

# Docker Cheatsheet - Inhaltsverzeichnis

---

## 1. Dateioperationen & Persistenz

- [Docker CP, Bind Mounts & Volumes](#)
  - Docker CP - Datei in Container kopieren
  - Bind Mounts - Live-Synchronisation mit Host
  - Volumes - Persistente Datenspeicherung
  - Vergleich der Methoden

## 2. Docker Volumes

- [Volumes mit Docker - Daten dauerhaft speichern](#)
  - Benannte Volumes erstellen
  - Volumes verwalten
  - Praktisches Beispiel mit MariaDB

## 3. Netzwerke

- [Docker-Netzwerke - Container vernetzen](#)
  - Netzwerkarten in Docker
  - Bridge-Netzwerk verstehen
  - Eigene Netzwerke erstellen
  - Kommunikation zwischen Containern
  - Debugging & Analyse
- [Container mit Docker-Netzwerken verbinden & trennen](#)
  - Container mit mehreren Netzwerken
  - Netzwerkverbindungen hinzufügen und entfernen
  - Prüfen der Netzwerkkonfiguration

## 4. Praktische Anwendungen

- [WordPress mit MariaDB & phpMyAdmin](#)
  - Komplettes Setup mit Netzwerk
  - Datenbank konfigurieren
  - WordPress einrichten

---

## Übersicht der Hauptthemen

Cheatsheet	Kernthemen	Praktische Anwendung
CP, Mounts & Volumes	Datenaustausch, Synchronisation	Entwicklungsumgebungen
Volumes	Persistente Datenspeicherung	Datenbanken, Konfigurationsdateien
Netzwerke	Container-Kommunikation	Microservices, Multi-Container-Apps

Cheatsheet	Kernthemen	Praktische Anwendung
WordPress-Setup	Komplettlösung	Webhosting, CMS-Deployment

# 💻 Docker Cheatsheet – cp, Bind Mounts & Volumes

## ◊ 1. docker cp – Datei vom Host in den Container kopieren

### Ziel:

Dateien vom Host in einen laufenden Container verschieben und dort weiterverwenden.

### Vorbereitung:

```
echo "Hallo, das ist eine Testdatei für Docker und M347!" > meine_datei.txt
docker run -d --name mein_container ubuntu sleep 6000
```

### Datei kopieren:

```
docker cp meine_datei.txt mein_container:/root/
```

### Überprüfung im Container:

```
docker exec -it mein_container bash
cd /root/
ls
cat meine_datei.txt
exit
```

### Erfolgsnachweis:

- Datei ist im Container vorhanden (`/root/meine_datei.txt`)
- Inhalt wird korrekt angezeigt

### Bonus – Aufräumen:

```
docker stop mein_container
docker rm mein_container
```

## ◊ 2. Bind Mounts – Ordner auf dem Host im Container verfügbar machen

### Ziel:

Ordner und Dateien vom Host mit dem Container synchron halten.

### 📋 Vorbereitung:

```
mkdir mein_mount  
cd mein_mount  
echo "Hallo, dies ist eine Datei für Bind Mounts!" > testdatei.txt
```

### 🔧 Container starten mit Bind Mount:

```
docker run -d --name mount_container -v ${pwd}:/mnt/daten ubuntu sleep 6000
```

### 🔍 Überprüfung im Container:

```
docker exec -it mount_container bash  
cd /mnt/daten  
ls  
cat testdatei.txt
```

### 📝 Änderung auf dem Host:

```
echo "Diese Zeile wurde nachträglich hinzugefügt!" >> mein_mount/testdatei.txt
```

### 📋 Check im Container:

```
cat /mnt/daten/testdatei.txt
```

### 📋 Erfolgsnachweis:

- Datei sichtbar & editierbar im Container
- Änderungen vom Host sofort im Container sichtbar

### 📝 Bonus – Datei im Container erstellen und auf dem Host prüfen:

```
touch /mnt/daten/container_datei.txt  
exit
```

```
ls mein_mount
```

### ◊ 3. Bind Mounts mit -v Syntax (explizit & Rechte testen)

#### 📋 Vorbereitung:

```
mkdir \mein_mount  
echo "Hallo, dies ist eine Datei für Bind Mounts mit -v!" >  
\mein_mount\testdatei.txt
```

#### 🚀 Container starten:

```
cd \mein_mount  
docker run -d --name mount_container -v ${pwd}:/mnt/daten ubuntu sleep 6000
```

#### 🔍 Container prüfen:

```
docker exec -it mount_container bash  
cd /mnt/daten  
ls  
cat testdatei.txt
```

#### 📝 Datei im Container bearbeiten:

```
echo "Diese Zeile wurde im Container hinzugefügt!" >> /mnt/daten/testdatei.txt
```

#### 📱 Host prüfen:

Öffne die Datei \mein\_mount\testdatei.txt mit z.B. **notepad** oder **cat**.

#### 📋 Erfolgsnachweis:

- Datei sichtbar im Container
- Änderungen beidseitig synchronisiert

#### 💡 Bonus – Read-Only Bind Mount:

```
docker run -d --name mount_READONLY -v ${pwd}:/mnt/daten:ro ubuntu sleep 60
docker exec -it mount_READONLY bash -c "echo 'Test' > /mnt/daten/neue_datei.txt"
```

⌚ **Erwartung:** Fehlermeldung wegen **read-only** → gute Grundlage zur Diskussion

## ◊ 4. Volumes (Kurzfassung für Kontext im Vergleich)

Volumes sind persistente Datencontainer, die unabhängig vom Lifecycle des Containers existieren.

### Anlegen & Verwenden:

```
docker volume create mein_volume
docker run -d --name vol_container -v mein_volume:/app/data ubuntu sleep 6000
```

### Check:

```
docker exec -it vol_container bash
cd /app/data
touch test_vol.txt
ls
```

⌚ **Vorteile von Volumes:**

- Persistenz auch bei Container-Neustart
- Besser für komplexe Datenspeicherung
- Einfaches Backup möglich

## ⌚ 5. Vergleich & Anwendungsfälle

Methode	Einsatzbereich	Persistenz	Synchronisierung	Zugriff
docker cp	Einzelne Dateien kopieren	Nein	Keine	Einmalig
Bind Mounts	Echtzeit-Zugriff auf lokale Dateien	Ja	Sofortig	Beidseitig
Volumes	Langfristige Datenspeicherung	Ja	Automatisch	Container-Only

## ⌚ 6. Reflexion & Abschluss

❖ **Mögliche Diskussionspunkte:**

- Wann setze ich welche Methode ein?
- Welche Variante ist am sichersten / flexibelsten?
- Wie beeinflussen Dateiberechtigungen die Arbeit?

## ◊ Volumes mit Docker – Daten dauerhaft speichern

### ▀ Benanntes Volume erstellen und mit einem Container verknüpfen

Mit einem Volume kannst du sicherstellen, dass Daten erhalten bleiben, selbst wenn der Container gelöscht wird.

```
docker run -d -v my_mariadb_data:/var/lib/mysql -e MARIADB_ROOT_PASSWORD=sml12345 mariadb
```

- **-v my\_mariadb\_data:/var/lib/mysql**: Mountet das Volume `my_mariadb_data` auf das Standard-Datenverzeichnis von MariaDB.
- **-e MARIADB\_ROOT\_PASSWORD=...**: Setzt das Root-Passwort für MariaDB.

### 🔍 Mount überprüfen

```
docker ps      # Container-ID kopieren  
docker inspect -f "{{.Mounts}}" <CONTAINER_ID>
```

Du solltest etwas Ähnliches sehen wie:

```
[{volume my_mariadb_data ... /var/lib/mysql ...}]
```

### ▀ Alle Volumes anzeigen

```
docker volume ls
```

Das benannte Volume `my_mariadb_data` wird hier gelistet und bleibt erhalten – auch wenn du den Container stoppst oder löschst.

### ✗ Volumes löschen (Vorsicht!)

Wenn du versuchst, ein Volume zu löschen, das noch mit einem (auch gestoppten) Container verknüpft ist:

```
docker volume rm <VOLUME_NAME>
```

... bekommst du **eine Fehlermeldung**, z.B.:

```
volume is in use - [<CONTAINER_ID>]
```

💡 Lösung:

```
docker rm <CONTAINER_ID>
docker volume rm <VOLUME_NAME>
```

Nur wenn **kein Container mehr das Volume benutzt**, kannst du es entfernen.

---

## 📝 Beispiel: Automatisch generiertes Volume

Wenn du **kein Volume angibst**, erstellt Docker automatisch eins:

```
docker run -d -e MARIADB_ROOT_PASSWORD=sml12345 mariadb
```

Mit `docker volume ls` siehst du ein kryptisch benanntes Volume, das mit dem Container verknüpft wurde.

⚠ Auch hier: Erst den Container löschen, **dann** das Volume entfernen.

---

## 📋 Persistente Datenbank mit MariaDB & Volume

MariaDB mit Volume + echte Daten rein:

```
docker run -d -v maria_db:/var/lib/mysql --name old_db -e
MARIADB_ROOT_PASSWORD=sml12345 mariadb:10.5
```

Dann rein in die DB:

```
docker exec -it old_db mysql -u root -psml12345
```

In der SQL-Shell:

```
CREATE DATABASE docker_containers_db;
USE docker_containers_db;

CREATE TABLE docker_containers_table (
    CONTAINER_ID VARCHAR(13),
    IMAGE VARCHAR(20),
    NAME VARCHAR(20)
);
```

```
INSERT INTO docker_containers_table VALUES
('605adf483577', 'mariadb', 'mariadb-test'),
('582c5cdbe015', 'mysql', 'mysql-test'),
('e09e0b567f53', 'postgres', 'postgres-test');

SELECT * FROM docker_containers_table;
```

Erwartete Ausgabe:

CONTAINER_ID	IMAGE	NAME
605adf483577	mariadb	mariadb-test
582c5cdbe015	mysql	mysql-test
e09e0b567f53	postgres	postgres-test

Beende die MySQL-Shell mit:

```
exit;
```

Dann den Container stoppen:

```
docker stop old_db
```

Das Volume **maria\_db** bleibt erhalten und enthält die komplette Datenbank – **persistente Speicherung funktioniert**

# 🌐 Docker-Netzwerke – Container vernetzen, verstehen & nutzen

## 🛠 Docker-Netzwerke

- Jeder Container ist **standardmäßig isoliert**.
- Kommunikation ist nur über ein gemeinsames Netzwerk möglich.
- Docker bietet vordefinierte Netzwerke und die Möglichkeit, eigene zu erstellen.

## 📝 Netzwerkarten in Docker

### Netzwerkmodus    Beschreibung

bridge	Standardnetzwerk; Container können über interne IPs kommunizieren
host	Container nutzt direkt das Netzwerkinterface des Hosts
none	Container hat <b>kein</b> Netzwerkzugang
Benutzerdefiniert	Manuell erstellte Netzwerke mit DNS-Namensauflösung

## 🔍 Bridge-Netzwerk verstehen

Wenn kein eigenes Netzwerk angegeben wird, landet der Container im Standard-Bridge-Netzwerk.

Eigenschaften:

Attribut	Beispiel
Netzwerkname	bridge
Subnetz	172.17.0.0/16
Gateway	172.17.0.1
Kommunikation	Container können sich anpingen
DNS	Nur in <b>benutzerdef. Netzwerken</b> voll funktionsfähig

## 🛠 Docker-Netzwerke anzeigen & analysieren

```
docker network ls
```

Zeigt alle verfügbaren Netzwerke an.

```
docker network inspect bridge
```

Liefert IP-Bereiche, Container im Netzwerk, Gateway etc.

---

### 📝 Container im Standardnetzwerk untersuchen

```
docker run -d --name web1 nginx
docker run -d --name ubuntu1 ubuntu sleep 9999
```

IP-Adresse prüfen:

```
docker inspect -f "{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}" web1
```

Ping vom zweiten Container:

```
docker exec -it ubuntu1 bash
apt update && apt install -y iputils-ping
ping <IP-Adresse-von-web1>
```

---

### 🌐 Eigenes Netzwerk erstellen und nutzen

```
docker network create webapp
```

Container gezielt in dieses Netzwerk starten:

```
docker run --network webapp -d --name web2 nginx
docker run --network webapp -it -d --name client ubuntu
```

---

### 💬 Kommunikation über Containernamen testen

```
docker exec -it client bash
apt update && apt install -y iputils-ping
ping web2
```

**Namensauflösung funktioniert**, weil benutzerdefinierte Netzwerke DNS-Support haben.

---

### ✍️ Netzwerk wieder löschen

⚠ Nur möglich, wenn **kein Container** mehr damit verbunden ist:

```
docker network rm webapp
```

## ⌚ Typische Probleme & Hinweise

Problem	Ursache / Lösung
Ping funktioniert nicht	Container sind in unterschiedlichen Netzwerken
Containername kann nicht aufgelöst werden	Nur möglich in benutzerdefinierten Netzwerken
Netzwerk kann nicht gelöscht werden	Container nutzt es noch – erst <code>docker rm</code>

## 🔍 Nützliche Befehle für Debugging & Analyse

```
docker container inspect <container>
docker exec -it <container> bash
ip addr      # Zeigt Netzwerkschnittstellen im Container
ping <ziel>
```

## ☑ Praxisrelevanz

Docker-Netzwerke ermöglichen die Verbindung von:

- **Webservern und Datenbanken**
- **Microservices**
- **Frontend + Backend**

Sie sorgen für **Isolation, Flexibilität** und **Sicherheit** in modernen Entwicklungsumgebungen.

## 🔗 Container mit Docker-Netzwerken verbinden & trennen

### ❖ Mehrere Netzwerke für einen Container nutzen

Ein Docker-Container kann gleichzeitig mit **mehreren Netzwerken** verbunden sein. Das ermöglicht z. B. Kommunikation zwischen verschiedenen Komponenten oder Migration von einem Netzwerk zum anderen.

### ▀ Beispiel: Container zwischen Netzwerken bewegen

#### ► Ausgangslage:

Ein Container läuft bereits im benutzerdefinierten Netzwerk **webapp** – z. B.:

```
docker network create webapp
docker run --network webapp -d --name webapp_container nginx
```

### ✚ Container mit einem weiteren Netzwerk verbinden

```
docker network connect bridge webapp_container
```

- Verbindet **webapp\_container** zusätzlich mit dem Standardnetzwerk **bridge**.

### ⌚ Netzwerk prüfen

```
docker network inspect bridge
```

Im Abschnitt **Containers** sollte **webapp\_container** jetzt auftauchen.

### — Container von einem Netzwerk trennen

```
docker network disconnect webapp webapp_container
```

- Entfernt den Container aus dem Netzwerk **webapp**.

### ☑ Aktuelle Netzwerke des Containers anzeigen

```
docker container inspect webapp_container
```

Suche im Abschnitt `NetworkSettings.Networks` – hier steht genau, **mit welchen Netzwerken** der Container noch verbunden ist.

Nach der Trennung vom `webapp`-Netzwerk sollte nur noch `bridge` angezeigt werden.

## ⌚ Wichtig zu wissen

Aktion	Ergebnis
<code>network connect</code>	Fügt Container einem weiteren Netzwerk hinzu
<code>network disconnect</code>	Entfernt Netzwerkverbindung
<code>inspect</code>	Zeigt alle aktiven Netzwerkverbindungen
Nur manuell möglich	Container springt <b>nicht automatisch</b> zwischen Netzwerken
IP-Adresse pro Netzwerk	Container bekommt <b>eine IP pro Netzwerk</b>

## 📝 Beispiel – Kompletter Ablauf

```
docker network create webapp
docker run --network webapp -d --name webapp_container nginx

docker network connect bridge webapp_container
docker network inspect bridge | grep webapp_container

docker network disconnect webapp webapp_container
docker container inspect webapp_container
```

### 🔍 Ergebnis:

- Container ist jetzt **nur noch im bridge-Netzwerk**.
- Kommunikation im ursprünglichen `webapp`-Netzwerk ist nicht mehr möglich.

Das ist perfekt, um z. B. **Container im Betrieb umzuhängen**, Netzwerkkonfigurationen zu testen oder auch bestimmte **Sicherheitszonen** aufzubauen.

# 🌐 📦 🖥️ WordPress mit MariaDB & phpMyAdmin in Docker

## 🔧 Ziel: Web-App mit WordPress + Datenbank in Docker betreiben

- Alle Container sind im **gemeinsamen benutzerdefinierten Netzwerk**
- Keine externen Links, nur interne Kommunikation
- Zugriff auf:
  - phpMyAdmin: <http://localhost:8080>
  - WordPress: <http://localhost:8081>

### ◊ 1. Netzwerk erstellen

```
docker network create blog-network
```

### ◊ 2. MariaDB starten

```
docker run -d \  
  --name mariadb \  
  --network blog-network \  
  -e MARIADB_ROOT_PASSWORD=sml12345 \  
  mariadb
```

## 🔍 Check: Ist MariaDB bereit?

```
docker logs mariadb
```

Achte auf die Zeile:

**ready for connections**

## 🔍 IP-Adresse der Datenbank herausfinden

```
docker inspect -f "{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}"  
mariadb
```

**Alternativ:** In **blog-network** funktioniert auch der Containername **mariadb** als Hostname!

### ◊ 3. phpMyAdmin starten (extern zugänglich)

```
docker run -d \
--name phpmyadmin \
-e PMA_HOST=mariadb \
-p 8080:80 \
phpmyadmin
```

### ☒ Netzwerk wechseln (von bridge zu blog-network)

```
docker network connect blog-network phpmyadmin
docker network disconnect bridge phpmyadmin
```

---

### 🌐 Zugriff über Browser

<http://localhost:8080>

#### Login:

- Benutzer: `root`
  - Passwort: `sml12345`
- 

### 🔒 Neuen Benutzer & DB in phpMyAdmin anlegen

1. Neuer Benutzer: `blog`
2. Passwort generieren (z.B. über Passwortmanager oder `openssl rand`)
3. Gleicher Name für Datenbank: `blog`
4. **Alle Rechte gewähren**

📝 Passwort notieren, wird gleich gebraucht!

---

### ◊ 4. WordPress starten

```
docker run -d \
--name wordpress \
--network blog-network \
-e WORDPRESS_DB_HOST=mariadb \
-e WORDPRESS_DB_USER=blog \
-e WORDPRESS_DB_PASSWORD="MEIN_SICHERES_PASSWORT" \
-e WORDPRESS_DB_NAME=blog \
-p 8081:80 \
wordpress
```

## 🌐 WordPress einrichten

Gehe zu: <http://localhost:8081>

1. Sprache wählen
  2. Admin-Zugang erstellen (Benutzername, Passwort, E-Mail)
  3. Installation abschliessen
- 

### Fertig!

Du hast jetzt eine komplett dockerisierte Webanwendung – **modular, vernetzt, einsatzbereit**.