ADMIN:DOCS

Bash Grundkurs #6: Fehlerbehandlung und Debugging in Bash

Mache deine Bash-Skripte fehlertolerant und robust: Signalverarbeitung mit trap, Bash Strict Mode, Trap auf ERR, Custom PS4 Debug-Prompts und strukturiertes Syslog-Logging.

Kategorie: Bash-Grundlagen **Veröffentlicht:** 27.10.2024 **Zuletzt geprüft:** 10.11.2024

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

In den bisherigen Modulen haben wir gelernt, wie man Skripte strukturiert, mit Variablen arbeitet, Kontrollstrukturen einsetzt, Funktionen modularisiert und I/O-Datenströme steuert.

In einer perfekten Welt läuft jeder Befehl fehlerfrei durch. In der Realität des Serverbetriebs bricht jedoch die Netzwerkverbindung ab, Festplatten laufen voll, Benutzer drücken mitten im Update `Strg + C` oder Berechtigungen fehlen. Ein Skript, das in solchen Momenten unkontrolliert weiterläuft oder temporäre Dateien als Müll zurücklässt, gefährdet die Systemstabilität.

In dieser Lektion lernst du, wie du deine Skripte gegen Ausfälle absicherst: Wir behandeln **Signal-Traps (`trap`)** für automatisches Aufräumen, den **Bash Strict Mode**, Fehlerabfänge mit **Trap auf `ERR`**, fortgeschrittene **Debugging-Techniken mit `set -x` und `$PS4`** sowie professionelles **Logging via Syslog**.

Signalverarbeitung & Aufräumroutinen mit `trap`

Wenn ein Skript temporäre Verzeichnisse anlegt, Hintergrundprozesse startet oder Locks setzt, müssen diese beim Beenden zuverlässig bereinigt werden – selbst wenn das Skript durch einen

---

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-fehlerbehandlung-und-debugging-in-bash

Fehler oder durch den Benutzer mit `Strg + C` (`SIGINT`) vorzeitig abgebrochen wird.

Der Befehl `trap` fängt Betriebssystem-Signale ab und führt eine definierte Aufräumfunktion (*Cleanup-Handler*) aus:



Die wichtigsten Signale für `trap`:

Signal	Name	Auslöser & Bedeutung
`EXIT`	Pseudo-Signal	Feuert **immer**, wenn das Skript beendet wird (regulär oder via `exit`)
`INT` (2)	`SIGINT`	Tastatur-Abbruch durch den Benutzer via `Strg + C`
`TERM` (15)	`SIGTERM`	Standard-Terminierungssignal (z. B. durch `kill` oder systemd)

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-fehlerbehandlung-und-debugging-in-bash

Signal	Name	Auslöser & Bedeutung
`HUP` (1)	`SIGHUP`	Terminal-Fenster geschlossen oder SSH-Verbindung getrennt
`ERR`	Bash-Signal	Feuert sofort, wenn ein Befehl einen Fehlercode ($\neq 0$) zurückgibt

Praxisbeispiel: Sicheres Arbeiten mit temporären Verzeichnissen

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

# 1. Sicheres temporäres Verzeichnis erstellen
TEMP_DIR=$(mktemp -d /tmp/backup_job.XXXXXX)

# 2. Aufräumfunktion definieren
cleanup() {
    local exit_code=$?
    echo "Führe Aufräumarbeiten durch..."
    # Temporäres Verzeichnis restlos entfernen
    rm -rf "$TEMP_DIR"
    echo "Temporäre Dateien gelöscht. Exit-Code: $exit_code"
    exit "$exit_code"
}

# 3. Trap auf EXIT, INT und TERM registrieren
trap cleanup EXIT INT TERM

# Hauptprogramm
echo "Arbeite im Verzeichnis: $TEMP_DIR"
tar -czf "$TEMP_DIR/data.tar.gz" /etc/hosts
sleep 2

# Egal ob das Skript hier regulär endet oder abgebrochen wird:
# Die Funktion 'cleanup' wird GARANTIERT ausgeführt!
```

Präzise Fehlerlokalisierung: Trap auf `ERR`

Mit `set -e` bricht die Bash bei Fehlern sofort ab. Sie verrät dir jedoch standardmäßig nicht, in welcher Zeile der Fehler aufgetreten ist.

Mit einem **Trap auf** `ERR` und den Bash-Spezialvariablen `$LINENO` und `$BASH_COMMAND` baust du dir einen präzisen Crash-Reporter:

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

error_handler() {
    local exit_code=$?
    local fehler_zeile=$1
    local fehler_befehl=$2

    echo "========================================" >&2
    echo "KRITISCHER FEHLER IM SKRIPT!" >&2
    echo "Fehlerzeile : $fehler_zeile" >&2
    echo "Befehl      : $fehler_befehl" >&2
    echo "Exit-Code   : $exit_code" >&2
    echo "========================================" >&2
    exit "$exit_code"
}

# Trap registrieren: Übergibt Zeilennummer und Befehl an den Handler
trap 'error_handler $LINENO "$BASH_COMMAND"' ERR

echo "Schritt 1: Initialisierung..."
echo "Schritt 2: Versuche ungültigen Befehl..."

# Dieser Befehl schlägt fehl:
ls /pfad/den_es_garantiert_nicht_gibt

echo "Schritt 3: Wird niemals erreicht."
```

Ausgabe beim Absturz:

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-fehlerbehandlung-und-debugging-in-bash

```
Schritt 1: Initialisierung...
Schritt 2: Versuche ungültigen Befehl...
ls: Zugriff auf '/pfad/den_es_garantiert_nicht_gibt' nicht möglich: Datei oder Verzeichnis
nicht gefunden
========================================
KRITISCHER FEHLER IM SKRIPT!
Fehlerzeile : 23
Befehl      : ls /pfad/den_es_garantiert_nicht_gibt
Exit-Code   : 2
========================================
```

Debugging-Techniken: Dem Fehler auf der Spur

Wenn ein Skript unerwartete Ergebnisse liefert, stehen dir mächtige Analysewerkzeuge zur Verfügung:

1. Der Trace-Modus (`-x`) & Feintuning mit `$PS4`

Standardmäßig zeigt `set -x` vor jedem Befehl ein einfaches `+` an. Indem du die Spezialvariable `$PS4` anpasst, blendet die Bash bei jedem Schritt die **Dateinamen, die exakte Zeilennummer und den Funktionsnamen** ein:

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

# Erweiterter Trace-Prompt für professionelles Debugging:
export PS4='+ [${BASH_SOURCE}:${LINENO}] ${FUNCNAME[0]:+${FUNCNAME[0]}(): }'

# Debugging gezielt für einen Block aktivieren:
set -x
```

```
a=10
b=20
summe=$(( a + b ))

set +x # Debugging wieder deaktivieren

echo "Berechnung abgeschlossen: $summe"
```

2. Syntax-Prüfung ohne Ausführung (`bash -n`)

Prüft Skripte rein auf syntaktische Korrektheit (z. B. ungeschlossene Quotes, vergessene `done`- oder `fi`-Keywords), **ohne** gefährliche Befehle auszuführen:

```
bash -n deploy_production.sh
```

3. Statische Analyse mit ShellCheck

```
# Skript auf Best Practices und typische Fallstricke prüfen:
shellcheck deploy_production.sh
```

Strukturiertes Logging: Datei- & Syslog-Integration

Professionelle Administrationsskripte schreiben Statusmeldungen nicht unformatiert mit `echo`, sondern nutzen einheitliche Log-Level (`INFO`, `WARN`, `ERROR`) und integrieren sich in das Linux-System-Journal (*Syslog*):

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

LOG_FILE="/tmp/app_maintenance.log"
APP_NAME="BackupService"

log_msg() {
    local level="$1"
    shift
    local msg="$*"
    local timestamp
    timestamp=$(date '+%Y-%m-%d %H:%M:%S')

    # 1. In Logdatei schreiben
    echo "[$timestamp] [$level] $msg" >> "$LOG_FILE"

    # 2. An System-Journal / Syslog senden (sichtbar via journalctl):
    logger -t "$APP_NAME" "[$level] $msg"

    # 3. Farbig im Terminal anzeigen
    case "$level" in
        INFO)  echo -e "\e[1;32m[INFO]\e[0m  $msg" ;;
        WARN)  echo -e "\e[1;33m[WARN]\e[0m  $msg" >&2 ;;
        ERROR) echo -e "\e[1;31m[ERROR]\e[0m $msg" >&2 ;;
    esac
}

log_msg "INFO" "Wartungsaufgabe gestartet."
log_msg "WARN" "Hohe CPU-Auslastung erkannt (88%)."
log_msg "ERROR" "Konnte Datenbankverbindung nicht aufbauen!"
```

Die mit `logger` gesendeten Nachrichten lassen sich systemweit mit `journalctl -t BackupService` filtern.

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-fehlerbehandlung-und-debugging-in-bash

Übungen & Praxis-Check

> **Praxisaufgaben zur Lektion #6:**
>
> 1. **Aufgabe 1 (Signal-Trap & Lockfile):**
>
> Schreibe ein Skript `single_instance.sh`, das beim Start eine Lockdatei `/tmp/my_app.lock` anlegt und per `trap` sicherstellt, dass diese Datei beim Beenden (egal ob regulär oder via `Strg + C`) gelöscht wird.
>
> 1. **Aufgabe 2 (Crash-Handler mit Alert):**
>
> Implementiere einen `ERR`-Trap, der im Fehlerfall die genaue Zeilennummer in `/tmp/crash.log` speichert und dem Benutzer eine strukturierte Warnung ausgibt.
>
> 1. **Aufgabe 3 (Custom PS4 Debugger):**
>
> Passe `$PS4` so an, dass vor jedem Befehl die aktuelle Uhrzeit im Format `[HH:MM:SS]` angezeigt wird.

Musterlösung zu Aufgabe 1:

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

LOCK_FILE="/tmp/single_instance.lock"

# Prüfen, ob bereits eine Instanz läuft
if [[ -f "$LOCK_FILE" ]]; then
    echo "Fehler: Skript läuft bereits (Lockdatei $LOCK_FILE existiert)!" >&2
    exit 1
fi
```

```
# Lockfile anlegen
touch "$LOCK_FILE"

# Aufräumen garantieren
cleanup() {
    rm -f "$LOCK_FILE"
    echo "Lockdatei freigegeben."
}
trap cleanup EXIT INT TERM

echo "Skript läuft exklusiv. Drücke Strg+C zum Testen des Traps..."
sleep 5
echo "Arbeit erfolgreich abgeschlossen."
```

Befehlsreferenz (Cheatsheet)

Befehl / Syntax	Funktion & Beschreibung
`trap 'cleanup' EXIT INT TERM`	Führt `cleanup` beim Beenden oder bei Signalunterbrechung aus
`trap 'error_func $LINENO' ERR`	Fängt Befehlsfehler ab und übergibt die Fehlernummern-Zeile
`set -euo pipefail`	Aktiviert den vollständigen Bash Strict Mode
`set -x` / `set +x`	Aktiviert / deaktiviert den interaktiven Ausführungs-Trace
`export PS4='+ [${LINENO}] '`	Definiert das Format der Debugging-Trace-Ausgabe
`bash -n skript.sh`	Prüft Skript rein auf Syntaxfehler ohne Ausführung
`logger -t TAG "Text"`	Schreibt eine Logmeldung direkt in das Linux-System-Journal
`$BASH_COMMAND`	Enthält den aktuell fehlgeschlagenen Befehl im `ERR` -Trap

Weiterführende Ressourcen

Ressource	Beschreibung
GNU Bash Trap Builtin Reference	Offizielle GNU-Dokumentation zur Signalbehandlung mit trap
Bash-Grundkurs #5: Ein- und Ausgabe in Bash	Der vorherige Teil: Datenströme, Pipes und Here-Docs
Linux Befehlszeilenprozessor Leitfaden	Fundamentales Wissen zu Shells, I/O-Streams und Pipes
chmod & Dateiberechtigungen	Linux-Berechtigungskonzepte im Detail

Fazit

Robuste Skripte unterscheiden sich von simplen Hobby-Skripten durch vorhersehbare Fehlerbehandlung und sauberes Ressourcen-Management. Durch den gezielten Einsatz von Signal-Traps (`trap`), den Bash Strict Mode und detailliertes Logging schützt du deine Server vor verwaisten Lockfiles, unbereinigtem Datenmüll und unbemerkten Fehlern im Hintergrund.

> □□ **Tipp:** Nutze bei allen Skripten, die temporäre Dateien in `/tmp` oder Lockfiles anlegen, sofort nach der Erstellung einen `trap cleanup EXIT INT TERM`. Das verhindert, dass abgebrochene Cronjobs die Festplatte mit Restmüll füllen.

Im siebten und letzten Teil unserer Reihe bündeln wir unser Wissen und widmen uns den fortgeschrittenen Profi-Techniken: □□ **Weiter geht es mit:** Bash Grundkurs #7: Fortgeschrittene Bash-Techniken

□□ **Zur Kursübersicht:** Alle Lektionen des Bash-Grundkurses