

Shell-Scripting

Linux-Kurs der Unix-AG

Benjamin Eberle

7. Juli 2015



Was ist ein Shell-Script?

- ▶ Aneinanderreihung von Befehlen, die ausgeführt werden
- ▶ Bedingte und wiederholende Ausführung möglich
- ▶ Nützlich bei wiederkehrenden Routine-Aufgaben, die aus mehreren Befehlen bestehen
- ▶ Scripte müssen ausführbar sein (x bei Zugriffsrechten)

Hello World!

```
1 #!/bin/sh  
2 echo "HelloWorld!"
```

- ▶ Gibt „Hello World!“ auf der Standard-Ausgabe aus
- ▶ # fängt einen einzeiligen Kommentar an
- ▶ #! (Shebang) gibt an, mit welchem Programm das Script ausgeführt werden soll
- ▶ Shell-Scripte werden von einer Shell (z. B. /bin/sh oder /bin/bash) ausgeführt
- ▶ echo gibt Text zwischen den Anführungszeichen auf der Standard-Ausgabe aus (echo -n ohne folgenden Zeilenumbruch)

/bin/bash **vs.** /bin/sh

- ▶ /bin/bash häufigste unter Linux verwendete Shell
- ▶ hoher Funktionsumfang, relativ groß
- ▶ nicht immer installiert (z. B. nicht in eingebetteten Systemen)
- ▶ /bin/sh stellt einen minimalen Funktionsumfang zur Verfügung
- ▶ immer vorhanden
- ▶ meistens Symlink auf eine andere Shell (z. B. bash, unter Debian/Ubuntu dash)
- ▶ für Scripte nach Möglichkeit /bin/sh verwenden

Variablen

Shellvariablen

- ▶ nur in der aktuellen Shell verfügbar
- ▶ wird mit Variable=Wert definiert
- ▶ mit \$Variable kann auf die Variable zugegriffen werden
- ▶ oder: \${Variable} (nützlich, wenn die Variable in einer Zeichenkette eingebettet ist)
- ▶ löschen mit unset

Umgebungsvariable

- ▶ Weitergabe an Kindprozesse mit export VARIABLE
- ▶ Mit export -n wird aus einer Umgebungsvariable wieder eine Shellvariable

Zeichenketten

- ▶ Zeichenketten werden z. B. als Argument für echo verwendet
- ▶ auch für Dateinamen, Parameter, etc.
- ▶ müssen in Anführungszeichen gesetzt werden wenn sie Leer- oder andere Sonderzeichen enthalten
- ▶ mit doppelten Anführungszeichen werden Variablennamen durch den Inhalt ersetzt

Zeichenketten – Beispiel

```
1 #!/bin/sh
2 nachricht='Hallo Welt!'
3 echo "Die Nachricht lautet: ${nachricht}"
4 echo 'Die Nachricht lautet: ${nachricht}',
```

Ausgabe:

```
1 Die Nachricht lautet: Hallo Welt!
2 Die Nachricht lautet: ${nachricht}
```

Programmparameter

- ▶ Programme/Scripte im aktuellen Verzeichnis werden mit
./programm aufgerufen
- ▶ es können Parameter angegeben werden:
./programm par1 par2
- ▶ auf diese kann im Script mit \$1, \$2 zugegriffen werden
- ▶ \$* enthält alle Parameter
- ▶ \$# enthält die Anzahl der übergebenen Parameter
- ▶ \$0 enthält den Programm-/Script-Namen

Exit-Status

- ▶ Programme können mit oder ohne Fehler beendet werden
- ▶ Kein Fehler bedeutet Rückgabewert 0
- ▶ „Fehler“ erzeugen einen Rückgabewert größer 0
- ▶ Kann mit \$? ausgelesen werden
- ▶ Kann durch exit Wert gesetzt werden (Bsp. exit 33)
- ▶ Code hinter && wird nur ausgeführt, wenn der erste Befehl „erfolgreich“ war
 - ▶ z.B. rm foo && echo "geloescht!"
- ▶ Code hinter || nur, wenn der erste Befehl nicht „erfolgreich“ war
 - ▶ z.B. rm verz/ || echo "geht nicht"

test

test überprüft eine Bedingung, Kurzschreibweise mit []

- ▶ -e DATEI (exists): wahr, wenn die DATEI existiert
- ▶ -d VERZ (directory): wahr, wenn die VERZ ein Verzeichnis ist
- ▶ s1 = s2: wahr, wenn die Zeichenkette s1 identisch mit s2 ist
- ▶ s1 != s2: wahr, wenn sie nicht identisch sind
- ▶ n1 -eq n2 (equal): wahr, wenn n1 und n2 gleich sind
- ▶ n1 -ne n2 (not equal): wahr, wenn n1 und n2 nicht gleich sind
- ▶ n1 -gt n2 (greater then): wahr, wenn n1 größer als n2 ist
- ▶ n1 -ge n2 (greater equal): wahr, wenn n1 größer gleich n2 ist
- ▶ n1 -lt n2 (lower then): wahr, wenn n1 kleiner als n2 ist
- ▶ n1 -le n2 (lower equal): wahr, wenn n1 kleiner gleich n2 ist

Beispiele

- ▶ `test -e foo`: überprüft ob die Datei `foo` im aktuellen Verzeichnis existiert
- ▶ `[-e foo]`: macht dasselbe
- ▶ `[abc = def]`: überprüft ob der Text „abc“ mit „def“ übereinstimmt
- ▶ `[23 -lt 42]`: überprüft ob 23 kleiner als 42 ist
- ▶ `[! ...]`: kehrt die Bedingung um
- ▶ Ergebnis der Überprüfung findet sich im Exit-Status und kann mit `echo $?` abgefragt werden

Wichtig: Leerzeichen nach der öffnenden und vor der schließenden eckigen Klammer

if-Bedingung

```
1 if Bedingung1; then
2     Befehl1.1
3     Befehl1.2
4 elif Bedingung2; then # elif ist optional
5     Befehl2.1
6     Befehl2.2
7 else # else ist optional
8     Befehl3.1
9     Befehl3.2
10 fi
```

- ▶ wenn Bedingung1 erfüllt ist, werden die Befehle 1.1 und 1.2 ausgeführt
- ▶ falls Bedingung1 nicht erfüllt ist, aber Bedingung2, dann werden die Befehle 2.1 und 2.2 ausgeführt
- ▶ in allen anderen Fällen die Befehle 3.1 und 3.2

Beispiele

```
1 #!/bin/sh
2 if [ $# -eq 2 ]; then # Anz. Parameter
3     if [ $1 -gt $2 ]; then # $1>$2
4         echo $1 "ist groesser als" $2
5     elif [ $1 -lt $2 ]; then # $2>$1
6         echo $1 "ist kleiner als" $2
7     else # ansonsten $1==$2
8         echo $1 "und" $2 "sind gleichgross"
9     fi
10 else # nicht genug Parameter
11     echo "Es muessen 2 Parameter uebergeben werden!"
12 fi
```

for-Schleifen

```
1 for i in XXX; do  
2     Befehl  
3 done
```

- ▶ XXX kann eine Liste oder ein Befehl sein
- ▶ Befehle müssen `befehl` oder \$(befehl) geschrieben werden
- ▶ `: Backtick (Akzent Gravis), links neben Backspace
- ▶ for i in \$* geht alle Parameter durch

Beispiele

```
1 #!/bin/sh
2 liste='1 2 3' # Hochkomma, auf der #-Taste
3 for i in $liste; do
4     echo $i
5 done
```

```
1 #!/bin/sh
2 for i in `seq 1 3`; do # Gravis, neben der ?-Taste
3     echo $i
4 done
```

```
1 #!/bin/sh
2 for i in $(seq 1 3); do
3     echo $i
4 done
```

nützliche Dinge

- ▶ read wartet auf eine Eingabe ins Terminal. Wird mit Enter bestätigt
- ▶ read -s liest die Eingabe ohne sie anzuzeigen
- ▶ sleep 10 wartet 10 Sekunden ab
- ▶ \$((Ausdruck)) wertet den Ausdruck arithmetisch aus
- ▶ x=\$((x+1)) erhöht x um eins
- ▶ lange Befehle können mit \ auf mehrere Zeilen aufgeteilt werden:

```
1 mv verzeichnis/mit/vielen/unterverzeichnissen/abc \
2   in/ein/anderes/verzeichnis/
```