Bash-Skripting

ITS-Net-Lin

Sebastian Meisel

3. Januar 2025

1 Einleitung

Bash-Skripte sind ein mächtiges Werkzeug, um wiederkehrende Aufgaben in Linux- und Unix-basierten Systemen zu automatisieren. Für Fachinformatiker Systemintegration (FISI) ist das Erstellen und Verwalten von Skripten ein wesentlicher Bestandteil der täglichen Arbeit. In diesem Dokument werden grundlegende Themen wie Bash-Skripte, Variablen, Schleifen und bedingte Anweisungen behandelt. Zusätzlich werden Best Practices vorgestellt, um sicherzustellen, dass Skripte robust, effizient und sicher sind.

2 Bash-Skripte

Bash-Skripte sind Textdateien, die eine Reihe von Befehlen enthalten, die in einer Shell ausgeführt werden. Sie werden häufig genutzt, um wiederkehrende Aufgaben zu automatisieren, wie z.B. das Einrichten von Benutzern, das Planen von Cronjobs oder das Durchführen von Tests.

```
1 #!/bin/bash
2 echo "Hallo, Welt!"
```

Der `#!/bin/bash`-Shebang[1] ist erforderlich, um das Skript im richtigen Kontext auszuführen. Ohne den Shebang wird das Skript mit der aktuellen Shell ausgeführt, was zu unerwarteten Ergebnissen führen kann.

3 Variablen

Variablen sind benannte Speicherorte, die Werte halten. In Bash werden Variablen ohne Typdeklaration erstellt und können verschiedene Datentypen (z.B. Zahlen, Strings) aufnehmen. Variablen sollten stets mit einem Namen versehen werden, der aus Buchstaben, Zahlen und Unterstrichen bestehen kann.

```
1 NAME="Max"
2 echo "Hallo, ${NAME}!"
```

Best Practice Verwenden Sie immer doppelte Anführungszeichen um Variablen, um Probleme mit Leerzeichen oder speziellen Zeichen zu vermeiden.

Best Practice Schließen Sie Variablen immer in geschweiften Klammern, um den Variablennamen klar von anderen Variablennamen abzugrenzen.

Best Practice Variablen sollten in Großbuchstaben benannt werden (z.B. $\text{USER}_{\text{NAME}}$), um sie von anderen Elementen wie Befehlen oder Systemvariablen zu unterscheiden.

[1] # wird im englischen Slang teilweise als "she", ! als "bang". Daher kommt Shebang als Bezeichnung für die erste Zeile von Skripten. Diese Zeile nennt das Programm, das das Skript ausführen soll.

4 Tests

In Bash sind Tests ein unverzichtbares Werkzeug, um Bedingungen zu prüfen und die Ausführung von Skripten basierend auf diesen Prüfungen zu steuern. Es gibt zwei Hauptarten von Testausdrücken: `[ … ]` (die klassische Testnotation) und `[[ … ]]` (die erweiterte Testnotation). Während beide Syntaxen zum Testen von Bedingungen verwendet werden können, bietet `[[ … ]]` erweiterte Funktionen und ist robuster. Daher wird empfohlen, die erweiterte Notation zu verwenden.

4.1 Test mit […] (klassische Syntax)

Die klassische Syntax `[ … ]` wird in älteren Bash-Skripten häufig verwendet und ist die POSIX-kompatible Methode. Sie prüft einfache Bedingungen wie die Existenz von Dateien oder Verzeichnissen, Vergleiche von Strings oder Zahlen.

```
if [ -d "/home/max" ]; then
    echo "Verzeichnis existiert"
fi
```

In diesem Beispiel wird überprüft, ob das Verzeichnis `/home/max` existiert.

4.2 Erweiterte Test-Syntax mit [[…]]

Die erweiterte Test-Syntax `[[ … ]]` bietet viele Vorteile im Vergleich zu `[ … ]` Sie ist flexibler und erlaubt komplexere Bedingungen, wie die Verwendung von logischen Operatoren (&& und ||), die Unterstützung von regulären Ausdrücken und die Handhabung von Leerzeichen und speziellen Zeichen in Variablen. Es wird dringend empfohlen, `[[ … ]]` anstelle von `[ … ]` zu verwenden, um diese zusätzlichen Funktionen zu nutzen und die Lesbarkeit und Robustheit des Codes zu verbessern.

```
if [[ -d "/home/max" && -w "/home/max" ]]; then
    echo "Verzeichnis existiert und ist beschreibbar"
fi
```

4.2.1 Vorteile von [[…]] im Vergleich zu […]

Unterstützung für reguläre Ausdrücke: Mit `[[ … ]]` können Sie reguläre Ausdrücke verwenden, um den Inhalt von Variablen oder Strings zu überprüfen.

```
    STRING="abc123"
    if [[ $STRING =~ ^abc ]]; then
        echo "String beginnt mit 'abc'"
    fi
```

Bessere Handhabung von Leerzeichen und Sonderzeichen: `[[ … ]]` behandelt Leerzeichen und spezielle Zeichen (wie `*`, `?`, und `[`) korrekt, ohne dass Anführungszeichen erforderlich sind, um sie zu escapen.

4.3 Typische Tests

Bash bietet eine Vielzahl von Tests, die in Bedingungen verwendet werden können. Diese Tests überprüfen verschiedene Eigenschaften von Dateien, Verzeichnissen und Variablen sowie numerische und stringbasierte Vergleiche.

4.3.1 Tests für Dateitypen

-d prüft, ob ein Verzeichnis existiert.

```
if [[ -d "/home/max" ]]; then
    echo "Verzeichnis existiert"
fi
```

-f prüft, ob eine reguläre Datei existiert.

```
if [[ -f "/etc/passwd" ]]; then
    echo "Die Datei /etc/passwd existiert"
fi
```

-e prüft, ob eine Datei (unabhängig vom Typ) existiert.

```
    if [[ -e "/home/max/file.txt" ]]; then
        echo "Datei existiert"
    fi
```

4.3.2 Numerische Vergleiche

-eq prüft, ob zwei Zahlen gleich sind.

```
if [[ $zahl1 -eq $zahl2 ]]; then
    echo "Die Zahlen sind gleich"
fi
```

-ne prüft, ob zwei Zahlen ungleich sind.

```
if [[ $zahl1 -ne $zahl2 ]]; then
    echo "Die Zahlen sind ungleich"
fi
```

-lt prüft, ob eine Zahl kleiner ist als eine andere.

```
if [[ $zahl1 -lt $zahl2 ]]; then
    echo "$zahl1 ist kleiner als $zahl2"
fi
```

-gt prüft, ob eine Zahl größer ist als eine andere.

```
if [[ $zahl1 -gt $zahl2 ]]; then
    echo "$zahl1 ist größer als $zahl2"
fi
```

-ge prüft, ob eine Zahl größer oder gleich einer anderen ist.

```
if [[ $zahl1 -ge $zahl2 ]]; then
    echo "$zahl1 ist größer oder gleich $zahl2"
fi
```

-le prüft, ob eine Zahl kleiner oder gleich einer anderen ist.

```
if [[ $zahl1 -le $zahl2 ]]; then
        echo "$zahl1 ist kleiner oder gleich $zahl2"
fi
```

3.3. String Vergleiche

== prüft, ob zwei Strings gleich sind.

```
if [[ "$string1" == "$string2" ]]; then
    echo "Die Strings sind gleich"
fi
```

!= prüft, ob zwei Strings ungleich sind.

```
if [[ "$string1" != "$string2" ]]; then
    echo "Die Strings sind ungleich"
fi
```

-z prüft, ob ein String leer ist.

```
if [[ -z "$string1" ]]; then
    echo "Der String ist leer"
fi
```

-n prüft, ob ein String nicht leer ist.

```
if [[ -n "$string1" ]]; then
   echo "Der String ist nicht leer"
fi
```

4.4 Logische Operatoren in Tests

Mit `[[ … ]]` können Sie logische Operatoren verwenden, um mehrere Bedingungen zu kombinieren.

AND (&&) Beide Bedingungen müssen wahr sein.

```
if [[ -f "/etc/passwd" && -r "/etc/passwd" ]]; then
    echo "Die Datei /etc/passwd existiert und ist lesbar"
fi
```

OR (||) Eine der Bedingungen muss wahr sein.

```
if [[ -f "/etc/passwd" || -f "/etc/group" ]]; then
    echo "Eine der Dateien existiert"
fi
```

- NOT (!): Die Bedingung muss falsch sein.

```
if [[ ! -d "/home/max" ]]; then
    echo "Das Verzeichnis /home/max existiert nicht"
fi
```

4.5 Subshells

In Bash können Variablen in Subshells (also in einem neuen Shell-Prozess) weitergegeben werden. Dies hat zur Folge, dass Änderungen an Variablen innerhalb einer Subshell nur lokal sind und die Variablen der übergeordneten Shell nicht beeinflussen. Eine Subshell wird durch das Ausführen von Befehlen in Klammern (…) erstellt.

```
#!/bin/bash
VAR="Hallo"
(
    VAR="Welt"  # Diese Änderung gilt nur in der Subshell
    echo "Innerhalb der Subshell: $VAR"
)
echo "Außerhalb der Subshell: $VAR"  # VAR bleibt unverändert
```

Die Änderung von VAR in der Subshell hat keinen Einfluss auf die Variable in der übergeordneten Shell. Nach der Subshell-Ausführung bleibt VAR="Hallo" in der übergeordneten Shell bestehen.

4.5.1 Das `export`-Kommando

Um eine Variable für nachfolgende Prozesse und Subshells verfügbar zu machen, muss sie mit dem `export`-Befehl exportiert werden. Variablen, die mit `export` versehen sind, sind für alle untergeordneten Prozesse und Subshells zugänglich.
Beispiel:

```
#!/bin/bash
VAR="Welt"
export VAR  # Die Variable VAR ist jetzt auch in Subshells und Prozessen verfügbar
(
    echo "Innerhalb der Subshell: $VAR"
)
```

Durch das `export` wird die Variable VAR für alle Prozesse und Subshells verfügbar gemacht. In der Subshell wird der Wert von VAR korrekt angezeigt, da sie geerbt wurde.

Best Practice Verwenden Sie export nur dann, wenn Sie eine Variable in Subshells oder neuen Prozessen benötigen. Wenn Sie nur in der aktuellen Shell arbeiten, ist es besser, die Variable ohne `export` zu setzen.

Best Practice Achten Sie darauf, Subshells möglichst zu vermeiden, wenn Sie Variablen ändern möchten, die auch außerhalb der Subshell verfügbar sein sollen. Falls nötig, verwenden Sie `export`, um Variablen an nachfolgende Prozesse weiterzugeben.

Best Practice Sehen Sie von der übermäßigen Nutzung von `export` ab, um ungewollte Nebeneffekte in größeren Skripten zu vermeiden.

Zusätzlich zu den grundlegenden Best Practices für Variablen und den typischen Verwendungsszenarien, haben wir hier auch den Umgang mit Subshells und den `export`-Befehl behandelt, um die Auswirkungen auf die Sichtbarkeit und Verfügbarkeit von Variablen zu verdeutlichen.

4.6 Schleifen

Schleifen ermöglichen es, eine Reihe von Befehlen mehrfach auszuführen. Sie sind besonders nützlich für Aufgaben, die sich wiederholen, wie z.B. das Durchlaufen einer Liste von Dateien oder Benutzern.
Beispiel einer for-Schleife:

```
for i in {1..5}
do
    echo "Zahl: $i"
done
```

4.6.1 Arten von Schleifen in Bash:

for Iteriert über eine Liste von Elementen oder über einen Bereich von Zahlen.

while Wiederholt einen Block von Befehlen, solange eine Bedingung wahr ist.

until Wiederholt einen Block von Befehlen, bis eine Bedingung wahr ist.

```
count=1
while [ $count -le 5 ]
do
    echo "Zahl: $count"
    ((count++))
done
```

Best Practice Wählen Sie die geeignete Schleifenart basierend auf der spezifischen Aufgabe. `for` eignet sich für eine bekannte Anzahl von Iterationen, `while` für bedingungsbasierte Wiederholungen.

5 `if-else`-Anweisungen

Mit `if-else`-Anweisungen können Sie Entscheidungen treffen, basierend auf Bedingungen. Dies ist eine der grundlegendsten Kontrollstrukturen in Bash.
Beispiel:

```
if [ -d "/home/max" ]; then
    echo "Verzeichnis existiert"
else
    echo "Verzeichnis existiert nicht"
fi
```

Best Practice Vermeiden Sie unnötig tiefe Verschachtelung von `if-else`-Blöcken, um die Lesbarkeit des Codes zu verbessern.

6 Aufgabenstellung Beispiele

Benutzer anlegen: Automatisieren Sie das Erstellen neuer Benutzer in einem System. Das folgende Skript fordert den Benutzer zur Eingabe eines Benutzernamens auf und erstellt diesen:

```
#!/bin/bash
echo "Geben Sie den Benutzernamen ein:"
read USERNAME
useradd $USERNAME
echo "Benutzer $USERNAME wurde erstellt."
```

Cronjobs einrichten: Cronjobs erlauben es, Skripte zu regelmäßigen Zeitpunkten auszuführen. Hier ein Beispiel für einen Cronjob, der ein Skript jeden Tag um 2 Uhr morgens ausführt:

(crontab -l ; echo "0 2 * * * /path/to/script.sh") | crontab -

7 Best Practices

Verwendung von Funktionen Funktionen helfen dabei, Code zu modularisieren und die Wiederverwendbarkeit zu erhöhen. Sie sind besonders nützlich, wenn bestimmte Aufgaben mehrfach im Skript auftreten.

```
function benutzer_anlegen() {
    echo "Geben Sie den Benutzernamen ein:"
    read USERNAME
    useradd $USERNAME
    echo "Benutzer $USERNAME wurde erstellt."
}
benutzer_anlegen
```

Fehlerbehandlung Stellen Sie sicher, dass Sie Fehler nach wichtigen Befehlen prüfen. Bash gibt durch den Rückgabewert $? an, ob der letzte Befehl erfolgreich war. Beispiel:

```
cp /source/file /destination/
if [ $? -ne 0 ]; then
    echo "Fehler beim Kopieren der Datei"
fi
```

Sicherheit Validieren Sie alle Benutzereingaben, um zu verhindern, dass bösartige Befehle (z.B. durch Shell-Injection) ausgeführt werden. Ein Beispiel für sichere Eingaben:

```
read -p "Bitte Benutzernamen eingeben: " username
if [[ ! "$username" =~ ^[a-zA-Z0-9_]+$ ]]; then
    echo "Ungültiger Benutzername!"
    exit 1
fi
```

Kommentare und Dokumentation: Verwenden Sie Kommentare, um wichtige Abschnitte Ihres Codes zu erklären. Dies hilft dabei, den Code verständlicher und wartungsfreundlicher zu machen. Beispiel:

8 Benutzer erstellen

```
useradd $USERNAME
```

Vermeidung von harten Pfaden: Vermeiden Sie absolute Pfadangaben, wenn es möglich ist. Nutzen Sie stattdessen Umgebungsvariablen oder relative Pfade. Beispiel:

9 Vermeidung von harten Pfaden

```
PATH_TO_SCRIPT="$HOME/scripts/myscript.sh"
```

10 Verwendung von `set -e`

Nutzen Sie `set -e` am Anfang eines Skripts, um sicherzustellen, dass das Skript bei einem Fehler sofort beendet wird. Dies hilft, unvorhergesehene Fehler zu vermeiden.

```
#!/bin/bash
set -e
```