ADMIN:DOCS

Bash Grundkurs #1: Erstelle Dein Erstes Bash-Shell-Skript

Willkommen zum ersten Teil unserer Bash-Grundkurs-Serie! Lerne Schritt für Schritt, wie du dein erstes Bash-Shell-Skript erstellst, ausführst, Berechtigungen vergibst und robuste Skripte schreibst.

Kategorie: Bash-Grundlagen **Veröffentlicht:** 22.10.2024 **Zuletzt geprüft:** 10.11.2024

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

In dieser mehrteiligen Reihe lernst du Schritt für Schritt, wie du deine eigenen Shell-Skripte erstellst, ausführst und zur Automatisierung administrativer Workflows einsetzt. Egal, ob du neu in der Linux-Kommandozeile bist oder deine praktischen Kenntnisse auffrischen und vertiefen möchtest: Dieser Grundkurs vermittelt dir das fundamentale Wissen für sauberes, sicheres und reproduzierbares Skripting.

Was bedeutet Bash?

Die **Bash** (*Bourne Again Shell*) ist die Standard-Shell in den allermeisten Linux-Distributionen und vielen Unix-Umgebungen. Sie fungiert einerseits als interaktiver Befehlszeilenprozessor zur Steuerung des Systems, andererseits als vollwertige, mächtige Skriptsprache. Mit Bash-Skripten kannst du komplexe Abläufe automatisieren, Systemdienste überwachen, Datensicherungen erstellen und deine Produktivität als Administrator oder Entwickler erheblich steigern.

In dieser ersten Lektion konzentrieren wir uns auf die absoluten Grundlagen:

Wir erstellen dein allererstes Skript, analysieren den Shebang-Mechanismus, vergeben Ausführungsberechtigungen, unterscheiden verschiedene Ausführungskontexte und verwandeln ein einfaches Skript in ein robustes Programm mit dem **Bash Strict Mode**.

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-erstelle-dein-erstes-bash-shell-skript

Der Kursüberblick: Deine Lern-Roadmap

Unser Bash-Grundkurs ist modular aufgebaut und führt dich schrittweise vom ersten Befehl bis hin zu professionellen Automatisierungs-Skripten:



> **Voraussetzungen:** Ein funktionierendes Linux-Terminal (z. B. unter Ubuntu, Debian, Arch Linux, Fedora oder openSUSE) und ein einfacher Texteditor wie `nano` oder `vim`.

Schritt 1: Das klassische „Hello World“-Skript erstellen

Lass uns mit dem bewährten „Hello World“-Beispiel starten. Dieses Skript hilft dir, die grundlegende Struktur einer Skriptdatei zu verstehen.

Terminal öffnen und Skriptverzeichnis anlegen

Öffne ein Terminal-Fenster (unter den meisten Desktop-Umgebungen mit der Tastenkombination `Strg + Alt + T`).

Erstelle einen dedizierten Ordner für deine Skripte und wechsle hinein:

```
mkdir -p ~/scripts && cd ~/scripts
```

Datei im Editor anlegen

Öffne eine neue Datei namens `hello_world.sh` im Texteditor `nano`:

```
nano hello_world.sh
```

Gib den folgenden Inhalt ein:

```
#!/bin/bash
echo "Hello, World!"
```

Lass uns diese beiden Zeilen genauer betrachten:

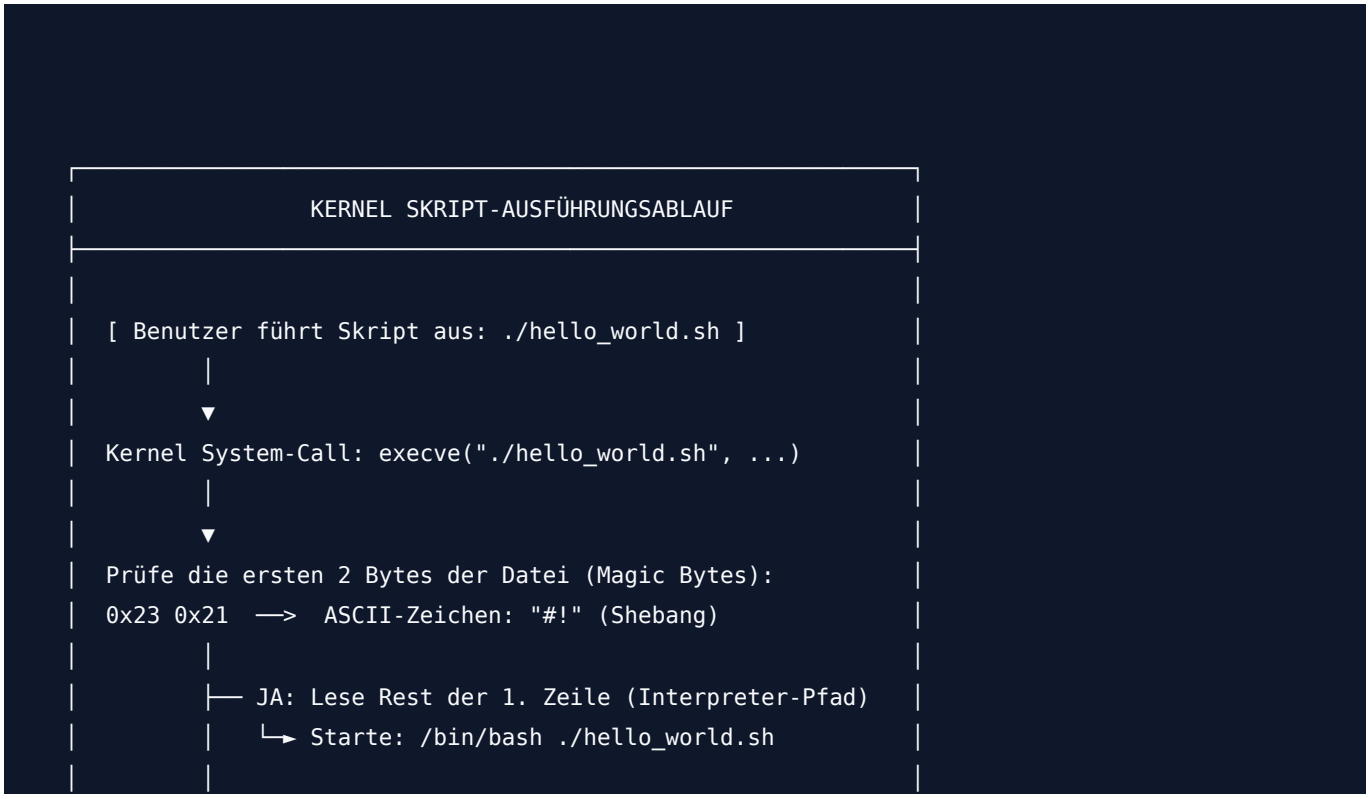
- **Zeile 1 (`#!/bin/bash`):** Wird als **Shebang** bezeichnet. Sie teilt dem Linux-Betriebssystem mit, welcher Interpreter zur Ausführung der Datei verwendet werden soll.
- **Zeile 2 (`echo "Hello, World!"`):** Der Befehl `echo` gibt die angegebene Zeichenkette auf dem Standard-Ausgabestrom (*stdout*) im Terminal aus.

Speichere die Datei in `nano` mit `Strg + O`, bestätige mit `Enter` und verlasse den Editor mit `Strg + X`.

Die Architektur hinter dem Shebang (`#!`)

Wenn du eine Binärdatei aufrufst, lädt der Linux-Kernel den Maschinencode direkt. Ein Skript hingegen besteht aus unformatiertem Text.

Damit das System weiß, welches Programm diesen Text verarbeiten soll, prüft der Kernel beim System-Call `execve()` die ersten beiden Bytes der Datei:



```
│          └── NEIN:                                          │
│              └─► Fallback: Starte mit Standard-System-Shell  │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

Portabler Shebang: `#!/usr/bin/env bash`

Neben dem klassischen Pfad `#!/bin/bash` existiert eine noch flexiblere Variante:

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

Das Hilfswerkzeug `env` sucht die Bash dynamisch anhand der Umgebungsvariable `$PATH` des Benutzers. Das ist besonders nützlich auf Systemen, auf denen die Bash nicht unter `/bin/bash` liegt (wie macOS oder FreeBSD).

Dateiberechtigungen: Das Skript ausführbar machen

Standardmäßig besitzen neu erstellte Textdateien unter Linux keine Ausführungsrechte (*Execute-Bit*). Wenn du versuchst, das Skript direkt zu starten, verweigert das System den Zugriff:

```
./hello_world.sh
# Ausgabe: bash: ./hello_world.sh: Keine Berechtigung (Permission denied)
```

Mit dem Befehl `chmod` fügst du das Ausführungsrecht (`+x`) für die Datei hinzu:

```
chmod +x hello_world.sh
```

Jetzt führst du das Skript aus, indem du seinen Namen mit vorangestelltem `./` eingibst:

```
./hello_world.sh
```

Ausgabe:

```
Hello, World!
```

> **Hinweis zum Pfad `./`:** Aus Sicherheitsgründen durchsucht Linux standardmäßig nur die im `$PATH` hinterlegten Systemverzeichnisse (wie `/usr/bin`), nicht aber das aktuelle Verzeichnis (`.`). Das verhindert, dass bösartige Skripte im aktuellen Ordner unabsichtlich ausgeführt werden. Mit `./` sagst du der Shell explizit: *„Führe die Datei genau hier im aktuellen Verzeichnis aus.“*

Ausführliche Hintergründe zum Linux-Rechtesystem findest du in unserem Leitfaden zu chmod und Dateiberechtigungen.

Ausführungsmethoden: Subshell vs. Interpreter vs. Sourcing

Es gibt drei grundlegende Wege, ein Shell-Skript unter Linux auszuführen:



1. Direkte Ausführung (`./hello_world.sh`)

Der Standardweg für fertige Skripte. Startet eine neue Kind-Shell (*Subshell*). Benötigt `chmod +x`.

2. Aufruf über den Interpreter (`bash hello_world.sh`)

Du übergibst die Datei direkt an die Bash:

```
bash hello_world.sh
```

- **Vorteil:** Die Datei benötigt kein Execute-Bit (`chmod +x`), sondern lediglich Leserechte (`r`), da die Bash die Datei als reine Textdatei liest.
- **Einsatz:** Ideal zum schnellen Testen während der Entwicklung.

3. Einlesen in die aktuelle Shell (`source` oder `.`)

Beim *Sourcen* wird kein neuer Prozess gestartet:

```
source hello_world.sh
# oder kurz:
. hello_world.sh
```

- **Wirkung:** Variablen und Funktionen, die im Skript definiert werden, bleiben nach Ablauf direkt in deiner aktuellen Terminal-Sitzung verfügbar (wird z. B. für `source ~/.bashrc` genutzt).

Vom einfachen Shell-Skript zum echten Bash-Skript

Bisher haben wir nur einfache Kommandos ausgeführt. Um das volle Potenzial der Bash zu nutzen und fortgeschrittene Sprachfeatures sowie Schutzmechanismen einzubinden, erweitern wir unser Skript:

Öffne `hello_world.sh` erneut im Editor:

```
nano hello_world.sh
```

Ersetze den Inhalt durch folgenden Code:

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

# Bash Strict Mode aktivieren
set -euo pipefail
IFS=$'\n\t'

# 1. Variable deklarieren
greeting="Hello, World!"

# 2. Parameter-Expansion: String in Großbuchstaben umwandeln
uppercase_greeting="${greeting^^}"

# 3. Farbige Ausgabe mit ANSI-Escape-Sequenzen
echo -e "\e[1;34m$greeting\e[0m"
echo -e "\e[1;31m$uppercase_greeting\e[0m"

# 4. Bash-spezifische Bedingung und Mustervergleich
if [[ "$greeting" == *"World"* ]]; then
    echo "Die Begrüßung enthält das Wort 'World'."
fi
```

Was bedeuten diese Neuerungen im Detail?

1. `set -euo pipefail` **(Bash Strict Mode):**

- `set -e` (*errexit*): Bricht das Skript sofort ab, wenn ein Befehl fehlschlägt.
- `set -u` (*nounset*): Verhindert die Ausführung bei Tippfehlern in Variablennamen.
- `set -o pipefail`: Erkennt Fehler auch innerhalb von Pipeline-Befehlen.

1. `IFS=$'\n\t'`: Verhindert Worttrennung bei Leerzeichen in Dateinamen.
2. `${greeting^^}`: Bash-spezifische Parameter-Expansion, die den Text in Großbuchstaben konvertiert.
3. `echo -e "\e[1;34m..."`: Aktiviert ANSI-Farbcodes (Blau und Rot) im Terminal.
4. `[[ "$greeting" == *"World"* ]]`: Der moderne Bash-Bedingungsoperator `[[ ]]` ermöglicht flexiblen Mustervergleich mit Wildcards.

Speichere die Änderungen und führe das Skript erneut aus:

```
./hello_world.sh
```

Du siehst nun eine farbige Ausgabe und die Bestätigung der Bedingungsprüfung im Terminal.

Exit-Codes & Rückgabewerte (`$?`)

Jedes saubere Skript gibt dem aufrufenden System einen Exit-Code zurück:

- `exit 0`: Signale an das Betriebssystem: **Erfolg** (*Success*).
- `exit 1` bis `exit 255`: Signalisiert einen **Fehlerzustand**.

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

echo "Systemprüfung läuft..."

# Erfolgreicher Abschluss
exit 0
```

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-erstelle-dein-erstes-bash-shell-skript

Den Exit-Code des zuletzt ausgeführten Befehls oder Skripts kannst du mit `echo $?` abfragen:

```
./hello_world.sh
echo "Status: $?"  # Gibt 0 aus
```

Debugging: Fehler in Skripten schnell finden

Wenn ein Skript nicht wie erwartet funktioniert, bietet die Bash eingebaute Diagnose-Optionen:

1. Ausführungs-Trace (`bash -x`)

Mit `-x` (*xtrace*) zeigt die Bash jeden ausgeführten Befehl mitsamt allen aufgelösten Variablen an:

```
bash -x hello_world.sh
```

2. Syntax-Prüfung (`bash -n`)

Prüft die Datei rein auf Syntaxfehler (wie vergessene Anführungszeichen), ohne Befehle auszuführen:

```
bash -n hello_world.sh
```

3. Statische Code-Analyse mit ShellCheck

Das Open-Source-Tool **ShellCheck** erkennt typische Fehler und schlechte Gewohnheiten:

```
# Installation unter Ubuntu/Debian
sudo apt install shellcheck

# Analyse durchführen
shellcheck hello_world.sh
```

Übungen & Praxis-Check

> **Praxisaufgaben zur Lektion #1:**
>
> 1. **Aufgabe 1 (Interaktiver Dialog):**
>
> Schreibe ein Skript `dialog.sh`, das den Benutzer mit `read -r` nach seinem Namen fragt und eine personalisierte, farbige Begrüßung ausgibt.
>
> 1. **Aufgabe 2 (Systemstatus-Report):**
>
> Erstelle ein Skript `status.sh`, das den aktuellen Benutzer (`$USER`), das Datum, den Hostnamen und den aktuellen Verzeichnispfad (`pwd`) strukturiert anzeigt.
>
> 1. **Aufgabe 3 (Strict Mode Test):**
>
> Erstelle ein Skript mit `set -u` und versuche, eine nicht definierte Variable auszugeben. Beobachte die Fehlermeldung der Bash.

Musterlösung zu Aufgabe 1:

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

# Frage nach dem Namen
echo -n "Wie heißt du? "
read -r name

# Personalisierte Begrüßung
greeting="Hallo, $name!"
echo -e "\e[1;32m$greeting\e[0m Willkommen zum Bash-Grundkurs."

exit 0
```

Befehlsreferenz (Cheatsheet)

Befehl / Syntax	Funktion & Beschreibung
`#!/usr/bin/env bash`	Portabler Shebang: Sucht Bash dynamisch über `$PATH`
`chmod +x skript.sh`	Macht die Skriptdatei für den Benutzer ausführbar
`./skript.sh`	Führt das Skript in einer isolierten Subshell aus
`bash skript.sh`	Führt das Skript über den Interpreter aus (benötigt nur Leserechte)
`source skript.sh`	Liest das Skript direkt in die aktuelle Shell ein
`set -euo pipefail`	Aktiviert den Bash Strict Mode für maximale Robustheit
`echo "$?"`	Zeigt den Exit-Code des zuletzt beendeten Programms
`bash -x skript.sh`	Startet das Skript im interaktiven Trace-Modus zur Fehlersuche

Befehl / Syntax	Funktion & Beschreibung
`bash -n skript.sh`	Prüft die Syntax der Skriptdatei ohne Ausführung

Weiterführende Ressourcen

Ressource	Beschreibung
GNU Bash Reference Manual	Die offizielle Dokumentation des GNU-Projekts
ShellCheck Tool	Online-Linter und Sicherheitsprüfer für Shell-Skripte
Linux Befehlszeilenprozessor Leitfaden	Ausführliche Grundlagen zur Shell-Architektur
chmod & Dateiberechtigungen	Detaillierte Rechteverwaltung unter Linux

Fazit

Mit dem Erstellen deines ersten Bash-Skripts, dem Verständnis für Shebang und Dateiberechtigungen sowie der Absicherung durch den Bash Strict Mode hast du das Fundament für alle weiteren Schritte in der Linux-Automatisierung gelegt.

> **Tipp:** Verwende bei jedem neuen Skript von Beginn an den portablen Shebang `#!/usr/bin/env bash` und aktiviere `set -euo pipefail`. Das verhindert unvorhergesehenes Verhalten und erleichtert die spätere Fehlersuche erheblich.

In der nächsten Lektion widmen wir uns der dynamischen Datenverarbeitung und schauen uns an, wie Variablen deklariert, manipuliert und in Skripten genutzt werden: **Weiter geht es mit:** Bash Grundkurs #2: Verwendung Von Variablen In Bash

Zur Kursübersicht: Alle Lektionen des Bash-Grundkurses