ADMIN:DOCS

Bash Grundkurs #5: Ein- und Ausgabe in Bash

Meistere Datenströme und I/O-Redirection in der Bash: Standard-Deskriptoren (0, 1, 2), Pipelines, Here-Docs, Here-Strings, tee und interaktive Eingaben mit read.

Kategorie: Bash-Grundlagen **Veröffentlicht:** 26.10.2024 **Zuletzt geprüft:** 10.11.2024

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

Nachdem wir in den vorherigen Modulen gelernt haben, wie man Skripte anlegt und absichert, mit Variablen arbeitet, Kontrollstrukturen und Schleifen einsetzt und Code in Funktionen modularisiert, widmen wir uns heute der Lebensader jedes Linux-Systems: **den Ein- und Ausgabe-Datenströmen (I/O)**.

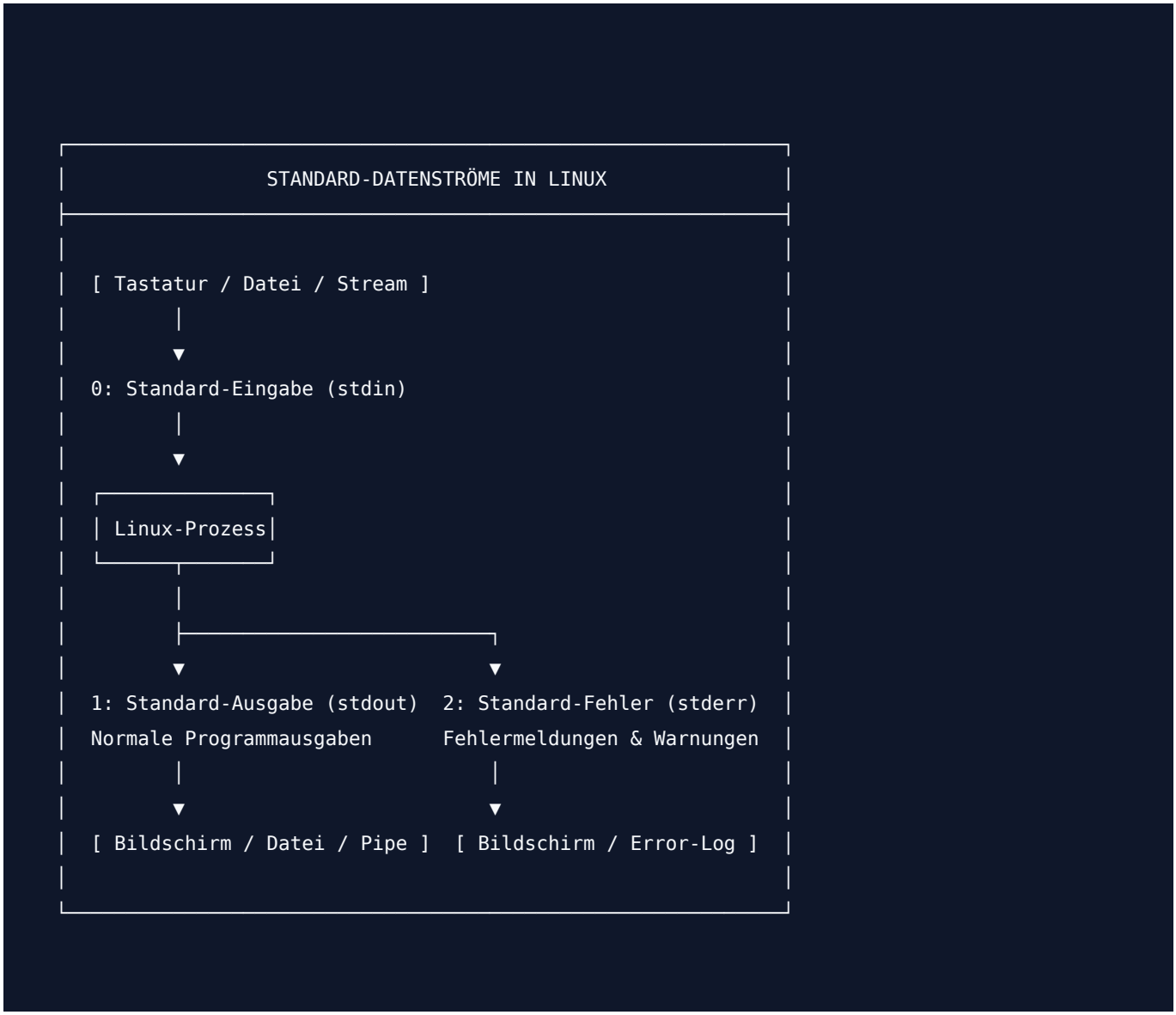
Das fundamentale Unix-Prinzip lautet: *„Everything is a stream / file“* und *„Write programs that do one thing and do it well. Write programs to work together“*. Ein Skript, das Daten nicht flexibel aufnehmen, verarbeiten, umleiten und dauerhaft speichern kann, ist für die praktische Systemadministration nutzlos.

In dieser Lektion lernst du, wie Linux Datenströme über **Dateideskriptoren (0, 1, 2)** verwaltet, wie du Standardausgaben und Fehlermeldungen gezielt trennst (`2>` , `2>&1` , `&>`), wie Pipelines (`|`) und `tee` Daten im Speicher verketten, wie du mehrzeilige Konfigurationen über **Here-Documents** erstellst und wie du Benutzereingaben mit `read -r` sicher einliest.

Dateideskriptoren: stdin (0), stdout (1) und stderr (2)

Jeder Prozess, den die Linux-Shell startet, erhält vom Kernel automatisch drei geöffnete Datenkanäle (*File Descriptors*):

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-ein-und-ausgabe-in-bash



Die drei Kanäle im Überblick:

1. `0` – `stdin` (***Standard Input***)**:** Der Eingabestrom, über den Programme Daten von der Tastatur, aus Dateien oder von anderen Programmen empfangen.
2. `1` – `stdout` (***Standard Output***)**:** Der Ausgabestrom für reguläre Programmergebnisse.
3. `2` – `stderr` (***Standard Error***)**:** Der separate Kanal für Fehlermeldungen, Status- und Warnhinweise.

> **Warum zwei getrennte Ausgabekanäle?** Die Trennung von `stdout` (1) und `stderr` (2) stellt sicher, dass ein Programm verwertbare Daten über eine Pipe weiterleiten kann, während Warnungen oder Fehler weiterhin auf dem Bildschirm sichtbar bleiben, anstatt das nachfolgende Programm zu beschädigen.

I/O-Umleitungen: Ausgaben und Fehler steuern

Mit Umleitungsoperatoren lenkst du die Datenströme in Dateien oder Pseudogeräte wie `/dev/null` um:

```
┌──────────────────────────────────────────────────────────┐
│                UMLEITUNGS-MATRIX IM ÜBERBLICK            │
├──────────────────────────────────────────────────────────┤
│                                                          │
│  Operator      Bedeutung & Verhalten                     │
│  ──────────────────────────────────────────────────────  │
│  > datei       Leitet stdout in datei (überschreibt Datei)  │
│  >> datei      Hängt stdout an datei an (Append-Modus)   │
│  2> datei      Leitet NUR stderr (Fehler) in datei um    │
│  2>> datei     Hängt NUR stderr an datei an              │
│  &> datei      Leitet stdout UND stderr zusammen in datei   │
│  2>&1          Dupliziert stderr auf den Kanal von stdout   │
│  < datei       Liest stdin aus datei ein                 │
│  >/dev/null 2>&1 Verwirft alle Ausgaben & Fehler restlos │
│                                                          │
└──────────────────────────────────────────────────────────┘
```

Praxisbeispiele für Umleitungen:

```
# 1. Reguläre Ausgabe überschreibend speichern:
echo "Server initialisiert" > /var/log/init.log

# 2. Ausgabe an bestehendes Log anhängen:
echo "[$(date '+%F %T')] Dienst gestartet" >> /var/log/app.log

# 3. Nur Fehlermeldungen in separates Fehler-Log schreiben:
find /var/ -name "*.conf" 2> /tmp/find_errors.log

# 4. Fehlermeldungen komplett stummschalten (/dev/null):
grep -r "SECRET_KEY" /etc/ 2>/dev/null

# 5. stdout und stderr zusammen in dieselbe Logdatei schreiben:
./backup.sh &> /var/log/backup.log
# POSIX-konforme Alternative:
./backup.sh > /var/log/backup.log 2>&1
```

Eigene Fehlermeldungen auf `stderr` ausgeben (`>&2`)

Wenn dein Skript auf einen Fehler stößt, solltest du die Meldung explizit auf `stderr` senden:

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

if [[ ! -f "/etc/app.conf" ]]; then
    # '> &2' leitet die Ausgabe dieses echo-Befehls auf stderr (Kanal 2) um:
    echo "KRITISCHER FEHLER: Konfigurationsdatei /etc/app.conf fehlt!" >&2
    exit 1
fi
```

Pipelines (`|`) und parallele Ausgabe mit `tee`

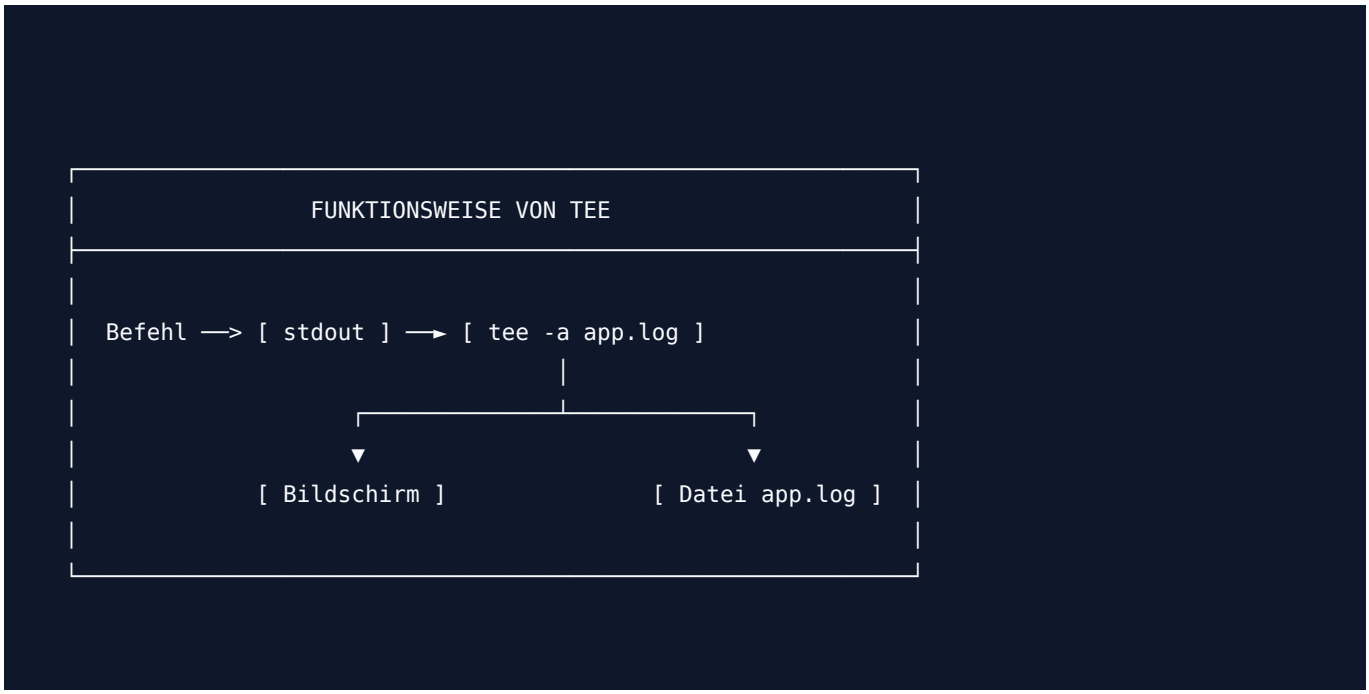
Eine **Pipeline** (`|`) verbindet die Standardausgabe (`stdout`) des linken Befehls direkt mit der Standardeingabe (`stdin`) des rechten Befehls im Arbeitsspeicher – ohne temporäre Zwischendateien.

```
# Alle laufenden Nginx-Prozesse filtern und zählen:
ps aux | grep nginx | grep -v grep | wc -l

# Die 5 größten Dateien im Home-Verzeichnis finden:
du -ah ~/ | sort -rh | head -n 5
```

Das Werkzeug `tee`: T-Stück für Datenströme

Oft möchtest du die Ausgabe eines Befehls **gleichzeitig auf dem Bildschirm sehen UND in eine Datei speichern**. Hierfür nutzt du `tee`:



```
# Deployment ausführen, auf dem Bildschirm verfolgen und im Log archivieren:
./deploy.sh | tee -a /var/log/deploy.log
```

Mehrzeiliger Text: Here-Documents (`<<EOF`) & Here-Strings (`<<<`)

Wenn du mehrzeilige Konfigurationsdateien, SQL-Queries oder E-Mails direkt in deinem Skript generieren möchtest, sind **Here-Documents** das sauberste Werkzeug.

1. Standard Here-Document mit Variablen-Expansion

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

DOMAIN="example.com"
PORT=8080

# Erzeuge eine Nginx-Konfigurationsdatei:
cat <<EOF > /tmp/nginx_${DOMAIN}.conf
server {
    listen 80;
    server_name ${DOMAIN};

    location / {
        proxy_pass http://127.0.0.1:${PORT};
        proxy_set_header Host \$host;
    }
}
EOF
```

> **Variablen-Expansion verhindern (`<<'EOF'`):** Wenn der Text Anführungszeichen, `$`-Zeichen oder Bash-Code enthält, der nicht vom Skript expandiert werden soll, schließe den Begrenzer in Anführungszeichen ein (`<<'EOF'`).

2. Tabs einrücken mit `<<-EOF`

Das Minuszeichen (`<<-EOF`) sorgt dafür, dass führende **Tabulator-Zeichen** (nicht Leerzeichen!) im erzeugten Text automatisch abgeschnitten werden, sodass dein Skriptcode sauber eingerückt bleibt.

3. Here-Strings (`<<<`)

Ein **Here-String** übergibt eine einzelne Variable oder einen kurzen String direkt an den `stdin` eines Programms:

```
IP="192.168.1.1"
# Übergibt $IP direkt an bc oder cut ohne echo-Pipe:
cut -d'.' -f1 <<< "$IP"  # Ausgabe: 192
```

Interaktive Eingaben mit `read -r`

Der Befehl `read` liest Benutzereingaben von der Tastatur oder aus Datenströmen ein.

Wichtige Parameter von `read`:

Option	Funktion	Beispiel
`-r`	**Zwingend:** Verhindert Fehlinterpretation von Backslashes	`read -r text`

Option	Funktion	Beispiel
`-p "Text"`	Zeigt einen Eingabeprompt an	`read -r -p "Name: " name`
`-s`	**Silent Mode:** Versteckt die Eingabe (ideal für Passwörter)	`read -rs -p "Passwort: " pw`
`-t sek`	**Timeout:** Bricht nach x Sekunden automatisch ab	`read -t 10 -p "Weiter? (10s): " a`
`-n anz`	Liest maximal *anz* Zeichen ein (ohne Enter drücken)	`read -n 1 -p "J/N? " answer`
`-a arr`	Liest Wörter direkt in ein indiziertes Array ein	`read -ra worte`

Praxisbeispiel: Sicherer Passwort- und Bestätigungsdialog

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

# 1. Name abfragen
read -r -p "Benutzername eingeben: " USERNAME

# 2. Passwort verdeckt abfragen
read -rs -p "Passwort eingeben: " PASSWORD
echo "" # Zeilenumbruch nach verdeckter Eingabe

# 3. Ein-Tasten-Bestätigung (1 Zeichen ohne Enter)
read -n 1 -r -p "Möchtest du fortfahren? (j/n): " BESTAETIGUNG
echo ""

if [[ "$BESTAETIGUNG" =~ ^[jJyY]$ ]]; then
    echo "Aktion für Benutzer '$USERNAME' wird ausgeführt..."
else
    echo "Vorgang abgebrochen."
    exit 0
```

```
fi
```

Dateien sicher zeilenweise einlesen

Um Logdateien oder CSV-Dateien zeilenweise zu verarbeiten, ist die Kombination aus `while` und `read -r` der absolute Standard:

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

CSV_FILE="/tmp/benutzer.csv"

# Beispiel-Daten erstellen
cat <<'EOF' > "$CSV_FILE"
max;admin;/home/max
anna;developer;/home/anna
lisa;support;/home/lisa
EOF

# Zeilenweise mit benutzerdefiniertem Trennzeichen (Semikolon) einlesen:
while IFS=';' read -r name rolle home_dir || [[ -n "$name" ]]; do
    [[ -z "$name" || "$name" =~ ^# ]] && continue
    echo "Benutzer: $name | Rolle: $rolle | Verzeichnis: $home_dir"
done < "$CSV_FILE"
```

Übungen & Praxis-Check

> **Praxisaufgaben zur Lektion #5:**
>
> 1. **Aufgabe 1 (Strukturiertes Log-Routing):**

> Schreibe ein Skript `log_router.sh`, das normale Ausgaben in `/tmp/output.log` und Fehlermeldungen in `/tmp/error.log` schreibt, während der Benutzer beide Ausgaben farbig im Terminal sieht.
>
> 1. **Aufgabe 2 (Konfigurations-Generator):**
>
> Erstelle ein Skript, das den Benutzer nach `DB_NAME`, `DB_USER` und `DB_PASS` fragt und per Here-Document eine Datei `database.env` mit Rechten `600` (`chmod 600`) generiert.
>
> 1. **Aufgabe 3 (Timeout-Abfrage):**
>
> Schreibe ein Skript, das mit `read -t 5` nach einer Bestätigung fragt und bei Ausbleiben einer Eingabe nach 5 Sekunden mit einem Standardwert automatisch fortfährt.

Musterlösung zu Aufgabe 2:

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

read -r -p "Datenbank-Name : " DB_NAME
read -r -p "Datenbank-User : " DB_USER
read -rs -p "Datenbank-Pass : " DB_PASS
echo ""

ENV_DATEI="/tmp/database.env"

cat <<EOF > "$ENV_DATEI"
# Automatisch generierte Datenbank-Konfiguration
DB_NAME="${DB_NAME}"
DB_USER="${DB_USER}"
DB_PASS="${DB_PASS}"
DB_HOST="127.0.0.1"
DB_PORT="3306"
EOF
```

```
# Berechtigungen auf Nur-Eigentümer einschränken
chmod 600 "$ENV_DATEI"
echo "Konfiguration erfolgreich erstellt unter $ENV_DATEI (Rechte: 600)."

exit 0
```

Befehlsreferenz (Cheatsheet)

Syntax / Befehl	Funktion & Beschreibung
`cmd > file`	Leitet stdout in Datei um (überschreibt Datei)
`cmd >> file`	Hängt stdout an Datei an
`cmd 2> file`	Leitet Fehlermeldungen (stderr) in Datei um
`cmd &> file`	Leitet stdout UND stderr zusammen in Datei um
`echo "msg" >&2`	Gibt Text explizit auf dem Fehlerkanal (stderr) aus
`cmd1	cmd2`	Verbindet stdout von cmd1 direkt mit stdin von cmd2
`cmd	tee -a file`	Schreibt stdout in Datei und zeigt ihn gleichzeitig auf dem Bildschirm
`cat <<EOF > file`	Erstellt Datei aus mehrzeiligem Here-Document
`read -r -p "Prompt" var`	Liest Benutzereingabe mit Anzeige eines Prompt-Textes ein
`read -rs var`	Liest Passwörter verdeckt ohne Bildschirmausgabe ein
`while IFS= read -r line; do`	Sicheres Einlesen von Dateien Zeile für Zeile

Weiterführende Ressourcen

Ressource	Beschreibung

GNU Bash Redirections Manual	Vollständige GNU-Dokumentation zu Datenströmen und Deskriptoren
Bash-Grundkurs #4: Funktionen in Bash	Der vorherige Teil: Modularisierung, Parameter und Scopes
Linux Befehlszeilenprozessor Leitfaden	Fundamentales Wissen zu Shells, I/O-Streams und Pipes
chmod & Dateiberechtigungen	Linux-Berechtigungskonzepte im Detail

Fazit

Die Beherrschung von Ein- und Ausgabeströmen, Deskriptoren und Umleitungen ist der Schlüssel zur nahtlosen Integration deiner Skripte in das Linux-Ökosystem. Durch die gezielte Trennung von `stdout` und `stderr`, den Einsatz von `tee` für lückenloses Logging und robuste Eingabedialoge mit `read -r` machst du deine Automatisierungen transparent und benutzerfreundlich.

> **Tipp:** Sende Fehlermeldungen in Skripten immer explizit nach `stderr` (`echo "Fehler..." >&2`). Das ermöglicht es anderen Admins und CI/CD-Pipelines, Fehler über `2>` sauber in separate Logdateien zu filtern, ohne die regulären Datenströme zu verunreinigen.

In der nächsten Lektion machen wir unsere Skripte unzerstörbar und lernen professionelle Fehlerbehandlung, Signalabfangung (*Traps*) und Debugging: **Weiter geht es mit:** Bash Grundkurs #6: Fehlerbehandlung und Debugging in Bash

Zur Kursübersicht: Alle Lektionen des Bash-Grundkurses