ADMIN:DOCS

Bash Grundkurs #3: Kontrollstrukturen In Bash

Meistere Verzweigungen und Schleifen in der Bash: if-then-else, moderner [[...]] Test-Operator, Regex-Matching, case-Anweisungen, for-, while-, until-Schleifen und select-Menüs.

Kategorie: Bash-Grundlagen **Veröffentlicht:** 24.10.2024 **Zuletzt geprüft:** 10.11.2024

Willkommen zum dritten Teil unserer technischen Serie über Linux und Bash-Programmierung!

Nachdem wir in Bash Grundkurs #1: Erstelle Dein Erstes Bash-Shell-Skript die Grundlagen der Skripterstellung und in Bash Grundkurs #2: Verwendung Von Variablen In Bash den Umgang mit Datentypen und Parameter-Expansionen vertieft haben, widmen wir uns heute dem logischen Nervensystem der Programmierung: **den Kontrollstrukturen**.

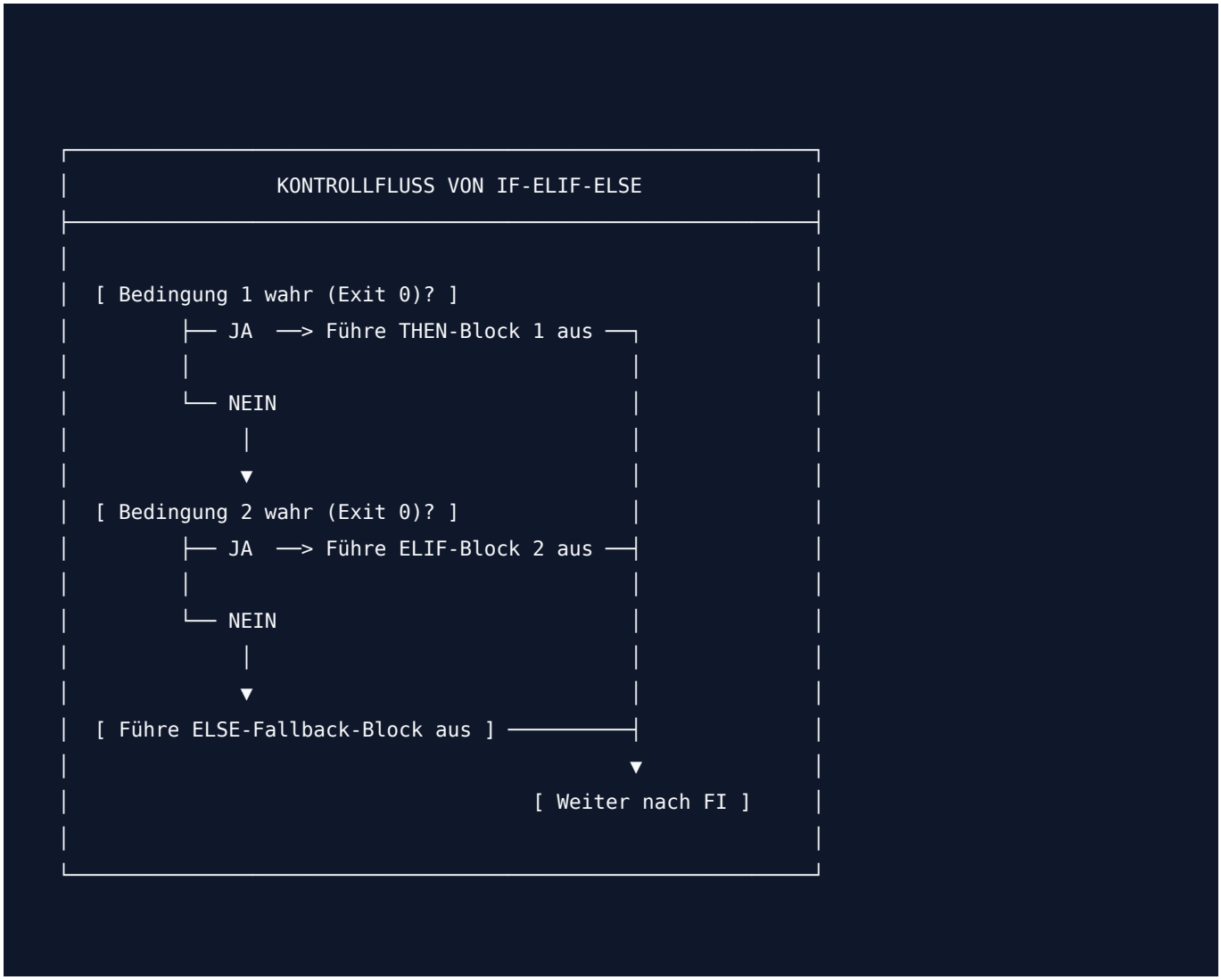
Reine Befehlsketten laufen starr von oben nach unten ab. In der realen Systemadministration müssen Skripte jedoch Entscheidungen treffen: Existiert eine Konfigurationsdatei? Ist ein Webserver-Port erreichbar? Hat ein Backup-Befehl fehlerfrei beendet? Müssen hunderte Logdateien nacheinander analysiert werden?

In dieser Lektion lernst du, wie du den Ablauf deiner Skripte präzise steuerst: Wir behandeln Verzweigungen mit `if-elif-else`, den modernen Test-Operator `[[ ... ]]` mitsamt regulären Ausdrücken, kompakte `case`-Musterabfragen, `for`- und `while`-Schleifen, sicheres zeilenweises Einlesen von Dateien sowie interaktive CLI-Menüs mit `select`.

Bedingte Verzweigungen: `if`, `elif`, `else` und `fi`

Die `if`-Anweisung prüft, ob ein Befehl oder ein Testausdruck erfolgreich (Exit-Code `0`) beendet wird. Ist die Bedingung erfüllt, wird der entsprechende Codeblock ausgeführt.

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-kontrollstrukturen-in-bash



Grundlegende Syntax

```
if [[ Bedingung1 ]]; then
    # Wird ausgeführt, wenn Bedingung 1 wahr ist
elif [[ Bedingung2 ]]; then
    # Wird ausgeführt, wenn Bedingung 2 wahr ist
else
    # Fallback: Wird ausgeführt, wenn keine Bedingung zutrifft
fi
```

> **Syntax-Tipp:** Das Semikolon vor `then` ist nur erforderlich, wenn `then` in derselben Zeile wie die Bedingung steht. Schreibst du `then` in eine neue Zeile, entfällt das Semikolon.

```
if [[ $alter -ge 18 ]]
then
    echo "Volljährig"
fi
```

Der moderne Test-Operator `[[ ... ]]` vs. POSIX `[ ... ]`

In älteren Skripten begegnet man häufig einfachen eckigen Klammern `[ ... ]`. Dabei handelt es sich um ein Synonym für das traditionelle Unix-Kommando `test`. In modernen Bash-Skripten ist die doppelte eckige Klammer `[[ ... ]]` der uneingeschränkte Goldstandard:

```
┌──────────────────────────────────────────────────────┐
│               VERGLEICH: [[ ... ]] VS. [ ... ]       │
├──────────────────────────────────────────────────────┤
│                                                      │
│   Eigenschaft             POSIX [ ... ]  Bash [[ ... ]]   │
│   ──────────────────────────────────────────────────   │
│   Word-Splitting          JA (Absturz!)  NEIN (Sicher)    │
│   Regex-Matching (=~)     NEIN           JA               │
│   Pattern-Matching        NEIN           JA (*.txt)       │
│   Logische Operatoren     -a / -o        && / ||          │
│   String-Vergleiche       \< / \>        < / >            │
│   Art des Konstrukts      Extern/Builtin Shell-Keyword    │
│                                                      │
```

```
    └──────────────────────────────────────────────────┘
```

Warum `[[ ... ]]` robuster und sicherer ist:

1. **Kein Absturz bei unvollständigen Variablen:**

``bash FILE="" # POSIX [ ]: Führt zu Syntaxfehler "[: =: unary operator expected" [ $FILE = "test" ] # Bash [[ ]]: Verarbeitet leere Variablen fehlerfrei ohne Absturz: [[ $FILE == "test" ]] ``

1. **Logische Verknüpfungen mit `&&` und `||` direkt im Ausdruck:**

```bash if [[ $alter -ge 18 && $ausweis == "ja" ]]; then

```
        echo "Zugang gewährt."
```

fi ```

1. **Mustervergleich mit Wildcards (`==`):**

```bash ARCHIV="backup_2024-10.tar.gz" if [[ "$ARCHIV" == *.tar.gz ]]; then

```
        echo "Es handelt sich um ein komprimiertes Tar-Archiv."
```

fi ```

1. **Reguläre Ausdrücke (*Regex-Matching* mit `=~`):**

```bash IP="192.168.1.100" # Prüfen, ob der String einer IPv4-Struktur entspricht: if [[ "$IP" =~ ^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$ ]]; then
```

```
        echo "Gültiges IP-Format."
```

fi ```

Vergleichsoperatoren: Zahlen, Strings & Dateisystem

Die Bash unterscheidet strikt zwischen numerischen Vergleichen, Zeichenketten-Prüfungen und Dateisystem-Tests.

1. Numerische Vergleiche (Ganzzahlen)

Operator	Bedeutung	Beispiel
`-eq`	Gleich (*equal*)	`[[ $a -eq $b ]]`
`-ne`	Ungleich (*not equal*)	`[[ $a -ne $b ]]`
`-lt`	Kleiner als (*less than*)	`[[ $a -lt $b ]]`
`-le`	Kleiner oder gleich (*less equal*)	`[[ $a -le $b ]]`
`-gt`	Größer als (*greater than*)	`[[ $a -gt $b ]]`
`-ge`	Größer oder gleich (*greater equal*)	`[[ $a -ge $b ]]`

Alternativ können für Zahlen auch doppelte runde Klammern `(( ... ))` mit den gewohnten mathematischen Operatoren (`==` , `!=` , `<` , `<=` , `>` , `>=`) genutzt werden:

```
if (( a >= 18 )); then
    echo "Volljährig via arithmetischer Evaluation."
```

```
fi
```

2. Zeichenketten-Vergleiche (Strings)

Operator	Bedeutung	Beispiel
`==` oder `=`	Strings sind identisch	`[[ "$a" == "$b" ]]`
`!=`	Strings sind ungleich	`[[ "$a" != "$b" ]]`
`-z`	String ist leer (*zero length*)	`[[ -z "$a" ]]`
`-n`	String ist **nicht** leer	`[[ -n "$a" ]]`
`<` / `>`	Lexikografische Sortierung	`[[ "$a" < "$b" ]]`

3. Dateisystem-Tests (*File Test Operators*)

Die Datei-Operatoren sind das wichtigste Werkzeug für Administrationsskripte:

Operator	Prüft, ob...	Anwendungsfall
`-f file`	...eine reguläre Datei existiert	`[[ -f /etc/nginx/nginx.conf ]]`
`-d dir`	...ein Verzeichnis existiert	`[[ -d /var/log/nginx ]]`
`-e path`	...der Pfad existiert (Datei, Ordner, Socket)	`[[ -e /tmp/app.sock ]]`
`-s file`	...die Datei existiert und **nicht leer** ist (> 0 Bytes)	`[[ -s /var/log/errors.log ]]`
`-r file`	...die Datei für den Benutzer **lesbar** ist	`[[ -r secret.key ]]`
`-w file`	...die Datei **beschreibbar** ist	`[[ -w /var/log/app.log ]]`
`-x file`	...die Datei **ausführbar** ist (*Execute-Bit*)	`[[ -x /usr/local/bin/deploy ]]`
`-L link`	...der Pfad ein symbolischer Link (*Symlink*) ist	`[[ -L /bin/sh ]]`
`f1 -nt f2`	...Datei 1 **neuer** ist als Datei 2 (*newer than*)	`[[ source.c -nt binary ]]`

Operator	Prüft, ob...	Anwendungsfall
`f1 -ot f2`	...Datei 1 **älter** ist als Datei 2 (*older than*)	`[[ backup.tar -ot current.tar ]]`

Mehrfachverzweigungen mit `case ... esac`

Wenn eine Variable gegen viele verschiedene Muster geprüft werden soll, ist `case` deutlich übersichtlicher und schneller als verschachtelte `if-elif-else`-Strukturen:

```
case "$VARIABLE" in
    muster1)
        # Befehle für muster1
        ;;
    muster2|muster3)
        # Mehrere Muster mit Pipe (|) verknüpfen
        ;;
    *)
        # Default-Zweig (entspricht 'else')
        ;;
esac
```

> ⚠ **Syntax-Hinweis:** Jeder Zweig in einer `case`-Anweisung muss zwingend mit zwei Semikolons (`;;`) abgeschlossen werden! Das schließende Schlüsselwort lautet `esac` (`case` rückwärts gelesen).

Praxisbeispiel: Universeller Service-Manager

```
#!/usr/bin/env bash
set -euo pipefail

AKTION="${1:-status}"
SERVICE="${2:-nginx}"

case "$AKTION" in
    start)
        echo "Starte Dienst $SERVICE..."
        sudo systemctl start "$SERVICE"
        ;;
    stop)
        echo "Stoppe Dienst $SERVICE..."
        sudo systemctl stop "$SERVICE"
        ;;
    restart|reload)
        echo "Lade Dienst $SERVICE neu..."
        sudo systemctl restart "$SERVICE"
        ;;
    status)
        sudo systemctl status "$SERVICE" --no-pager
        ;;
    *)
        echo "Fehler: Unbekannte Aktion '$AKTION'!" >&2
        echo "Nutzung: $0 {start|stop|restart|status} [dienstname]" >&2
        exit 1
        ;;
esac
```

For-Schleifen: Iteration über Listen, Sequenzen & Arrays

Die `for`-Schleife wiederholt einen Codeblock für jedes Element einer Aufzählung:

1. Iteration über Listen und Arrays

```
# Über ein indiziertes Array iterieren
SERVERS=("web01" "web02" "db01" "cache01")

for server in "${SERVERS[@]}"; do
    echo "Prüfe Erreichbarkeit von $server..."
done
```

2. Iteration über Zahlenbereiche mit Brace Expansion

Vermeide den veralteten Subprozess-Aufruf `for i in $(seq 1 10)`. Nutze stattdessen die native **Brace Expansion** der Bash:

```
# Zähle von 1 bis 5
for i in {1..5}; do
    echo "Durchlauf: $i"
done

# Zähle in 5er-Schritten von 0 bis 20 ({start..ende..schrittweite}):
for port in {8080..8090..2}; do
    echo "Teste Port $port..."
done
```

3. C-Style For-Schleifen

Für Zähler und Berechnungen unterstützt die Bash die klassische C-Syntax:

```
for (( i=0; i<10; i++ )); do
    echo "Index: $i"
done
```

4. Iteration über Dateien (Dateinamen-Globbing)

```
# Alle Logdateien in /var/log verarbeiten
for logdatei in /var/log/*.log; do
    # Wichtig: Prüfen, ob die Datei wirklich existiert (falls keine .log-Datei da ist)
    [[ -f "$logdatei" ]] || continue

    echo "Komprimiere: $logdatei..."
    gzip -k "$logdatei"
done
```

While- und Until-Schleifen

- **`while`:** Führt den Codeblock aus, **solange** die Bedingung wahr (Exit-Code `0`) ist.
- **`until`:** Führt den Codeblock aus, **bis** die Bedingung wahr wird (solange sie falsch ist).

Zählerbasierte While-Schleife:

```
zaehler=1
```

```
while [[ $zaehler -le 5 ]]; do
    echo "Schritt $zaehler von 5"
    (( zaehler++ ))
done
```

Praxis-Goldstandard: Dateien sicher zeilenweise einlesen

Um Logdateien, Serverlisten oder CSVs zeilenweise zu verarbeiten, ist die Kombination aus `while` und `read -r` der absolute Best-Practice-Weg:

```
#!/usr/bin/env bash
set -euo pipefail

HOSTS_FILE="/etc/hosts"

# IFS= verhindert das Abschneiden führender Whitespaces
# -r verhindert die Fehlinterpretation von Backslashes
while IFS= read -r zeile || [[ -n "$zeile" ]]; do
    # Leere Zeilen und Kommentarzeilen überspringen:
    [[ -z "$zeile" || "$zeile" =~ ^# ]] && continue

    echo "Eintrag: $zeile"
done < "$HOSTS_FILE"
```

> ⚠ **Anti-Pattern Warnung:** Nutze **niemals** `for line in $(cat file.txt)`. Dieser Befehl zerlegt Zeilen an jedem Leerzeichen in einzelne Wörter (*Word Splitting*) und bricht bei Leerzeilen oder Sonderzeichen zusammen.

Schleifensteuerung: `break` und `continue`

Mit `break` und `continue` greifst du gezielt in den Ablauf von Schleifen ein:

- `continue`**:** Bricht den **aktuellen Durchlauf** vorzeitig ab und springt sofort zur nächsten Iteration.
- `break`**:** Beendet die **gesamte Schleife** sofort.

```
# Beispiel: Suche nach einer Datei und beende die Schleife beim ersten Treffer
for file in /tmp/reports/*; do
    [[ -f "$file" ]] || continue

    if [[ "$file" == *"critical.log"* ]]; then
        echo "Kritische Datei gefunden: $file!"
        break  # Schleife sofort beenden
    fi
done
```

Bei verschachtelten Schleifen kannst du mit `break 2` oder `continue 2` mehrere Schleifenebenen auf einmal verlassen.

Interaktive CLI-Menüs mit `select`

Die Bash bietet mit dem Konstrukt `select` eine eingebaute Möglichkeit, interaktive textbasierte Auswahlmenüs ohne externe Bibliotheken zu erstellen:

```
#!/usr/bin/env bash
set -euo pipefail

PS3="Bitte eine Option wählen (1-4): "
options=("System-Status" "Speicherplatz anzeigen" "Nginx neu starten" "Beenden")

select opt in "${options[@]}"; do
```

```
    case "$opt" in
        "System-Status")
            uptime
            ;;
        "Speicherplatz anzeigen")
            df -h /
            ;;
        "Nginx neu starten")
            sudo systemctl restart nginx
            echo "Nginx wurde neu gestartet."
            ;;
        "Beenden")
            echo "Auf Wiedersehen!"
            break
            ;;
        *)
            echo "Ungültige Eingabe! Bitte Zahl zwischen 1 und 4 eingeben."
            ;;
    esac
done
```

Übungen & Praxis-Check

> **Praxisaufgaben zur Lektion #3:**
>
> 1. **Aufgabe 1 (IP-Adressen-Validator):**
>
> Schreibe ein Skript `check_ip.sh`, das eine IP-Adresse als `$1` entgegennimmt und mit `[[ "$1" =~ ... ]]` prüft, ob es sich um ein gültiges IPv4-Muster handelt.
>
> 1. **Aufgabe 2 (Port-Warteschleife mit Until):**
>
> Erstelle ein Skript, das mit `until nc -z 127.0.0.1 80; do sleep 1; done` wartet, bis ein lokaler Webserver auf Port 80 antwortet, und nach erfolgreicher Verbindung eine Erfolgsmeldung ausgibt.
>
> 1. **Aufgabe 3 (Logdatei-Filter):**

> Schreibe ein Skript, das `/var/log/syslog` (oder `/var/log/messages`) zeilenweise mit `while IFS= read -r line` einliest und nur Zeilen ausgibt, die das Wort `error` oder `failed` (Groß-/Kleinschreibung ignorieren) enthalten.

Musterlösung zu Aufgabe 1:

```
#!/usr/bin/env bash
set -euo pipefail
IFS=$'\n\t'

IP="${1:?Bitte eine IP-Adresse als Parameter übergeben!}"

# Regex für 4 Zahlenblöcke von 0-255
REGEX='^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$'

if [[ "$IP" =~ $REGEX ]]; then
    echo "OK: '$IP' ist eine formal gültige IPv4-Adresse."
    exit 0
else
    echo "FEHLER: '$IP' ist keine gültige IPv4-Adresse!" >&2
    exit 1
fi
```

Befehlsreferenz (Cheatsheet)

Konstrukt / Operator	Funktion & Beschreibung
`if [[ bedingung ]]; then ... fi`	Standard-Verzweigung in der Bash
`elif [[ bedingung ]]; then`	Zusätzliche Bedingung im Verzweigungsbaum

Konstrukt / Operator	Funktion & Beschreibung
`case "$var" in muster) ... ;; esac`	Mehrfachverzweigung mit Musterabgleich
`[[ -f $file ]]`	Wahr, wenn reguläre Datei existiert
`[[ -d $dir ]]`	Wahr, wenn Verzeichnis existiert
`[[ -s $file ]]`	Wahr, wenn Datei existiert und nicht leer ist (> 0 Bytes)
`[[ -z $str ]]`	Wahr, wenn String die Länge 0 hat (leer ist)
`[[ -n $str ]]`	Wahr, wenn String nicht leer ist
`[[ $str =~ regex ]]`	Prüft String gegen einen regulären Ausdruck
`for item in "${arr[@]}"; do ... done`	Iteriert über alle Elemente eines Arrays
`for (( i=0; i<n; i++ )); do`	C-Style Zählschleife
`while IFS= read -r line; do ... done < file`	Sicheres zeilenweises Einlesen von Dateien
`select opt in "${opts[@]}"; do`	Erstellt interaktives CLI-Auswahlmenü
`break` / `continue`	Bricht Schleife ab bzw. überspringt aktuellen Durchlauf

Weiterführende Ressourcen

Ressource	Beschreibung
GNU Bash Conditional Constructs	Offizielles GNU-Handbuch zu Verzweigungen und Tests
GNU Bash Looping Constructs	Offizielle Dokumentation zu For-, While-, Until- und Select-Schleifen
Bash-Grundkurs #2: Variablen in Bash	Der vorherige Teil: Variablen, Scopes und Parameter-Expansion
Linux Befehlszeilenprozessor Leitfaden	Grundlagen zu I/O-Streams, Pipes und Prozessverwaltung
chmod & Dateiberechtigungen Referenz	Berechtigungsbits für Skripte und Verzeichnisse

Fazit

Mit Verzweigungen, robusten Tests über `[[ ... ]]` und Schleifen hast du die Werkzeuge an der Hand, um Skripte von einfachen Befehlsabfolgen in intelligente, fehlertolerante Systemprogramme zu verwandeln. Die Wahl der richtigen Kontrollstruktur – ob kompaktes `case` für Statusabfragen oder `while read -r` zur Dateiverarbeitung – sorgt für lesbaren, wartbaren und sicheren Code.

> **Tipp:** Nutze in modernen Bash-Skripten konsequent den doppelten Test-Operator `[[ ... ]]` statt der veralteten POSIX-Klammern `[ ... ]`. Kombiniere ihn bei Dateiprüfungen stets mit `-f` oder `-d`, bevor du Lese- oder Schreibaktionen startest.

In der nächsten Lektion bringen wir Modularität und Wiederverwendbarkeit in unsere Skripte:
Weiter geht es mit: Bash Grundkurs #4: Funktionen in Bash

Zur Kursübersicht: Alle Lektionen des Bash-Grundkurses

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-kontrollstrukturen-in-bash