10. Shell, Pipes und Umleitungen

10.1 Shell, Pipes und Umleitungen

Die Shell ist eines der wichtigsten Werkzeuge unter Linux.

Sie nimmt Befehle entgegen, führt sie aus und kann Ausgaben, Eingaben und Fehlermeldungen gezielt weiterleiten.

Besonders wichtig sind dabei:

```
- Pipes
- Umleitungen
- Wildcards
- Variablen
- Befehlsverkettung
- Exit-Codes
- einfache Filterbefehle
```

Für Fachinformatiker Systemintegration ist dieses Thema wichtig, weil viele administrative Aufgaben unter Linux aus mehreren kleinen Befehlen bestehen, die miteinander kombiniert werden.

Merksatz:

```
Die Stärke der Shell liegt darin,
einfache Werkzeuge sinnvoll zu kombinieren.
```

Lernziele

Nach dieser Seite solltest du erklären können:

```
- was eine Shell ist
- was Bash ist
- was Standardausgabe,
  Standardeingabe
  und Standardfehler sind
- was Pipes machen
- wie Umleitungen funktionieren
- was >,
  >>,
```

```
  <,
  2>
  und &> bedeuten
- wie man Befehle verkettet
- was Exit-Codes sind
- wie man grep,
  sort,
  uniq,
  wc,
  cut
  und tee sinnvoll nutzt
```

Grundidee

Unter Linux geben viele Befehle Text aus.

Diese Ausgabe kann man:

```
- anzeigen
- filtern
- speichern
- an andere Befehle weitergeben
- in Dateien schreiben
- zählen
- sortieren
- durchsuchen
```

Beispiel:

```
ps aux | grep nginx
```

Bedeutung:

```
ps aux zeigt Prozesse.


grep nginx filtert nur Zeilen mit nginx.
```

Merksatz:

```
Befehle können ihre Ausgabe an andere Befehle weitergeben.
```

Shell

Eine Shell ist ein Befehlsinterpreter.

Sie verarbeitet eingegebene Befehle und startet Programme.

Bekannte Shells:

Shell	Bedeutung
sh	klassische Unix-Shell
bash	Bourne Again Shell
zsh	moderne interaktive Shell
fish	benutzerfreundliche Shell

Viele Linux-Systeme nutzen Bash als Standardshell.

Merksatz:

```
Shell = Befehlsinterpreter.
```

Bash

Bash steht für:

```
Bourne Again Shell
```

Bash kann:

```
- Befehle ausführen
- Variablen verwenden
- Ausgaben umleiten
- Pipes nutzen
- Bedingungen auswerten
- Schleifen ausführen
- Skripte starten
```

Beispiel:

```
echo "Hallo Linux"
```

Merksatz:

```
Bash ist Shell und Skriptumgebung.
```

Prompt

Der Prompt ist die Eingabeaufforderung.

Beispiel:

```
felix@server:~$
```

Bedeutung:

Teil	Bedeutung
felix	Benutzer
server	Hostname
~	Home-Verzeichnis
$	normaler Benutzer

Bei Root sieht man häufig:

```
root@server:~#
```

Merksatz:

```
$ normaler Benutzer.
# häufig Root.
```

Standardkanäle

Linux-Programme verwenden drei wichtige Standardkanäle.

Kanal	Nummer	Bedeutung
stdin	0	Standardeingabe
stdout	1	Standardausgabe
stderr	2	Standardfehler

Beispiel:

```
Ein Befehl liest Eingaben über stdin,
schreibt normale Ergebnisse nach stdout
und Fehlermeldungen nach stderr.
```

Merksatz:

```
0 = Eingabe.
1 = Ausgabe.
2 = Fehler.
```

Standardausgabe

Die Standardausgabe ist die normale Ausgabe eines Befehls.

Beispiel:

```
ls
```

zeigt Dateien im Terminal an.

Diese Ausgabe kann man umleiten oder weiterverarbeiten.

Beispiel:

```
ls > dateiliste.txt
```

Merksatz:

```
stdout ist die normale Ausgabe.
```

Standardfehler

Die Standardfehlerausgabe enthält Fehlermeldungen.

Beispiel:

```
ls /nicht-vorhanden
```

Mögliche Fehlermeldung:

```
No such file or directory
```

Diese Fehlermeldung kommt über stderr.

Merksatz:

```
stderr ist für Fehlermeldungen.
```

Ausgabe umleiten mit >

Mit > wird die Standardausgabe in eine Datei geschrieben.

Beispiel:

```
ls > dateien.txt
```

Wichtig:

```
Wenn die Datei bereits existiert,
wird sie überschrieben.
```

Beispiel:

```
echo "Test" > datei.txt
```

schreibt Test in datei.txt und ersetzt vorhandenen Inhalt.

Merksatz:

```
> schreibt neu und überschreibt.
```

Ausgabe anhängen mit >>

Mit >> wird die Standardausgabe an eine Datei angehängt.

Beispiel:

```
echo "Neue Zeile" >> datei.txt
```

Der vorhandene Inhalt bleibt erhalten.

Typische Nutzung:

```
Logs erweitern

Ergebnisse sammeln

Text an Datei anhängen
```

Merksatz:

```
>> hängt an.
```

Eingabe umleiten mit <

Mit < wird eine Datei als Eingabe für einen Befehl verwendet.

Beispiel:

```
wc -l < datei.txt
```

Bedeutung:

```
wc zählt Zeilen,
liest die Eingabe aber aus datei.txt.
```

Merksatz:

```
< nutzt Datei als Eingabe.
```

Fehler umleiten mit 2>

Mit 2> leitet man Fehlermeldungen um.

Beispiel:

```
ls /nicht-vorhanden 2> fehler.txt
```

Bedeutung:

```
Fehlermeldungen werden in fehler.txt geschrieben.
```

Normale Ausgabe bleibt davon getrennt.

Merksatz:

```
2> leitet Fehlermeldungen um.
```

Fehler anhängen mit 2>>

Mit 2>> werden Fehlermeldungen an eine Datei angehängt.

Beispiel:

```
ls /nicht-vorhanden 2>> fehler.log
```

Merksatz:

```
2>> hängt Fehlermeldungen an.
```

Ausgabe und Fehler gemeinsam umleiten

Ausgabe und Fehler gemeinsam umleiten:

```
befehl > ausgabe.log 2>&1
```

Bedeutung:

```
stdout geht nach ausgabe.log.

stderr wird ebenfalls dorthin geleitet.
```

Kurzform bei Bash:

```
befehl &> ausgabe.log
```

Anhängen:

```
befehl &>> ausgabe.log
```

Merksatz:

```
&> leitet Ausgabe und Fehler gemeinsam um.
```

/dev/null

/dev/null ist ein spezielles Gerät, das Ausgaben verwirft.

Beispiel:

```
befehl > /dev/null
```

Fehler verwerfen:

```
befehl 2> /dev/null
```

Alles verwerfen:

```
befehl &> /dev/null
```

Typische Nutzung:

```
unerwünschte Ausgaben unterdrücken
```

Wichtig:

```
Nicht blind verwenden,
weil Fehlermeldungen dann verloren gehen.
```

Merksatz:

```
/dev/null ist der digitale Papierkorb.
```

Pipe

Eine Pipe leitet die Ausgabe eines Befehls an den nächsten Befehl weiter.

Zeichen:

```
|
```

Beispiel:

```
ls -l | grep ".log"
```

Bedeutung:

```
ls -l erzeugt Ausgabe.

grep filtert diese Ausgabe.
```

Merksatz:

```
Pipe verbindet Befehle.
```

Pipe-Beispiel mit Prozessen

Beispiel:

```
ps aux | grep ssh
```

Bedeutung:

ps aux zeigt Prozesse.

grep ssh filtert Zeilen mit ssh.

Typische Nutzung:

Prozesse suchen

Logs filtern

Paketlisten durchsuchen

Ausgaben zählen

Merksatz:

Pipe macht aus mehreren einfachen Befehlen eine Auswertung.

Pipe-Beispiel mit Logs

Beispiel:

```
journalctl -u ssh | grep -i "failed"
```

Bedeutung:

journalctl -u ssh zeigt SSH-Logs.

grep -i "failed" filtert fehlgeschlagene Einträge.

Merksatz:

Logs werden mit Pipes schnell auswertbar.

---

grep

grep sucht nach Textmustern.

Beispiele:

```
grep "error" logfile.txt

grep -i "error" logfile.txt

grep -n "error" logfile.txt

grep -r "PermitRootLogin" /etc/ssh
```

Wichtige Optionen:

Option	Bedeutung
-i	Groß-/Kleinschreibung ignorieren
-n	Zeilennummer anzeigen
-r	rekursiv suchen
-v	Treffer umkehren
-l	nur Dateinamen mit Treffern anzeigen

Merksatz:

```
grep filtert Text.
```

---

sort

sort sortiert Textzeilen.

Beispiel:

```
cat namen.txt | sort
```

Numerisch sortieren:

```
sort -n zahlen.txt
```

Rückwärts sortieren:

```
sort -r namen.txt
```

Merksatz:

```
sort sortiert Zeilen.
```

---

uniq

uniq entfernt direkt aufeinanderfolgende doppelte Zeilen.

Beispiel:

```
sort namen.txt | uniq
```

Wichtig:

```
uniq erkennt doppelte Zeilen nur,
wenn sie nebeneinander stehen.
Deshalb wird oft vorher sort verwendet.
```

Anzahl anzeigen:

```
sort namen.txt | uniq -c
```

Merksatz:

```
sort und uniq werden oft kombiniert.
```

---

wc

wc zählt Zeilen, Wörter und Zeichen.

Beispiele:

```
wc datei.txt

wc -l datei.txt
```

```
ps aux | wc -l
```

Bedeutung:

Option	Bedeutung
-l	Zeilen zählen
-w	Wörter zählen
-c	Bytes zählen

Merksatz:

```
wc -l zählt Zeilen.
```

---

cut

cut schneidet Spalten oder Zeichen aus Text.

Beispiel mit Trennzeichen:

```
cut -d ":" -f 1 /etc/passwd
```

Bedeutung:

```
-d ":" verwendet Doppelpunkt als Trenner.

-f 1 gibt das erste Feld aus.
```

Typische Nutzung:

```
einfache strukturierte Dateien auswerten

Spalten ausgeben
```

Merksatz:

```
cut extrahiert Felder aus Text.
```

---

awk kurz eingeordnet

awk ist ein Werkzeug zur Textverarbeitung.

Beispiel:

```
awk '{print $1}' datei.txt
```

Bedeutung:

```
gibt die erste Spalte aus.
```

awk ist mächtiger als cut, weil es Bedingungen, Berechnungen und komplexere Ausgaben kann.

Merksatz:

```
awk verarbeitet spaltenorientierten Text.
```

---

sed kurz eingeordnet

sed ist ein Stream-Editor.

Er kann Textströme verändern.

Beispiel:

```
sed 's/alt/neu/g' datei.txt
```

Bedeutung:

```
ersetzt alt durch neu in der Ausgabe.
```

Wichtig:

```
Ohne passende Option wird die Datei nicht automatisch geändert,
sondern nur die Ausgabe.
```

Merksatz:

```
sed verändert Textströme.
```

---

tee

tee schreibt Ausgabe gleichzeitig auf den Bildschirm und in eine Datei.

Beispiel:

```
echo "Test" | tee datei.txt
```

Anhängen:

```
echo "Neue Zeile" | tee -a datei.txt
```

Nützlich, wenn man Ausgabe sehen und speichern möchte.

Beispiel:

```
systemctl status ssh | tee ssh-status.txt
```

Merksatz:

```
tee zeigt und speichert gleichzeitig.
```

Befehle nacheinander ausführen mit ;

Mit ; führt man Befehle nacheinander aus, unabhängig davon, ob der vorherige erfolgreich war.

Beispiel:

```
echo "Start"; echo "Ende"
```

Auch wenn der erste Befehl fehlschlägt, wird der zweite ausgeführt.

Merksatz:

```
; führt einfach nacheinander aus.
```

Befehle nur bei Erfolg ausführen mit &&

Mit && wird der zweite Befehl nur ausgeführt, wenn der erste erfolgreich war.

Beispiel:

```
mkdir backup && cp datei.txt backup/
```

Bedeutung:

```
Nur wenn mkdir erfolgreich ist,
wird cp ausgeführt.
```

Merksatz:

```
&& bedeutet:
nur bei Erfolg weiter.
```

Befehle nur bei Fehler ausführen mit ||

Mit || wird der zweite Befehl nur ausgeführt, wenn der erste fehlschlägt.

Beispiel:

```
ping -c 1 server || echo "Server nicht erreichbar"
```

Merksatz:

```
|| bedeutet:
nur bei Fehler ausführen.
```

Exit-Code

Jeder Befehl liefert einen Exit-Code zurück.

Typisch:

Exit-Code	Bedeutung
0	erfolgreich

Exit-Code	Bedeutung
ungleich 0	Fehler oder besonderer Zustand

Den letzten Exit-Code anzeigen:

```
echo $?
```

Beispiel:

```
ls /etc

echo $?
```

Merksatz:

```
Exit-Code 0 bedeutet Erfolg.
```

Warum Exit-Codes wichtig sind

Exit-Codes sind wichtig für:

```
- Skripte
- Automatisierung
- && und ||
- Monitoring
- Fehlerbehandlung
- Cronjobs
- systemd-Units
```

Beispiel:

```
grep "root" /etc/passwd

echo $?
```

Wenn grep etwas findet, ist der Exit-Code 0. Wenn nichts gefunden wird, ist der Exit-Code ungleich 0.

Merksatz:

```
Skripte entscheiden oft anhand von Exit-Codes.
```

Variablen

In der Shell können Variablen verwendet werden.

Beispiel:

```
NAME="Felix"
```

Ausgeben:

```
echo "$NAME"
```

Wichtig:

```
Keine Leerzeichen um das Gleichheitszeichen.
```

Richtig:

```
NAME="Felix"
```

Falsch:

```
NAME = "Felix"
```

Merksatz:

```
Variablen speichern Werte.
```

Umgebungsvariablen

Umgebungsvariablen beeinflussen Programme und Shell-Verhalten.

Beispiele:

Variable	Bedeutung
HOME	Home-Verzeichnis
USER	Benutzername
PATH	Suchpfade für Programme
SHELL	aktuelle Shell
PWD	aktuelles Verzeichnis

Anzeigen:

```
echo "$HOME"

echo "$PATH"
```

Alle Umgebungsvariablen anzeigen:

```
env
```

Merksatz:

```
Umgebungsvariablen beschreiben die Umgebung eines Prozesses.
```

PATH

PATH enthält Verzeichnisse, in denen die Shell nach Programmen sucht.

Anzeigen:

```
echo "$PATH"
```

Beispielinhalt:

```
/usr/local/bin:/usr/bin:/bin
```

Wenn man einen Befehl eingibt, sucht die Shell in diesen Verzeichnissen.

Prüfen, welches Programm ausgeführt wird:

```
which ssh
```

Merksatz:

```
PATH bestimmt,
wo Befehle gesucht werden.
```

Quotes

Anführungszeichen beeinflussen, wie die Shell Text verarbeitet.

Zeichen	Bedeutung
" "	Variablen werden ausgewertet
' '	Text bleibt fast vollständig unverändert
\	maskiert ein einzelnes Zeichen

Beispiele:

```
NAME="Felix"

echo "Hallo $NAME"
```

Ausgabe:

```
Hallo Felix
```

Beispiel:

```
echo 'Hallo $NAME'
```

Ausgabe:

```
Hallo $NAME
```

Merksatz:

Doppelte Quotes erlauben Variablen.

Einfache Quotes schützen Text stärker.

Wildcards

Wildcards sind Platzhalter.

Wildcard	Bedeutung
*	beliebige Zeichen
?	genau ein Zeichen
[abc]	eines der Zeichen a, b oder c

Beispiele:

```
ls *.log

rm *.tmp

cp backup-?.tar /tmp/
```

Merksatz:

```
* steht für beliebige Zeichen.
```

Achtung bei Wildcards und rm

Gefährlich:

```
rm *
```

Noch gefährlicher:

```
rm -rf *
```

Sicherer prüfen:

```
echo *

ls

pwd
```

Dann erst löschen.

Merksatz:

```
Vor rm mit Wildcards immer prüfen,
was betroffen ist.
```

Befehlshistorie

Die Shell speichert häufig eine Befehlshistorie.

Anzeigen:

```
history
```

Letzten Befehl erneut ausführen:

```
!!
```

Nach Befehlen suchen:

```
Strg + R
```

Wichtig:

```
Keine Passwörter oder Geheimnisse direkt in Befehle schreiben,
weil sie in der Historie landen können.
```

Merksatz:

```
history ist praktisch,
kann aber Geheimnisse verraten.
```

Tab-Vervollständigung

Mit Tab kann die Shell Befehle, Dateinamen und Pfade vervollständigen.

Vorteile:

```
weniger Tippfehler

schnelleres Arbeiten

Pfade prüfen

Befehle entdecken
```

Merksatz:

```
Tab spart Zeit und reduziert Tippfehler.
```

Aliases

Aliases sind Abkürzungen für Befehle.

Beispiel:

```
alias ll='ls -la'
```

Danach kann man eingeben:

```
ll
```

und es wird ausgeführt:

```
ls -la
```

Wichtig:

```
Aliases gelten je nach Konfiguration nur in der aktuellen Shell
oder dauerhaft über Dateien wie ~/.bashrc.
```

Merksatz:

```
Aliases sind Befehlsabkürzungen.
```

.bashrc und .profile

Benutzerspezifische Shell-Konfigurationen liegen häufig in:

```
~/.bashrc

~/.profile
```

Dort können stehen:

```
Aliases

Umgebungsvariablen

Prompt-Konfiguration

Shell-Funktionen

PATH-Erweiterungen
```

Wichtig:

```
Fehler in diesen Dateien können Shell-Verhalten beeinflussen.
```

Merksatz:

```
~/.bashrc beeinflusst die interaktive Bash.
```

Typische Befehlsketten

Prozesse nach nginx suchen:

```
ps aux | grep nginx
```

Fehler in Logs zählen:

```
grep -i "error" /var/log/syslog | wc -l
```

Größte Ordner anzeigen:

```
du -sh * | sort -h
```

Benutzer aus /etc/passwd anzeigen:

```
cut -d ":" -f 1 /etc/passwd
```

Doppelte Zeilen zählen:

```
sort datei.txt | uniq -c
```

Merksatz:

```
Pipes machen Auswertungen aus einfachen Befehlen.
```

Typische Fehlerbilder

Fehlerbild	Wahrscheinliche Ursache
Datei leer nach Befehl	> hat Datei überschrieben
Fehler nicht in Datei	stderr wurde nicht umgeleitet
Befehl nicht gefunden	PATH oder Paket fehlt
Variable leer	falsch gesetzt oder falsch zitiert
rm löscht zu viel	Wildcard falsch verwendet
grep findet nichts	Groß-/Kleinschreibung oder Muster falsch
Pipe liefert nichts	erster Befehl erzeugt keine passende Ausgabe

Fehlerbild	Wahrscheinliche Ursache
Permission denied	fehlende Rechte
Ausgabe fehlt	nach /dev/null umgeleitet
Skript reagiert falsch	Exit-Code nicht beachtet

Befehl nicht gefunden

Fehlermeldung:

```
command not found
```

Mögliche Ursachen:

```
Paket nicht installiert

Tippfehler

Befehl liegt nicht im PATH

Skript nicht ausführbar

falsche Shell
```

Prüfen:

```
which befehl

echo "$PATH"

ls -l datei
```

Merksatz:

```
command not found bedeutet:
Shell findet den Befehl nicht.
```

Permission denied bei Skript

Fehlerbild:

```
./script.sh: Permission denied
```

Mögliche Ursachen:

```
Skript hat kein Ausführrecht

Dateisystem ist mit noexec gemountet

falsche Rechte
```

Prüfen:

```
ls -l script.sh
```

Ausführbar machen:

```
chmod +x script.sh
```

Starten:

```
./script.sh
```

Merksatz:

```
Skript direkt starten braucht x-Recht.
```

Sichere Arbeitsweise in der Shell

Vor riskanten Befehlen:

```
pwd

ls -la

echo *.tmp
```

```
Befehl ohne Löschung testen,
wenn möglich
```

Bei Umleitungen:

```
> überschreibt

>> hängt an
```

Bei Root-Rechten:

```
Befehl genau prüfen

Pfad genau prüfen

Wildcards vermeiden oder vorher testen
```

Merksatz:

```
In der Shell erst prüfen,
dann ausführen.
```

Typische Prüfungsfragen

Frage	Kurzantwort
Was ist eine Shell?	Befehlsinterpreter
Was ist Bash?	verbreitete Linux-Shell
Was macht eine Pipe?	Ausgabe an nächsten Befehl weitergeben
Was macht >?	Ausgabe in Datei schreiben und überschreiben
Was macht >>?	Ausgabe an Datei anhängen
Was macht 2>?	Fehlerausgabe umleiten
Was ist stdout?	Standardausgabe
Was ist stderr?	Standardfehler
Was ist stdin?	Standardeingabe
Was bedeutet Exit-Code 0?	Erfolg

Frage	Kurzantwort
Was macht grep?	Text filtern
Was macht wc -l?	Zeilen zählen
Was macht sort?	Zeilen sortieren
Was macht uniq?	doppelte Nachbarzeilen entfernen
Was macht tee?	Ausgabe anzeigen und speichern

Typische Prüfungsfallen

Falle	Richtig denken
> und >> verwechseln	> überschreibt, >> hängt an
stderr nicht umleiten	Fehler gehen über Kanal 2
Pipe mit Umleitung gleichsetzen	Pipe zu Befehl, Umleitung zu Datei
grep ohne -i bei unbekannter Schreibweise	-i ignoriert Groß-/Kleinschreibung
uniq ohne sort nutzen	doppelte Zeilen müssen nebeneinander stehen
rm mit * unterschätzen	vorher mit echo oder ls prüfen
Variablen mit Leerzeichen setzen	NAME="Wert", nicht NAME = "Wert"
einfache und doppelte Quotes verwechseln	einfache Quotes werten Variablen nicht aus
command not found falsch deuten	Paket, PATH oder Tippfehler prüfen
Exit-Codes ignorieren	wichtig für Skripte und Automatisierung

---

IHK-sichere Kurzformulierung

Die Shell ist ein Befehlsinterpreter, der Befehle ausführt und deren Ein- und Ausgaben steuern kann. Linux unterscheidet Standardeingabe, Standardausgabe und Standardfehler mit den Kanälen 0, 1 und 2. Mit Pipes wird die Ausgabe eines Befehls an einen weiteren Befehl weitergegeben. Mit > wird Ausgabe in eine Datei geschrieben und überschrieben, mit >> wird sie angehängt. Fehlerausgaben können mit 2> umgeleitet werden. Werkzeuge wie grep, sort, uniq, wc, cut, awk, sed und tee werden häufig kombiniert, um Text, Logs und Befehlsausgaben auszuwerten. Exit-Codes zeigen an, ob ein Befehl erfolgreich war, und sind besonders wichtig für Skripte und Automatisierung.

Merksätze

```
Shell = Befehlsinterpreter.

Bash ist eine verbreitete Shell.
```

stdin = Eingabe.

stdout = Ausgabe.

stderr = Fehlerausgabe.

0 = stdin.

1 = stdout.

2 = stderr.

> überschreibt.

>> hängt an.

2> leitet Fehler um.

&> leitet Ausgabe und Fehler um.

/dev/null verwirft Ausgaben.

Pipe verbindet Befehle.

grep filtert Text.

sort sortiert Zeilen.

uniq entfernt doppelte Nachbarzeilen.

wc -l zählt Zeilen.

cut extrahiert Felder.

awk verarbeitet Spalten.

sed verändert Textströme.

tee zeigt und speichert gleichzeitig.

; führt nacheinander aus.

&& läuft nur bei Erfolg weiter.

|| läuft nur bei Fehler weiter.

Exit-Code 0 bedeutet Erfolg.

PATH bestimmt,
wo Befehle gesucht werden.

Doppelte Quotes werten Variablen aus.

Einfache Quotes schützen Text stärker.

* steht für beliebige Zeichen.

Vor rm mit Wildcards immer prüfen.

Erst testen,
dann ausführen.