ADMIN:DOCS

Linux Administration #5: Shell-Scripting und Automatisierung

Lerne Shell-Scripting und Automatisierung unter Linux: Von Grundlagen der Shell über Scripting-Konzepte bis hin zur praktischen Automatisierung von Systemaufgaben.

Kategorie: Linux-Administration **Veröffentlicht:** 02.11.2024 **Zuletzt geprüft:** 02.11.2026

Willkommen zum fünften Teil unserer technischen Wiki-Serie über Linux-Administration!

Nachdem wir in den vorherigen Artikeln die Grundlagen der Systemadministration, die Benutzerverwaltung, die Prozessverwaltung und die Netzwerkkonfiguration behandelt haben, widmen wir uns heute dem Shell-Scripting und der Automatisierung von Aufgaben.

Als Linux-Administrator verbringst du viel Zeit in der Shell. Die Fähigkeit, Aufgaben zu automatisieren, ist dabei ein wichtiger Schlüssel zur effizienten Systemverwaltung. Stell dir Shell-Scripting wie einen persönlichen Assistenten vor, der wiederkehrende Aufgaben für dich zuverlässig erledigt.

> ⚠ **Hinweis:** In diesem Artikel verwenden wir Ubuntu/Debian als Beispiel-Distribution. Die grundlegenden Konzepte des POSIX- und Bash-Scriptings sind auf allen Linux-Systemen identisch; Paketmanagement und Pfade zu Konfigurationsdateien können sich je nach Distribution leicht unterscheiden.

Shell-Grundlagen verstehen

Die Shell ist deine primäre Befehlsschnittstelle zum Linux-Betriebssystem:

Quelle: https://admindocs.de/de/linux-administration/linux-administration-shell-scripting-und-automatisierung

- **GNU Bash (Bourne Again Shell):** Der de facto Standard auf fast allen Linux-Servern. Mächtig, abwärtskompatibel zu `sh` und reich an modernen Programmierfeatures (Arrays, Parameter-Expansion, Prozess-Substitution).
- **Zsh (Z-Shell):** Beliebt im interaktiven Desktop- und macOS-Umfeld mit starker Autovervollständigung und Plugin-Ökosystem.
- **Dash / POSIX sh:** Extrem schlanke, ressourcenschonende Shells (oft unter `/bin/sh` verlinkt), die vor allem für schnelle System-Boot-Skripte genutzt werden.

Shell-Umgebung und Variablen

```
# Aktive Shell und Bash-Version ermitteln
echo "Shell: $SHELL"
echo "Bash-Version: $BASH_VERSION"

# Wichtige Systemumgebungsvariablen abfragen
echo "Suchpfad für ausführbare Binaries: $PATH"
echo "Heimatverzeichnis des Benutzers: $HOME"
echo "Aktueller Benutzer: $USER"
```

Shell-Konfiguration und Initialisierung

Die Shell liest beim Start verschiedene Konfigurationsdateien ein:

- `~/.bashrc`: Wird bei jeder interaktiven Non-Login-Shell ausgeführt (Aliase, Funktionen, Prompt).
- `~/.bash_profile` bzw. `~/.profile`: Wird einmalig beim Login (z. B. via SSH) geladen.
- `~/.bash_history`: Speichert den chronologischen Befehlsverlauf.

Quelle: https://admindocs.de/de/linux-administration/linux-administration-shell-scripting-und-automatisierung

```
# Eigene Aliase in ~/.bashrc definieren
alias ll='ls -lah --color=auto'
alias update='sudo apt update && sudo apt upgrade -y'
```

Grundlegende Shell-Befehle und Ein-/Ausgabe

```
# Textausgabe auf stdout
echo "Systemstatus:"

# Verzeichniswechsel und Pfadanzeige
cd /var/log && pwd

# Verzeichnisinhalt detailliert auflisten
ls -lh

# Dateien und Verzeichnisse anlegen, kopieren und entfernen
mkdir -p /tmp/backup
cp /etc/hosts /tmp/backup/
mv /tmp/backup/hosts /tmp/backup/hosts.bak
rm /tmp/backup/hosts.bak
```

Shell-Scripting

Nach den Shell-Grundlagen kommen wir nun zu den Details des Shell-Scriptings. Hier lernen wir, wie wir effektive Skripte schreiben:

Grundlegende Syntax

Ein modernes Bash-Skript beginnt mit dem portablen Shebang und soliden Grundeinstellungen:

```
#!/usr/bin/env bash
# Shebang - ruft Bash portabel über das Environment auf

# Variablen definieren
name="Anna"              # String
alter=25                 # Integer
datum=$(date +%Y-%m-%d)  # Kommando-Substitution

# Ausgaben formatieren
echo "Hallo $name, heute ist der $datum"       # Variablen-Expansion in doppelten Anführungszeichen
echo 'Keine Expansion: $name'                  # Wörtliche Ausgabe in einfachen Anführungszeichen
```

> **Enterprise Best Practice: Der Standard-Header für robuste Bash-Skripte**
>
> Produktionsreife Skripte sollten immer mit folgender Sicherheitskonfiguration starten:
>
> ```
> #!/usr/bin/env bash
> set -Eeuo pipefail
> ```
>
> - **`-e` (`errexit`):** Bricht das Skript sofort ab, wenn ein Befehl mit einem Exit-Code ungleich 0 fehlschlägt.
> - **`-u` (`nounset`):** Behandelt ungesetzte Variablen als fatalen Fehler – verhindert katastrophale Kommandos wie `rm -rf "$BACKUP_DIR/*"`, falls `BACKUP_DIR` versehentlich leer ist.
> - **`-o pipefail`:** Eine Pipeline schlägt fehl, wenn *irgendein* beteiligter Befehl fehlschlägt (nicht nur der letzte).

- **`-E` (`errtrace`):** Vererbt den `ERR`-Trap an Funktionen, Subshells und Kommandosubstitutionen.

Variablen und Datentypen

```
# String-Operationen und Parameter-Expansion
text="Hallo Welt"
echo "Länge: ${#text}"                # Länge: 10
echo "Substring: ${text:0:5}"         # Substring: "Hallo"
echo "Ersetzung: ${text/Welt/Linux}"  # Ersetzung: "Hallo Linux"

# Indizierte Arrays
declare -a fruits=("Apfel" "Birne" "Orange")
echo "Erstes Element: ${fruits[0]}"
echo "Alle Elemente: ${fruits[@]}"
echo "Array-Länge: ${#fruits[@]}"
fruits+=("Banane")                    # Element anhängen

# Arithmetische Operationen
declare -i zahl=42
((zahl+=5))                           # Arithmetische Zuweisung (47)
echo "Direkte Berechnung: $((16 / 4))"
```

Kontrollstrukturen

1. If-Bedingungen (Modern mit `[[ ... ]]`)

```
# Dateiprüfung und numerische Vergleiche
ziel="/etc/hosts"

if [[ -f "$ziel" ]]; then
```

```
    echo "Datei $ziel existiert"
elif [[ -d "$ziel" ]]; then
    echo "$ziel ist ein Verzeichnis"
else
    echo "$ziel nicht gefunden"
fi

# Numerische Bedingungen
alter=25
if (( alter >= 18 && alter < 67 )); then
    echo "Person ist im erwerbsfähigen Alter"
fi
```

2. Schleifen

```
# For-Schleife über eine Liste
for benutzer in anna bob charlie; do
    echo "Richte Verzeichnis für $benutzer ein"
    mkdir -p "/home/$benutzer/dokumente"
done

# For-Schleife mit Zahlenbereich
for i in {1..5}; do
    echo "Schritt $i von 5"
done

# While-Schleife mit Zähler
counter=1
while (( counter <= 3 )); do
    echo "Durchlauf: $counter"
    ((counter++))
done
```

3. Case-Anweisungen für Menüs und CLI-Optionen

```
case "${1:-}" in
    start)
        echo "Starte Dienst..."
        systemctl start nginx
        ;;
    stop)
        echo "Stoppe Dienst..."
        systemctl stop nginx
        ;;
    restart)
        echo "Starte Dienst neu..."
        systemctl restart nginx
        ;;
    *)
        echo "Verwendung: $0 {start|stop|restart}" >&2
        exit 1
        ;;
esac
```

Funktionen

Funktionen modularisieren Skripte und halten Variablen über `local` isoliert:

```
# Funktion zur Prüfung von Systembenutzern
check_user() {
    local username="$1"
    local required_uid="${2:-1000}"

    if id "$username" &>/dev/null; then
        local user_uid
        user_uid=$(id -u "$username")
        if (( user_uid >= required_uid )); then
            echo "Benutzer $username existiert mit UID $user_uid"
```

```
            return 0
        fi
    fi

    echo "Benutzer $username erfüllt Kriterien nicht" >&2
    return 1
}

# Aufruf
check_user "anna" 1000
```

Parameter-Verarbeitung

Spezielle Variablen stellen CLI-Argumente und Skript-Zustände bereit:

Parameter	Bedeutung & Funktion
`$0`	Name des ausgeführten Skripts bzw. Aufrufpfad.
`$1`, `$2`, ...	Positionsparameter (Argumente beim Aufruf).
`$#`	Anzahl der übergebenen Argumente.
`"$@"`	Alle Argumente als getrennte Wörter (sicher bei Leerzeichen).
`"$*"`	Alle Argumente als ein einziger zusammenhängender String.
`$?`	Exit-Status des zuletzt ausgeführten Befehls (0 = Erfolg).
`$$`	Prozess-ID (PID) des laufenden Skripts.

Sichere Argumentbehandlung und Standardwerte

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

```
# Argumente prüfen
if (( $# < 1 )); then
    echo "Verwendung: $0 <quelle> [ziel]" >&2
    exit 1
fi

quelle="$1"
# Parameter mit Fallback-Standardwert
ziel="${2:-/backup/default}"

echo "Sichere von $quelle nach $ziel"
```

Fehlerbehandlung und Traps

- **Exit-Codes:** `0` signalisiert Erfolg, Werte von `1` bis `255` kennzeichnen spezifische Fehler.
- **Operatoren:** `&&` führt den Folge befehl nur bei Erfolg aus, `||` fängt Fehler ab.
- **`trap`:** Registriert Handler-Funktionen für Signale (`EXIT`, `ERR`, `SIGINT`).

Temporäre Arbeitsverzeichnisse mit Trap absichern:

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

# Sicheres temporäres Verzeichnis erstellen
TMP_DIR=$(mktemp -d "/tmp/backup_work.XXXXXX")

# Garantiertes Aufräumen beim Beenden oder Abbruch
cleanup() {
    local exit_code=$?
    echo "Räume temporäre Dateien in $TMP_DIR auf..."
    rm -rf "$TMP_DIR"
    exit "$exit_code"
}
trap cleanup EXIT ERR INT TERM
```

```
# Eigentliche Arbeitslogik
echo "Arbeite im Verzeichnis: $TMP_DIR"
touch "$TMP_DIR/daten.tar"
```

Fehlerbehandlung in Funktionen und Skripten

Für robuste Automatisierungsskripte sollten Funktionen klare Exit-Codes zurückgeben und Fehler frühzeitig abfangen:

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

# Funktion mit strukturierter Fehlerprüfung
backup_directory() {
    local source_dir="$1"
    local target_dir="$2"

    if [[ ! -d "$source_dir" ]]; then
        echo "FEHLER: Quellverzeichnis '$source_dir' existiert nicht!" >&2
        return 1
    fi

    if [[ ! -w "$target_dir" ]]; then
        echo "FEHLER: Keine Schreibrechte im Zielverzeichnis '$target_dir'!" >&2
        return 2
    fi

    cp -a "$source_dir"/* "$target_dir"/ || {
        echo "FEHLER: Kopieren fehlgeschlagen!" >&2
        return 3
    }

    return 0
}
```

Logging und statische Code-Analyse

1. Strukturiertes Logging mit dem System-Journal (`logger`):

Statt rohe Textdateien zu füllen, leitet `logger` Ausgaben direkt in das zentrale `journald`-System:

```
# Schreibt mit Tag und Priorität direkt ins System-Journal
logger -t "backup-script" -p user.info "Backup-Vorgang erfolgreich gestartet"
logger -t "backup-script" -p user.err "FEHLER: Speicherplatz reicht nicht aus"

# Abfrage über journalctl
journalctl -t backup-script -n 20
```

2. Statische Code-Analyse mit ShellCheck:

Bevor ein Skript im Produktivbetrieb oder als Root-Timer eingeplant wird, sollte es mit `shellcheck` auf Sicherheits- und Logikfehler geprüft werden:

```
# ShellCheck installieren und Skript prüfen
sudo apt install -y shellcheck
shellcheck mein-skript.sh
```

Automatisierung

Automatisierungsskripte entlasten Administratoren von monotonen Aufgaben wie Backups, Log-Bereinigungen und Statusprüfungen.

System-Wartung: Alte Logs rotieren und Backups anlegen

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

BACKUP_DIR="/backup/daily"
SRC_DATA="/var/www/html"
TIMESTAMP=$(date +'%Y-%m-%d_%H%M')

mkdir -p "$BACKUP_DIR"

# Archiv komprimieren
tar -czf "$BACKUP_DIR/web_${TIMESTAMP}.tar.gz" -C "$SRC_DATA" .

# Backups älter als 14 Tage automatisch entfernen
find "$BACKUP_DIR" -type f -name "web_*.tar.gz" -mtime +14 -delete

logger -t "maintenance" "Tägliches Backup $TIMESTAMP erfolgreich erstellt."
```

Ressourcen-Monitoring mit Benachrichtigung

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

DISK_LIMIT=85
CURRENT_USAGE=$(df -h / | awk 'NR==2 {print $5}' | tr -d '%')

if (( CURRENT_USAGE >= DISK_LIMIT )); then
    MSG="WARNUNG: Root-Dateisystem ist zu ${CURRENT_USAGE}% belegt!"
    logger -t "resource-monitor" -p user.warn "$MSG"
    echo "$MSG"
fi
```

Zeitgesteuerte Aufgaben: Cron vs. systemd-Timers

Wiederkehrende Aufgaben wie Backups, Log-Bereinigungen oder Zertifikatserneuerungen müssen automatisiert zu festen Zeiten ablaufen.

> **Modernes Scheduling: systemd-Timers als Standard** Auf modernen Linux-Systemen (insbesondere Minimal-Installationen von Debian 12/13 und Ubuntu Server) ist der klassische `cron`-Daemon standardmäßig oft gar nicht mehr installiert. **systemd-Timers** (`.timer` in Kombination mit `.service`) bieten entscheidende Vorteile im Serverbetrieb:
>
> - **Integriertes Logging:** stdout und stderr landen lückenlos im System-Journal (`journalctl -u backup.service`) – keine verlorenen Ausgaben mehr.
> - **Verpasste Läufe nachholen:** Mit `Persistent=true` führt systemd Aufgaben sofort beim nächsten Booten aus, falls der Server zur geplanten Zeit ausgeschaltet oder überlastet war.
> - **Zentraler Überblick:** `systemctl list-timers` zeigt alle anstehenden System- und Benutzer-Timer übersichtlich mit der nächsten Ausführungszeit an.
> - **Cgroup-Ressourcenlimits:** Direkte Drosselung von Speicher (`MemoryMax=`) und CPU (`CPUQuota=`) direkt in der Service-Unit.

Praxis-Beispiel: Tägliches Backup als systemd-Timer einrichten

1. Service-Unit anlegen (`/etc/systemd/system/daily-backup.service`):

```
[Unit]
Description=Taeglicher Backup-Lauf
After=network-online.target
Wants=network-online.target

[Service]
```

```
Type=oneshot
ExecStart=/usr/local/bin/system_backup.sh
User=root
```

2. Timer-Unit anlegen (`/etc/systemd/system/daily-backup.timer`):

```
[Unit]
Description=Trigger fuer taegliches Backup

[Timer]
# Jeden Tag um 02:00 Uhr ausfuehren
OnCalendar=*-*-* 02:00:00
# Verpassten Lauf nach Systemstart sofort nachholen
Persistent=true
# Bis zu 5 Minuten zufaelliger Jitter zur Lastverteilung
RandomizedDelaySec=300

[Install]
WantedBy=timers.target
```

3. Timer aktivieren und überwachen:

```
# systemd neu laden und Timer scharf schalten
sudo systemctl daemon-reload
sudo systemctl enable --now daily-backup.timer

# Alle aktiven Timer und naechste Ausfuehrungszeiten auflisten
systemctl list-timers

# Letzte Ausfuehrung und Logs des Dienstes pruefen
journalctl -u daily-backup.service -n 50
```

Klassische Cron-Jobs (Crontab)

Wenn ein traditioneller Cron-Daemon aktiv ist, erfolgt die Konfiguration über `crontab -e`:

```
# ┌──────────── Minute (0 - 59)
# │ ┌──────────── Stunde (0 - 23)
# │ │ ┌──────────── Tag des Monats (1 - 31)
# │ │ │ ┌──────────── Monat (1 - 12)
# │ │ │ │ ┌──────────── Wochentag (0 - 7, 0 und 7 = Sonntag)
# │ │ │ │ │
# * * * * *  auszuführender_befehl
```

```
# Crontab des aktuellen Benutzers bearbeiten
crontab -e

# Typische Cron-Einträge:
# Jeden Tag um 03:30 Uhr nachts
30 3 * * * /usr/local/bin/daily_cleanup.sh

# Jeden Montag um 08:00 Uhr morgens
0 8 * * 1 /usr/local/bin/weekly_report.sh

# Einmal direkt nach dem Systemneustart
@reboot /usr/local/bin/on_boot_check.sh
```

Update-Automatisierung

> **Sicherheitsupdates: `unattended-upgrades` statt eigener Skripte** Für automatische Sicherheitsupdates sollte kein eigenes Bash-Skript mit `apt-get upgrade` in Cron laufen, da

dies Paketmanager-Locks blockieren und bei interaktiven Abfragen hängenbleiben kann. Verwende auf Debian/Ubuntu den offiziellen Standard **`unattended-upgrades`**:

```
sudo apt update && sudo apt install -y unattended-upgrades
sudo dpkg-reconfigure --priority=low unattended-upgrades
```

Konfiguriert wird das Verhalten in `/etc/apt/apt.conf.d/50unattended-upgrades`, inklusive automatischem Neustart bei Bedarf (`Unattended-Upgrade::Automatic-Reboot "true";`).

Übung

In dieser Übung erstellen wir ein praxistaugliches Wartungsskript mit robuster Fehlerbehandlung, sicherem temporären Verzeichnis und strukturierter Protokollierung.

Szenario: Ein tägliches Bereinigungs- und Prüfskript soll das Root-Dateisystem überwachen und Statusmeldungen an das System-Journal senden.

Mögliche Lösung (`/usr/local/bin/server_health.sh`):

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

# Temporäres Arbeitsverzeichnis anlegen
TMP_DIR=$(mktemp -d "/tmp/health_check.XXXXXX")

cleanup() {
    local exit_code=$?
```

```
    rm -rf "$TMP_DIR"
    exit "$exit_code"
}
trap cleanup EXIT ERR INT TERM

DISK_LIMIT=85
USAGE=$(df / | awk 'NR==2 {print $5}' | tr -d '%')

if (( USAGE >= DISK_LIMIT )); then
    logger -t "server_health" -p user.warn "WARNUNG: Festplattenauslastung bei ${USAGE}%"
else
    logger -t "server_health" -p user.info "Systemzustand OK: Festplatte bei ${USAGE}%"
fi
```

Checkliste für produktive Automatisierungsskripte:

- [ ] Portabler Shebang (`#!/usr/bin/env bash`)
- [ ] Strikter Header gesetzt (`set -Eeuo pipefail`)
- [ ] Alle Variablen in Anführungszeichen (`"$VAR"`)
- [ ] Variablen in Funktionen mit `local` deklariert
- [ ] Temporäre Verzeichnisse mit `mktemp -d` und `trap cleanup EXIT` abgesichert
- [ ] Skript mit `shellcheck` auf Fehler geprüft
- [ ] Timer mit `systemctl list-timers` verifiziert

Befehlsreferenz (Cheatsheet)

Für den schnellen Zugriff bei der Skriptentwicklung und Automatisierung fasst die folgende Referenz die wichtigsten Konstrukte und Kommandos zusammen:

Portabler Shebang Standard-Interpreter-Aufruf für Bash-Skripte über das Environment

$ `#!/usr/bin/env bash`

Strikter Sicherheits-Header Bricht bei Fehlern, ungesetzten Variablen und Pipe-Fehlern ab

```
$ set -Eeuo pipefail
```

Signal- und Exit-Handler registrieren Führt Aufräumfunktionen bei Skript-Ende oder Fehlerfall garantiert aus

```
$ trap '<befehl>' EXIT ERR
```

Aktive systemd-Timer auflisten Zeigt alle aktiven systemd-Timer mit Status und nächster Ausführungszeit

```
$ systemctl list-timers
```

Meldungen ins System-Journal schreiben Schreibt Meldungen direkt mit Tag und Priorität in journald

```
$ logger -t <TAG> "<msg>"
```

Skript statisch analysieren (ShellCheck) Prüft Shell-Skripte auf Syntaxfehler, Fallstricke und Best Practices

```
$ shellcheck <script.sh>
```

Skript im Debug-Tracing ausführen Führt Skript mit Ausgabe jedes ausgeführten Befehls und Variablenwerts aus

```
$ bash -x <script.sh>
```

Crontab des Benutzers bearbeiten Öffnet die persönliche Crontab im Standard-Editor

```
$ crontab -e
```

Aktive Cronjobs anzeigen Listet alle konfigurierten Cron-Einträge des Benutzers auf

```
$ crontab -l
```

Sicheres temporäres Verzeichnis erstellen Erzeugt ein geschütztes Verzeichnis unter /tmp/ mit zufälligem Suffix

```
$ mktemp -d "/tmp/task.XXXXXX"
```

Parameter-Fallback verwenden Verwendet Standardwert, falls die Variable ungesetzt oder leer ist

```
$ ${VAR:-fallback}
```

Sortierbaren Zeitstempel generieren Liefert kompakten Zeitstempel für Backup-Archive und Logdateien

```
$ date +'%Y-%m-%d_%H%M'
```

Weiterführende Ressourcen

Die folgenden Leitfäden, Spezifikationen und internen Kursmodule vertiefen das Shell-Scripting und die Systemautomatisierung:

Bash Reference Manual Offizielle GNU Bash-Referenzdokumentation der Free Software Foundation

systemd.timer Documentation Offizielle Dokumentation zu systemd-Timern und Kalender-Ereignissen

ShellCheck Gallery of Bad Code Umfassende Übersicht typischer Shell-Scripting-Fehler und Gegenmaßnahmen

Linux Administration #4: Netzwerk Das vorherige Modul: IP-Routing, Netplan, nmcli, DNS & UFW

Linux Administration #6: Datensicherung Das nachfolgende Modul: Backup-Strategien, rsync, Borg & Recovery

Befehlszeilenprozessor in Linux Fundamentales Wissen zu Shells, I/O-Streams und Pipes

Fazit

Shell-Scripting und Automatisierung bilden das Rückgrat moderner Linux-Administration. Durch den konsequenten Einsatz von Best Practices wie `set -euo pipefail`, modularen Funktionen mit `local`-Variablen, zuverlässiger Fehlerbehandlung mit `trap`, flexibler Parameterverarbeitung und periodischer Ausführung über Cron oder systemd-Timer verwandelst du fehleranfällige manuelle Routineaufgaben in reproduzierbare, fehlerresistente Hintergrundprozesse.

> □□ **Praxis-Tipp:** Starte jedes produktive Bash-Skript ausnahmslos mit `set -euo pipefail` und einer `trap`-Aufräumfunktion für temporäre Dateien unter `/tmp/`. Dadurch verhinderst du, dass unbemerkte Fehler oder ungesetzte Variablen verheerende Folgeschäden auf deinem Produktivserver anrichten.

Im nächsten Modul unseres Administrationskurses widmen wir uns der Absicherung deiner Serverdaten: □□ **Weiter geht es mit:** Linux Administration #6: Datensicherung und Wiederherstellung

□□ **Zur Kursübersicht:** Alle Linux-Administrations-Artikel & Module