohlenes Vorgehen:

- Docker Desktop bleibt unter Windows installiert.
- WSL 2 wird als Backend verwendet.
- In Docker Desktop wird die gewünschte WSL-Distribution integriert.
- Projektdateien liegen im Linux-Dateisystem von WSL, zum Beispiel unter `~/projects`.
- `docker` und `docker compose` werden im WSL-Terminal ausgeführt.

Wichtig:

- Projektdateien für Linux-Workflows nicht dauerhaft unter `/mnt/c/...` ablegen.
- Besser: Repository in WSL klonen, zum Beispiel nach `~/projects/docker-intro`.
- VS Code mit `code .` aus WSL starten.

Dadurch werden Dateizugriffe schneller, Pfade sind konsistenter und typische Linux-Tools funktionieren besser.

Installation unter Windows

Voraussetzungen prüfen

Vor der Installation:

- Windows ist aktuell.
- Virtualisierung ist im BIOS/UEFI aktiviert.
- WSL 2 ist installiert oder kann installiert werden.
- Der Benutzer hat lokale Administratorrechte für die Installation.

Prüfen in PowerShell:

```
wsl --status
```

Falls WSL noch nicht installiert ist:

```
wsl --install
```

Danach Windows neu starten, falls gefordert.

Docker Desktop installieren

Vorgehen:

1. Docker Desktop für Windows von der offiziellen Docker-Webseite herunterladen.
2. Installer starten.
3. Option für WSL 2 Backend aktiviert lassen.
4. Installation abschliessen.
5. Windows neu starten, falls gefordert.
6. Docker Desktop starten.
7. Warten, bis Docker Desktop den Status “Running” anzeigt.

Nach der Installation in PowerShell prüfen:

```
docker version
docker compose version
docker run hello-world
```

Wenn `hello-world` erfolgreich läuft, ist Docker grundsätzlich einsatzbereit.

WSL-Integration aktivieren

In Docker Desktop:

1. Settings öffnen.
2. Resources auswählen.
3. WSL Integration auswählen.
4. Gewünschte Distribution aktivieren, zum Beispiel Ubuntu oder Debian.
5. Apply & Restart ausführen.

Danach im WSL-Terminal prüfen:

```
docker version
docker compose version
docker run hello-world
```

Grundbegriffe

Image

Ein Image ist eine Vorlage für Container. Es enthält ein Dateisystem, Programme, Bibliotheken und Startinformationen.

Beispiele:

- `nginx`
- `postgres`
- `node`
- `python`
- eigenes Image aus einem `Dockerfile`

Images werden aus Registries geladen, meistens von Docker Hub.

```
docker pull nginx
docker images
```

Container

Ein Container ist eine laufende oder gestoppte Instanz eines Images.

Beispiel:

```
docker run nginx
```

Ein Container kann gestartet, gestoppt, gelöscht und neu erstellt werden. Daten im Container sind ohne Volume normalerweise nicht dauerhaft.

Registry

Eine Registry ist ein Speicherort für Images. Die bekannteste Registry ist Docker Hub.

Beispiel:

```
docker pull postgres:16
```

`postgres` ist der Image-Name, 16 ist der Tag.

Dockerfile

Ein `Dockerfile` beschreibt, wie ein eigenes Image gebaut wird.

Ein einfaches Beispiel:

```
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
```

Daraus wird mit `docker build` ein Image erstellt.

Docker Compose

Docker Compose beschreibt mehrere zusammengehörende Container in einer YAML-Datei. Diese Datei heisst meistens `compose.yaml` oder `docker-compose.yml`.

Compose eignet sich besonders für Entwicklungsumgebungen mit mehreren Services, zum Beispiel:

- Webanwendung
- Datenbank
- Admin-Tool
- Cache
- Message Qüü

Erste Befehle

Version anzeigen

```
docker version
docker compose version
```

Testcontainer starten

```
docker run hello-world
```

Laufende Container anzeigen

```
docker ps
```

Alle Container anzeigen

```
docker ps -a
```

Container stoppen

```
docker stop <container-name-oder-id>
```

Container löschen

```
docker rm <container-name-oder-id>
```

Images anzeigen

```
docker images
```

Logs anzeigen

```
docker logs <container-name-oder-id>
```

docker run

Mit `docker run` wird aus einem Image ein Container erstellt und gestartet.

Ein einfacher Nginx-Webserver:

```
docker run --name web -p 8080:80 nginx
```

Danach im Browser öffnen:

```
http://localhost:8080
```

Bedeutung:

- `--name web`: Container heisst `web`
- `-p 8080:80`: Port 8080 auf dem Host wird auf Port 80 im Container weitergeleitet
- `nginx`: verwendetes Image

Container im Hintergrund starten:

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

Container stoppen und löschen:

```
docker stop web
docker rm web
```

docker build

Mit `docker build` wird aus einem `Dockerfile` ein eigenes Image erstellt.

Beispielprojekt

Dateien:

```
docker-build-demo/
  Dockerfile
  index.html
```

`index.html`:

```
<!doctype html>
<html lang="de">
<head>
  <meta charset="utf-8">
  <title>Docker Demo</title>
</head>
<body>
  <h1>Hallo Docker</h1>
</body>
</html>
```

Dockerfile:

```
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
```

Image baün:

```
docker build -t docker-demo .
```

Container starten:

```
docker run -d --name docker-demo-web -p 8080:80 docker-demo
```

Browser:

```
http://localhost:8080
```

Aufräumen:

```
docker stop docker-demo-web
docker rm docker-demo-web
```

Volumes

Container sind grundsätzlich kurzlebig. Wenn ein Container gelöscht wird, gehen Daten im Container verloren. Für dauerhafte Daten verwendet man Volumes.

Volume erstellen:

```
docker volume create postgres-data
```

Volumes anzeigen:

```
docker volume ls
```

Postgres mit Volume starten:

```
docker run -d \
  --name postgres \
  -e POSTGRES_USER=app \
  -e POSTGRES_PASSWORD=secret \
  -e POSTGRES_DB=appdb \
  -p 5432:5432 \
  -v postgres-data:/var/lib/postgresql/data \
  postgres:16
```

Bedeutung:

- `-e`: setzt Umgebungsvariablen
- `-v postgres-data:/var/lib/postgresql/data`: speichert Daten in einem Docker Volume
- `postgres:16`: Postgres Image mit Version 16

Container löschen, Daten behalten:

```
docker stop postgres
docker rm postgres
docker volume ls
```

Das Volume bleibt bestehen.

Volume löschen:

```
docker volume rm postgres-data
```

Vorsicht: Dadurch werden die gespeicherten Daten gelöscht.

Networks

Container können über Docker-Netzwerke miteinander kommunizieren. In einem gemeinsamen Netzwerk können sie sich über ihre Container-Namen erreichen.

Netzwerk erstellen:

```
docker network create app-net
```

Postgres im Netzwerk starten:

```
docker run -d \
  --name postgres \
  --network app-net \
  -e POSTGRES_USER=app \
  -e POSTGRES_PASSWORD=secret \
  -e POSTGRES_DB=appdb \
  -v postgres-data:/var/lib/postgresql/data \
  postgres:16
```

Wichtig:

- Andere Container im gleichen Netzwerk erreichen die Datenbank unter dem Hostnamen `postgres`.
- Vom Windows-Host aus ist Postgres nur erreichbar, wenn ein Port mit `-p` veröffentlicht wird.
- Für Container-zu-Container-Kommunikation braucht es normalerweise kein `-p`.

Docker Compose

Mit Docker Compose wird eine Umgebung in einer Datei beschrieben. Das ist übersichtlicher als lange `docker run` Befehle.

Grundstruktur

`compose.yaml`:

```yaml
services:
  web:
    image: nginx:alpine
    ports:
      - "8080:80"
```

Starten:

```
docker compose up
```

Im Hintergrund starten:

```
docker compose up -d
```

Stoppen:

```
docker compose down
```

Logs anzeigen:

```
docker compose logs
docker compose logs -f
```

Status anzeigen:

```
docker compose ps
```

Beispiel: Postgres und pgAdmin mit Compose

Dieses Beispiel startet zwei Services:

- `postgres`: Datenbankserver
- `pgadmin`: Weboberfläche zur Verwaltung von Postgres

Datei `compose.yaml`:

```yaml
services:
  postgres:
    image: postgres:16
    container_name: intro-postgres
    environment:
      POSTGRES_USER: app
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: appdb
    volumes:
      - postgres-data:/var/lib/postgresql/data
    networks:
      - app-net

  pgadmin:
    image: dpage/pgadmin4:latest
    container_name: intro-pgadmin
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: secret
    ports:
      - "8080:80"
    depends_on:
      - postgres
    networks:
      - app-net

volumes:
  postgres-data:

networks:
  app-net:
```

Starten:

```
docker compose up -d
```

Status prüfen:

```
docker compose ps
```

Logs anzeigen:

```
docker compose logs -f
```

pgAdmin im Browser öffnen:

```
http://localhost:8080
```

Login:

- E-Mail: `admin@example.com`
- Passwort: `secret`

Server in pgAdmin registrieren:

- Name: `intro-postgres`
- Host name/address: `postgres`
- Port: `5432`
- Maintenance database: `appdb`
- Username: `app`
- Password: `secret`

Warum ist der Hostname `postgres`?

In Docker Compose erhalten Services im gleichen Netzwerk automatisch DNS-Namen. Der Service `pgadmin` kann deshalb den Service `postgres` direkt unter dem Namen `postgres` erreichen.

Stoppen:

```
docker compose down
```

Stoppen und Volume löschen:

```
docker compose down -v
```

Vorsicht: `-v` löscht die Datenbankdaten.

Typischer Lernpfad

Modul 1: Was ist Docker?

Inhalte:

- Problem: “Läuft bei mir, aber nicht bei dir”
- Unterschied zwischen VM und Container
- Image, Container, Registry
- Docker Desktop und Docker Engine

Übung:

```
docker run hello-world
docker run -d --name web -p 8080:80 nginx
docker ps
docker logs web
docker stop web
docker rm web
```

Modul 2: Eigene Images baün

Inhalte:

- `Dockerfile`
- `FROM`
- `COPY`
- Build-Kontext
- Image-Tags

Übung:

- einfache HTML-Datei erstellen
- eigenes Nginx-Image baün
- Container starten
- Seite im Browser prüfen

Modul 3: Ports und Umgebungsvariablen

Inhalte:

- Host-Port und Container-Port
- `-p 8080:80`
- `localhost`
- `-e` für Umgebungsvariablen

Übung:

- Nginx auf verschiedenen Host-Ports starten
- Postgres mit Benutzer, Passwort und Datenbank starten

Modul 4: Volumes

Inhalte:

- kurzlebige Container
- persistente Daten
- named volumes
- Unterschied zwischen Volume und Bind Mount

Übung:

- Postgres mit Volume starten
- Container löschen
- Container mit gleichem Volume neu starten
- Datenbestand prüfen

Modul 5: Networks

Inhalte:

- Docker-Netzwerke
- Container-zu-Container-Kommunikation
- Service-Namen als DNS-Namen
- Unterschied zwischen interner Kommunikation und veröffentlichten Ports

Übung:

- eigenes Netzwerk erstellen
- Postgres und pgAdmin im gleichen Netzwerk starten
- Verbindung über Container-Namen testen

Modul 6: Docker Compose

Inhalte:

- `compose.yaml`
- Services
- Volumes

- Networks
- depends_on
- `docker compose up`, `down`, `logs`, `ps`

Übung:

- Postgres und pgAdmin als Compose-Projekt starten
- Datenbank in pgAdmin registrieren
- Stack stoppen und erneut starten

Modul 7: Troubleshooting

Inhalte:

- Logs lesen
- Containerstatus prüfen
- Portkonflikte erkennen
- falsche Passwörter und Umgebungsvariablen
- Volumes bewusst löschen
- Images neu baün

Nützliche Befehle:

```
docker ps
docker ps -a
docker logs <container>
docker inspect <container>
docker compose ps
docker compose logs -f
docker compose down
docker compose down -v
docker system df
```

Wichtige Konzepte für Applikationsentwicklung

Applikationsentwickler sollten besonders verstehen:

- Wie man eine lokale Entwicklungsumgebung reproduzierbar macht.
- Wie eine App mit einer Datenbank verbunden wird.
- Wie Ports zwischen Host und Container funktionieren.
- Wie Umgebungsvariablen für Konfiguration verwendet werden.
- Warum Datenbankdaten in Volumes liegen sollten.
- Wie `compose.yaml` im Team geteilt wird.
- Warum Secrets nicht direkt in Git gespeichert werden sollten.

Typisches Beispiel:

```yaml
services:
  app:
    build: ../..
    ports:
      - "3000:3000"
    environment:
      DATABASE_URL: postgres://app:secret@postgres:5432/appdb
    depends_on:
      - postgres
```

```
  postgres:
    image: postgres:16
    environment:
      POSTGRES_USER: app
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: appdb
    volumes:
      - postgres-data:/var/lib/postgresql/data

volumes:
  postgres-data:
```

In dieser Umgebung verbindet sich die App nicht mit `localhost`, sondern mit `postgres`, weil die Datenbank als Compose-Service so heisst.

Wichtige Konzepte für Plattformentwicklung

Plattformentwickler sollten besonders verstehen:

- Wie Container-Prozesse auf dem Host laufen.
- Wie Images heruntergeladen, versioniert und gelöscht werden.
- Wie Netzwerke, Ports und DNS in Docker funktionieren.
- Wie persistente Daten mit Volumes verwaltet werden.
- Wie Logs, Status und Ressourcenverbrauch analysiert werden.
- Welche Sicherheitsgrenzen Container haben und welche nicht.
- Warum Container nicht automatisch vollwertige VMs ersetzen.

Vertiefungsthemen:

- eigene Registries
- Image-Scanning
- Rootless Docker
- Reverse Proxy
- Backup von Volumes
- Healthchecks
- Resource Limits
- CI/CD mit Container-Builds

Häfige Fehler

Port ist bereits belegt

Symptom:

`bind: address already in use`

Lösung:

- anderen Host-Port verwenden, zum Beispiel `8081:80`
- laufende Container prüfen:

`docker ps`

Container startet und beendet sich sofort

Prüfen:

```
docker ps -a
docker logs <container>
```

Mögliche Ursachen:

- falsche Umgebungsvariablen
- falscher Startbefehl
- Anwendung crasht beim Start
- fehlende Datei im Image

Daten sind nach `docker compose down -v` weg

`docker compose down -v` löscht Volumes. Für normale Stopps nur verwenden:

```
docker compose down
```

App findet Datenbank nicht

In Compose gilt:

- Von Host zu Datenbank: `localhost`, wenn Port veröffentlicht ist.
- Von Container zu Container: Service-Name, zum Beispiel `postgres`.

Falsch innerhalb eines App-Containers:

```
localhost:5432
```

Richtig innerhalb eines App-Containers:

```
postgres:5432
```

Mindestbefehle, die sitzen müssen

```
docker version
docker run hello-world
docker ps
docker ps -a
docker stop <container>
docker rm <container>
docker images
docker logs <container>
docker build -t <name> .
docker run -d --name <name> -p <host-port>:<container-port> <image>
docker volume ls
docker network ls
docker compose up -d
docker compose ps
docker compose logs -f
docker compose down
```

Empfohlene Regeln für Lernende

- Container dürfen gelöscht und neu erstellt werden.

- Wichtige Daten gehören in Volumes.
- Compose-Dateien gehören ins Git-Repository.
- Passwörter in Beispielen sind nur für lokale Übungen geeignet.
- Für echte Projekte werden Secrets anders verwaltet.
- Container untereinander sprechen in Compose über Service-Namen.
- `docker compose down -v` nur verwenden, wenn Daten bewusst gelöscht werden sollen.
- Bei Fehlern zuerst `docker ps -a` und `docker logs` verwenden.

Abschlussaufgabe

Die Lernenden erstellen ein eigenes Compose-Projekt mit:

- einer einfachen Webanwendung oder statischen Webseite
- einer Postgres-Datenbank
- pgAdmin
- einem persistenten Volume für Postgres
- einem gemeinsamen Docker-Netzwerk
- einer kurzen README mit Start-, Stop- und Login-Anleitung

Mindestanforderungen:

- `docker compose up -d` startet alle Services.
- pgAdmin ist im Browser erreichbar.
- pgAdmin kann sich mit Postgres verbinden.
- Daten bleiben nach `docker compose down` erhalten.
- Die Umgebung kann mit `docker compose down -v` vollständig zurückgesetzt werden.

Optionale Erweiterungen:

- eigene App mit Datenbankverbindung
- `.env` Datei für Konfiguration
- Healthcheck für Postgres
- Adminer statt pgAdmin vergleichen
- Backup und Restore eines Datenbank-Dumps