Buchseiten 3
1. Skript-Grundlagen & Datenfluss 4
2. Wichtige Variablen 4
3. Logik & Bedingungen 4
Die If-Abfrage: 4
1. for – loop Beispiel: Über eine Liste iterieren 5
2. Beispiel: Über einen Zahlenbereich iterieren 5
3. Profi-Tipp: Dateien im Verzeichnis verarbeiten 6
4. Test-Operatoren (Bedingungen prüfen) 7
4.2 If ausführlich 8
4.2.1 Die einfache if-Anweisung 8
4.2.2 Das Programm test 8
4.2.2.1 Die verschiedenen Bedingungsüberprüfungen mit test bzw. [(Buch Seite 312) 9
4.2.2.2 Die erweiterte if-else Anweisung 10
4.2.2.3 Die if-elif-else Anweisung 11
4.2.3. Mehrfachauswahl mit case 11
5. Essenzielle Werkzeuge 11
7. Docker 14
1. Der Workflow: Vom Code zum Container 14
Images verwalten (Die Vorbereitungen) 14
2. Docker Volumes: Das Langzeitgedächtnis 15
Die drei Arten der Speicherung 15
A. Named Volumes (Der Docker-Standard) 15
B. Bind Mounts (Der Entwickler-Liebling) 15
C. Anonymous Volumes 15
3. Die ultimative Kombination (Beispiel) 16
Kurze Checkliste für den Befehl oben: 16
Nützliche Volume-Commands 16
2. Container steuern (Die Action) 17
Der `docker run` Befehl 17
3. Laufende Container verwalten (Das Management) 17

4. Insider-Befehle (Für Profis) ... 18
Ein typischer Workflow in der Praxis: ... 18
8. Docker File ... 19
Beispiel ... 19
9. Wichtige andere Befehle ... 20
9.1. `grep` ... 20
Wichtige Optionen ... 21
9.2. `sed` ... 22
Wichtige Optionen ... 22
Häufige Befehle ... 22
Ersetzen ... 22
Zeilen löschen ... 22
Zeilen anzeigen ... 23
Text einfügen ... 23
9.3. `find` ... 23
Wichtige Optionen ... 23
Dateitypen ... 24
Größe ... 24
Zeit ... 24
Aktionen ... 24
9.4. `awk` ... 25
Grundprinzip ... 25
Beispiele ... 25
Erste Spalte anzeigen ... 25
Mehrere Spalten ... 25
Bedingungen ... 26
Trennzeichen ändern ... 26
Berechnungen ... 26
Wann benutzt man welches Tool? ... 26
Typische Kombinationen ... 26
Kurzvergleich ... 26
Anhang "**Juli** Nützlich" ... 27

Analyse.sh 27

Backup.sh 29

Count.sh 30

Envinfo.sh 30

List_info.sh 30

Buchseiten

Seite	Inhalt	Beschreibung
180	SystemD	Diensteverwaltung und Init-System.
28	Verzeichnisbaum	Hierarchische Dateistruktur ab /.
161	Sudo	Befehle mit Root-Rechten ausführen.
63	Ls-Befehl	Verzeichnisinhalt auflisten.
90	Touch-Befehl	Neue, leere Datei erstellen.
65	Cp-Befehl	Dateien/Ordner kopieren.
66	Mv-Befehl	Verschieben oder Umbenennen.
67	Rm-Befehl	Dateien oder Ordner löschen.
72	Mkdir-Befehl	Neues Verzeichnis erstellen.
41	Umgebungsvariablen	Werte zur Systemkonfiguration (z. B. $PATH).
60	Man-Befehl	Handbuch für Befehle anzeigen.
109	Grep-Befehl	Text nach Mustern durchsuchen.
124	Awk-Befehl	Text- und Spaltenbearbeitung.
125	Sed-Befehl	Automatisierte Textveränderung.
100	Find-Befehl	Dateien im System suchen.
76	Head-Befehl	Dateianfang anzeigen (Standard: erste 10 Zeilen).
77	Tail-Befehl	Dateiende anzeigen (Standard: letzte 10 Zeilen).
86	Stat-Befehl	Zeigt detaillierte Datei-Metadaten an.
124	Tee-Befehl	Schreibt Ausgabe in Datei und zeigt sie zeitgleich an.

1. Skript-Grundlagen & Datenfluss

- **Shebang:** Beginne ein Skript immer mit #! (z. B. #!/bin/bash), um die Shell festzulegen.
- **Ausführbar machen**: chmod +x <scriptname>.sh
- **Ausgabe überschreiben:** > file erstellt oder überschreibt eine Datei.
- **Ausgabe anhängen:** >> file fügt Text ans Ende einer Datei an.
- **Eingabe lesen:** < file liest den Inhalt einer Datei ein.
- **Pipe:** a | b leitet die Ausgabe von Befehl "a" als Eingabe an Befehl "b" weiter.
- **Kommentare**: # am Beginn der Zeile
- **Variablen-Zuweisung**: variablenname="<wert>" (Kein Leerzeichen um das =)
- **Variablen-Aufruf**: $variablenname (Case-Sensitive)
- **Umgebungsvariable**: export global="<wert>"
    - Für Unterprozesse sichtbar
- **Script-Aufruf**: ./scriptname.sh [param1]
- **User-Eingabe während des Scripts**: read [-s] [-r] [-p "<prompt>"] <variablenname>
    - -p "<prompt>": Prompt vor der Eingabe
    - -s: verdeckte Eingabe (Passwörter)
    - -r: verhindert Escape-Sequenzen

2. Wichtige Variablen

- **$0**: Der Name des aufgerufenen Programms.
- **$1, $2, …**: Das erste und zweite übergebene Argument.
- **$#**: Die Anzahl der übergebenen Argumente.
- **$***: Alle übergebenen Argumente auf einmal.
- **$?**: Der Rückgabecode des letzten Befehls (0 = Fehlerfrei).
- **$$**: Die Prozess-ID (PID) des aktuellen Skripts.
- **${V:-default}**: Gibt den Wert von $V aus, oder "default", falls die Variable leer ist.

3. Logik & Bedingungen

Befehle direkt verketten:

- cmd1 && cmd2: Führe cmd1 aus; wenn er erfolgreich war, führe cmd2 aus.
- cmd1 || cmd2: Führe cmd1 aus; wenn er fehlschlägt, führe cmd2 aus.

Die If-Abfrage:

Bash

```
if [ "$x" -lt "$y" ]; then
  # do something
fi
```

Die Case-Anweisung (Mehrfach-Auswahl):

Bash
```
case $foo in
  a) echo "foo is A" ;; #(foo = a)
  b) echo "foo is B" ;; #(foo = b)
  *) echo "foo is not A or B" ;; #(alles anderes)
esac
```
(Wichtig: Das ;; am Ende jedes Blocks ist zwingend erforderlich).

1. for – loop Beispiel: Über eine Liste iterieren

Dies ist der klassische Anwendungsfall: Du gehst eine Liste von Dateien oder Werten nacheinander durch.

Bash
```
#!/bin/bash

MEIN_ARRAY=("Apfel Birne" "Banane" "Kirsche")
MEINE_LISTE="Apfel Birne Banane"
for obst in $MEINE_LISTE; do
    echo "Ich mag $obst"
Done

# Liste von Werten durchgehen
for tier in Hund Katze Maus; do
    echo "Das ist ein(e): $tier"
done
```

2. Beispiel: Über einen Zahlenbereich iterieren

Wenn du eine Aktion eine bestimmte Anzahl an Malen wiederholen willst (z. B. 5-mal):

Bash
```
#!/bin/bash

# Zählen von 1 bis 5
for i in {1..5}; do
    echo "Durchlauf Nummer: $i"
done
```

3. Profi-Tipp: Dateien im Verzeichnis verarbeiten

Ein sehr häufiger Anwendungsfall in der Shell ist es, alle Dateien eines bestimmten Typs zu bearbeiten:

Bash
```
#!/bin/bash

# Alle .txt Dateien im aktuellen Verzeichnis finden und ausgeben
for datei in *.txt; do
    echo "Verarbeite Datei: $datei"
    # Hier könnte z.B. ein grep oder cat Befehl folgen
done
```

Zusammenfassung der Struktur:

- **`for`**: Startet die Schleife.
- **`variable`**: Ein frei wählbarer Name, der bei jedem Durchlauf den aktuellen Wert annimmt.
- **`in`**: Definiert die Liste oder den Bereich, der abgearbeitet wird.
- **do**: Leitet den Codeblock ein, der für jedes Element ausgeführt wird.
- **done**: Beendet den Schleifenblock.

for	Wenn die Anzahl der Elemente feststeht.
while	Wenn du auf einen Zustand wartest.
continue	Wenn ein einzelnes Element ignoriert werden soll.
break	Wenn die Arbeit vorzeitig erledigt ist.

4. Test-Operatoren (Bedingungen prüfen)

Typ	Operatoren	Bedeutung
Zahlen	-eq / -ne	Gleich / Ungleich
	-lt / -le	Kleiner als / Kleiner oder gleich
	-gt / -ge	Größer als / Größer oder gleich
Text	=	Strings sind gleich
	-z / -n	Länge ist Null (leer) / Nicht leer
Dateien	-d / -x	Ist ein Verzeichnis / Ist ausführbar
	-r / -w	Ist lesbar / Ist schreibbar
	-nt	Ist neuer als (newer than)
Logik	&& / `	
	!	Logisches NICHT

While-Schleife (Datei zeilenweise einlesen):

Bash

```
while read f
do
   echo "Line is $f"
done < dateiname.txt


while read benutzer
do
  mkdir "/home/$benutzer"
  echo "Ordner für $benutzer wurde erstellt."
done < benutzerliste.txt #Datei einlesen wird als 1. ausgeführt
```

Case-Anweisung (Mehrfach-Auswahl):

Bash

```
case $foo in
  a) echo "foo is A" ;;
  *) echo "foo is not A" ;;
esac
```

Funktionen definieren und aufrufen:

Bash

```
doubleit() {
   expr $1 \* 2
}
doubleit 3
```

4.2 If ausführlich

4.2.1 Die einfache if-Anweisung

Grundsätzlich hat die if-Anweisung der Bourne-Shell eine sehr einfache Form. Nach dem if steht ein Befehl, der ausgeführt wird. Gibt dieses Kommando eine 0 als Rückgabewert zurück, so gilt die Bedingung als erfüllt und die Aktionen, die zwischen dem folgenden then und fi stehen werden ausgeführt.

```
if Kommando
then
  Aktion
  Aktion
  ...
fi
```

Natürlich sind die Aktionen auch wieder normale Unix-Befehle. Das „fi“, das den Block beendet, der durch „if ... then“ begonnen wurde, ist einfach nur das „if“ rückwärts geschrieben.

4.2.2 Das Programm test

Damit es jetzt sinnvolle Möglichkeiten gibt, Bedingungen zu überprüfen brauchen wir ein Programm, das verschiedene Tests durchführt und jeweils bei gelungenem Test eine 0 als Rückgabewert zurückgibt, bei mislingenem Test eine 1. Dieses Programm heißt *test* und ermöglicht alle wesentlichen Bedingungsüberprüfungen, die für das Shell-Programmieren notwendig sind.

Damit wir nicht jedesmal schreiben müssen

```
if test ...
```

gibt es einen symbolischen Link auf das Programm test, der einfach [heißt. Allerdings verlangt das Programm test, wenn es merkt, dass es als [aufgerufen wurde, auch als letzten Parameter eine eckige Klammer zu. Damit ist es also möglich zu schreiben:

```
if [ ... ]
```

Wichtig ist hierbei, dass unbedingt ein Leerzeichen zwischen if und der Klammer und zwischen der Klammer und den eigentlichen Tests stehen muß. Es handelt sich bei der Klammer ja tatsächlich um einen Programmaufruf!

```
DATEI="test.txt" # Prüfen, ob die Datei existiert UND sowohl lesbar als auch beschreibbar ist
if [ -f "$DATEI" ] && [ -r "$DATEI" ] && [ -w "$DATEI" ]; then
```

4.2.2.1 Die verschiedenen Bedingungsüberprüfungen mit test bzw. [(Buch Seite 312)

-r *Dateiname: if [-r file.txt]; then:* Die Datei *Dateiname* existiert und ist lesbar

-w *Dateiname*: Die Datei *Dateiname* existiert und ist beschreibbar

-x *Dateiname*: Die Datei *Dateiname* existiert und ist ausführbar

-d *Dateiname*: Die Datei *Dateiname* existiert und ist ein Verzeichnis

-s *Dateiname*: Die Datei *Dateiname* existiert und ist nicht leer

-b *Dateiname*: Die Datei *Dateiname* existiert und ist ein blockorientiertes Gerät

-c *Dateiname*: Die Datei *Dateiname* existiert und ist ein zeichenorientiertes Gerät

-g *Dateiname*: Die Datei *Dateiname* existiert und das SGID-Bit ist gesetzt

-k *Dateiname*: Die Datei *Dateiname* existiert und das Sticky-Bit ist gesetzt

-u *Dateiname*: Die Datei *Dateiname* existiert und das SUID-Bit ist gesetzt

-p *Dateiname*: Die Datei *Dateiname* existiert und ist ein Named Pipe

-e *Dateiname*: Die Datei *Dateiname* existiert

-f *Dateiname*: Die Datei *Dateiname* existiert und ist eine reguläre Datei

-L *Dateiname*: Die Datei *Dateiname* existiert und ist ein symbolischer Link

-S *Dateiname*: Die Datei *Dateiname* existiert und ist ein Socket

-O *Dateiname*: Die Datei *Dateiname* existiert und ist Eigentum des Anwenders, unter dessen UID das test-Programm gerade läuft

-G *Dateiname*: Die Datei *Dateiname* existiert und gehört zu der Gruppe, zu der der User gehört, unter dessen UID das test-Programm gerade läuft

Datei1* -nt *Datei2: *Datei1* ist neuer als *Datei2* (newer than)

Datei1* -ot *Datei2: *Datei1* ist älter als *Datei2* (older than)

Datei1* -ef *Datei2: *Datei1* und *Datei2* benutzen die gleiche I-Node (equal file)

-z *Zeichenkette*: Wahr wenn *Zeichenkette* eine Länge von Null hat.

-n *Zeichenkette*: Wahr wenn *Zeichenkette* eine Länge von größer als Null hat.

Zeichenkette1* = *Zeichenkette2: Wahr wenn *Zeichenkette1* gleich *Zeichenkette2*

Zeichenkette1* != *Zeichenkette2: Wahr wenn *Zeichenkette1* ungleich *Zeichenkette2*

Wert1* -eq *Wert2: Wahr, wenn *Wert1* gleich *Wert2* (equal)

Wert1* -ne *Wert2: Wahr, wenn *Wert1* ungleich *Wert2* (not equal)

Wert1* -gt *Wert2: Wahr, wenn *Wert1* größer *Wert2* (greater than)

Wert1* -ge *Wert2: Wahr, wenn *Wert1* größer oder gleich *Wert2* (greater or equal)

Wert1* -lt *Wert2: Wahr, wenn *Wert1* kleiner *Wert2* (less than)

Wert1* -le *Wert2: Wahr, wenn *Wert1* kleiner oder gleich *Wert2* (less or equal)

!*Ausdruck*: Logische Verneinung von *Ausdruck*

Ausdruck* -a *Ausdruck: Logisches UND. Wahr, wenn beide Ausdrücke wahr sind

Ausdruck* -o *Ausdruck: Logisches ODER. Wahr wenn mindestens einer der beiden Ausdrücke wahr ist

Mit diesen Tests sind so ziemlich alle denkbaren Bedingungsüberprüfungen möglich, die in einem Shellscript notwendig sind.

4.2.2.2 Die erweiterte if-else Anweisung

Natürlich bietet die if-Anweisung auch eine Erweiterung zur normalen Form, die sogenannte if-else Anweisung. Es ist also möglich zu schreiben:

```
if [ Ausdruck ];
then
  Kommandos
else
  Kommandos
fi
```

4.2.2.3 Die if-elif-else Anweisung

Um noch einen Schritt weiterzugehen bietet die if-Anweisung sogar ein weiteres if im else, das sogenannte elif, das wieder eine Bedingung überprüft:

```
if [ Ausdruck ];
then
  Kommandos
elif [ Ausdruck ];
then
  Kommandos
else
  Kommandos
fi
```

4.2.3. Mehrfachauswahl mit case

Oft kommt es vor, dass eine Variable ausgewertet werden muß und es dabei viele verschiedenen Möglichkeiten gibt, welche Werte diese Variable annehmen kann. Natürlich wäre es mit einer langen if-elif-elif-elif... Anweisung möglich, so etwas zu realisieren, das wäre aber sowohl umständlich, als auch schwer zu lesen. Damit solche Fälle einfacher realisiert werden können, gibt es die Mehrfachauswahl mit case. Der prinzipielle Aufbau einer case-Entscheidung sieht folgendermaßen aus:

```
case Variable in
  Muster1) Kommando1 ;;
  Muster2) Kommando2 ;;
  Muster3) Kommando3 ;;
  ...
esac
```

5. Essenzielle Werkzeuge

- **Suchen:** grep foo myfile findet Zeilen mit dem Text "foo" in einer Datei.
- **Dateien finden:** find . -name "*.txt" -print sucht nach Textdateien im aktuellen Verzeichnis.
- **Text filtern:** awk '{print $5}' file gibt nur das 5. Wort jeder Zeile aus.
- **Text ersetzen:** sed s/foo/bar/g file ersetzt "foo" durch "bar".

6. Funktionen

1. Der Exit-Status (Die Rückgabe von Erfolg/Fehler)

Jeder Befehl in Linux – und somit auch jede Funktion – endet mit einem Exit-Status. Dieser ist eine Zahl zwischen **0 und 255**.

- **0**: Der Befehl war erfolgreich.
- **1–255**: Es trat ein Fehler auf (1 ist oft ein allgemeiner Fehler).

Du setzt diesen Status explizit mit dem Befehl `return`.

Bash

```bash
check_file() {
    if [ -f "$1" ]; then
        return 0  # Datei existiert
    else
        return 1  # Datei existiert nicht
    fi
}

# Aufruf:
if check_file "test.txt"; then
    echo "Alles okay."
else
    echo "Fehler: Datei nicht gefunden."
fi
```

Wichtig: Der Status wird in der Shell-Variable `$?` gespeichert. Er ist **ausschließlich** für die Logik (Erfolg/Fehler) gedacht, nicht um Daten zu übergeben.

2. Die Standard-Ausgabe (Die Rückgabe von Daten)

Wenn du einen **Wert** (z. B. einen Dateinamen, eine Berechnung oder einen String) aus der Funktion an dein Skript zurückgeben willst, musst du die **Standard-Ausgabe (stdout)** nutzen.

Die Funktion „schreibt" das Ergebnis einfach auf die Konsole (mit `echo`), und du fängst es mit der **Command Substitution** `$(...)` ab.

Bash

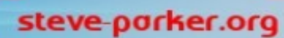
```bash
# Funktion berechnet etwas und "gibt es aus"
addiere() {
    local summe=$(( $1 + $2 ))
    echo $summe  # Das ist die Rückgabe der Daten
}

# Aufruf: Fange die Ausgabe der Funktion in einer Variable ab
ergebnis=$(addiere 10 20)

echo "Das Ergebnis ist: $ergebnis"
```

.

http://steve-parker.org/sh/sh.shtml

v1.1 – 7 Aug 2007

UNIX / Linux Shell Cheat Sheet



File Manipulation

> file	create (overwrite) file
>> file	append to file
>file 2>&1	both output and errors to file
< file	read from file
a | b	pipe output from 'a' as input to 'b'

Common Constructs

```
while read f
do
  echo "Line is $f"
done < file
```

read text file line by line

```
$ grep foo myfile
afoo
foo
foobar
```

find matching lines

```
$ cut -d: -f5 /etc/passwd
Dilbert
```

get field with delimiter

```
foo=`ls`
```

get output of command

```
case $foo in
  a)
    echo "foo is A"
    ;;
  b)
    echo "foo is B"
    ;;
  *)
    echo "foo is not A or B"
    ;;
 esac
```

case is a good way to avoid iterating through many if/elif/elif/elif constructs.

```
doubleit() {
  expr $1 \* 2
}
doubleit 3    # returns 6
```

function declaration and calling syntax

```
for i in *
do
  echo "File is $i"
done
```

A for loop iterates through its input (which is subject to globbing)

Useful Variables

$IFS	Internal File Separator
$?	return code from last program
$SHELL	what shell is running this script?
LANG	Language; C is US English

Test Operators

```
if [ "$x" -lt "$y" ]; then
  # do something
fi
```

Numeric Tests

lt	less than
gt	greater than
eq	equal to
ne	not equal
ge	greater or equal
le	less or equal

File Tests

nt	newer than
d	is a directory
f	is a file
r	readable
w	writeable
x	executable

String Tests

=	equal to
z	zero length
n	not zero length

Logical Tests

&&	logical AND
||	logical OR
!	logical NOT

Argument Variables

$0	program name
$1	1st argument
$2	2nd argument
...	...
$9	9th argument
$*	all arguments
$#	No. of arguments

Variable Substitution

${V:-def}	$V, or "def" if unset
${V:=def}	$V (set to "def" if unset)
${V:?err}	$V, or "err" if unset

Conditional Execution

c1 || c2	run c1; if it fails, run c2
c1 && c2	run c1; if it works, run c2

Common utilities and switches

ls -lSr	list files, biggest last
ls -ltr	list files, newest last
ls -lh	human-readable filesizes
du -sk *	directory sizes (slow)
sort -n	sort numerically (not alpha)
ps -ef	list my commands
wget URL	download URL
time cmd	stopwatch on `cmd`
touch file	create file
read x	read "x" from keyboard
cmd | \ tee file.txt	cmd output to stdout and also to file.txt
nice cmd	run cmd with low priority

Networking

ifconfig -a	list all network interfaces
netstat -r	show routers
ssh u@host	log in to host as user "u"
scp file.txt \ u@host:	copy file.txt to host as user "u"

General Admin

less file	display file, page by page
alias l='ls -l'	create "l" as alias for "ls -l"
tar cf t.tar \ list_of_files	create a tar archive t.tar from the listed dirs/files
cal 3 1973	display a calendar (Mar 73)
df -h	show disk mounts
truss -p PID	show syscalls of PID

Files: Contents / Attributes

find . -size 10k -print	files over 10Kb
find . -name "*.txt" -print	find text files
find /foo -type d -ls	ls all directories under /foo
three=`expr 1 + 2`	simple maths
echo "scale = 5 ; 5121 / 1024" | bc	better maths
egrep "(foo|bar)" file	find "foo" or "bar" in file
awk '{ print $5 }' file	print the 5th word of each line
sed s/foo/bar/g file	replace "foo" in file with "bar"

7. Docker

1. Der Workflow: Vom Code zum Container

Dieser Prozess lässt sich in drei Schritten zusammenfassen: **Build, Pull, Run.**

Images verwalten (Die Vorbereitungen)

Bevor etwas läuft, muss das Image existieren – entweder selbst gebaut oder heruntergeladen.

- **`docker build -t name:tag .`**

Baut ein Image aus dem Dockerfile im aktuellen Verzeichnis (`.`). Das `-t` (Tag) gibt dem Kind einen Namen.

- **`docker pull image_name`**

Lädt ein fertiges Image vom Docker Hub herunter (z. B. `docker pull nginx`).

- **`docker images`**

Listet alle Images auf, die aktuell auf deinem Rechner gespeichert sind.

- **`docker rmi image_id`**

Löscht ein Image (Remove Image).

2. Docker Volumes: Das Langzeitgedächtnis

Container sind **ephemer**. Das ist ein schickes Wort für: Wenn du den Container löschst, sind alle Daten darin (Datenbanken, Uploads, Logs) **weg**. Volumes sind die Lösung, um Daten "auszulagern".

Die drei Arten der Speicherung

A. Named Volumes (Der Docker-Standard)

Docker verwaltet den Speicherort selbst (meist irgendwo tief in `/var/lib/docker/`).

- **Syntax:** `-v mein_goldfisch_glas:/app/data`
- **Vorteil:** Performant, sicher und Docker kümmert sich um Backups und Verwaltung.
- **Einsatz:** Datenbanken (PostgreSQL, MySQL).

B. Bind Mounts (Der Entwickler-Liebling)

Du verbindest einen ganz konkreten Ordner von deiner Festplatte mit dem Container.

- **Syntax:** `-v /Users/name/projekte/app:/app`
- **Vorteil:** Änderungen an deinem Code auf dem Host sind **sofort** im Container aktiv (Live-Reload).
- **Einsatz:** Quellcode während der Entwicklung.

C. Anonymous Volumes

Werden erstellt, wenn du nur den Zielpfad angibst (`-v /app/temp`).

- **Nachteil:** Schwer wiederzufinden, wenn der Container weg ist.

3. Die ultimative Kombination (Beispiel)

Nehmen wir an, du willst eine Website entwickeln. Du brauchst:

1. **Port Mapping**, um die Seite im Browser zu sehen.
2. **Bind Mount**, damit dein Code-Update sofort sichtbar ist.
3. **Named Volume**, damit die User-Daten erhalten bleiben.

Bash
```
docker run -d \
  --name meine_super_app \
  -p 3000:3000 \
  -v $(pwd):/app \
  -v app_db_data:/var/lib/mysql \
  meine_app_image
```

Kurze Checkliste für den Befehl oben:

- **`-p 3000:3000`**: "Browser-Port 3000 funkt an App-Port 3000."
- **`-v $(pwd):/app`**: "Spiegle mein aktuelles Verzeichnis in den Container-Ordner /app."
- **`-v app_db_data:...`**: "Speichere die Datenbank-Daten sicher in einem Volume namens app_db_data."

Nützliche Volume-Commands

- `docker volume ls`: Zeigt alle "Festplatten" an, die Docker gerade verwaltet.
- `docker volume inspect <name>`: Verrät dir, wo genau auf deiner echten Festplatte die Daten liegen.
- `docker volume prune`: Löscht alle Volumes, die gerade von keinem Container benutzt

2. Container steuern (Die Action)

Hier erwachen deine Anwendungen zum Leben. Der Befehl `docker run` ist dabei das Schweizer Taschenmesser.

Der `docker run` Befehl

```
docker run [FLAGS] IMAGE [COMMAND]
```

Die wichtigsten Flags für `run`:

- **`-d` (detached):** Der Container läuft im Hintergrund. Dein Terminal bleibt frei.
- **`-p 8080:80` (-p [HOST_PORT]:[CONTAINER_PORT]):** Mapping. Leitet Port 8080 deines Rechners auf Port 80 des Containers um.
- **`--name mein_server`:** Gibt dem Container einen festen Namen statt eines zufälligen (wie *focused_curie*).
- **`-v /host:/container` (volume):** Verbindet einen Ordner auf deinem Rechner mit einem Ordner im Container (wichtig für Datenbank-Daten).
- **`-it` (interactive + tty):** Brauchst du, wenn du interaktiv mit dem Container arbeiten willst (z.B. eine Shell öffnen).

Beispiel:

```
docker run -d -p 8080:80 --name web-server nginx
```

(Startet einen Nginx-Webserver im Hintergrund, erreichbar unter localhost:8080)

3. Laufende Container verwalten (Das Management)

Wenn die Container erst einmal laufen, musst du sie überwachen und kontrollieren.

Befehl	**Zweck**
`docker ps`	Zeigt alle **laufenden** Container an.
`docker ps -a`	Zeigt **alle** Container an (auch die gestoppten).
`docker stop <ID/Name>`	Hält einen laufenden Container sanft an.
`docker start <ID/Name>`	Startet einen gestoppten Container wieder.
`docker rm <ID/Name>`	Löscht einen (gestoppten) Container endgültig.
`docker logs -f <ID/Name>`	Zeigt die Konsolen-Ausgabe des Containers live an (super zum Debuggen!).

4. Insider-Befehle (Für Profis)

Manchmal musst du „in“ den Container hineinschauen oder aufräumen:

- **`docker exec -it <containername> bash`** (oder sh)

Damit öffnest du ein Terminal **innerhalb** eines bereits laufenden Containers. Es ist, als würdest du dich per SSH einloggen.

- **`docker inspect <containername>`**

Gibt dir alle technischen Details (IP-Adresse, Mounts, Konfiguration) im JSON-Format aus.

- **`docker system prune`**

Der „Frühlingsputz“. Löscht alle gestoppten Container, ungenutzten Netzwerke und verwaisten Images auf einmal.

Ein typischer Workflow in der Praxis:

1. **Code schreiben** und Dockerfile erstellen.
2. **`docker build -t meine-app .`** (Image bauen)
3. **`docker run -d -p 3000:3000 meine-app`** (Container starten)
4. **`docker logs -f meine-app`** (Prüfen, ob alles okay ist)
5. Fehler gefunden? **`docker stop meine-app`** -> Code ändern -> Zurück zu Schritt 2.

8. Docker File

1. Build-Argumente (Optional beim Bauen übergeben)
ARG PYTHON_VERSION=3.9-slim
2. Basis-Image mit Variable
FROM python:${PYTHON_VERSION}
3. Metadaten (Gut für die Organisation)
LABEL maintainer="dein-name@example.com" LABEL description="Produktions-Image für meine Python Web-App"
4. Umgebungsvariablen setzen
Verhindert, dass Python .pyc Dateien schreibt und puffert die Ausgabe (loggt sofort)
ENV PYTHONDONTWRITEBYTECODE=1 ENV PYTHONUNBUFFERED=1 ENV APP_HOME=/app
WORKDIR $APP_HOME
5. Abhängigkeiten (Caching-Optimierung)
Wir kopieren NUR die requirements zuerst, damit der 'pip install' Layer nur neu gebaut wird, wenn sich die Abhängigkeiten ändern.
COPY requirements.txt . RUN pip install --no-cache-dir --upgrade pip && pip install --no-cache-dir -r requirements.txt
6. Sicherheits-Check: Einen Non-Root User anlegen
Standardmäßig läuft Docker als 'root'. Das ist ein Sicherheitsrisiko.
RUN useradd -m myuser USER myuser
7. App-Code kopieren (als der neue User)
COPY --chown=myuser:myuser . .
8. Healthcheck (Sagt Docker, ob die App wirklich "lebt")
HEALTHCHECK --interval=30s --timeout=3s
CMD curl -f http://localhost:8080/health || exit 1
EXPOSE 8080 #Port
9. Startbefehl
CMD ["python", "app.py"]

Beispiel

Docker-File Beispiel zum Bauen einer Node.js Applikation innerhalb eines Docker-Containers

```
# Stage 1: Build the application
FROM node:18-alpine as build-stage

# Install pnpm
RUN npm install -g pnpm

# Set the working directory
WORKDIR /app

# Copy the package.json and pnpm-lock.yaml files
COPY source/package.json source/pnpm-lock.yaml ./

# Install dependencies
RUN pnpm install
```

```
# Copy the rest of the application code
COPY source/ ./

# Build the application
RUN pnpm build

# Stage 2: Serve the application
FROM node:18-alpine as production-stage

# Set the working directory
WORKDIR /app

# Copy the built application from the build stage
COPY --from=build-stage /app/.output ./

# Start the application
CMD ["node", "server/index.mjs"]
```

9. Wichtige andere Befehle

9.1. grep

Zweck:
Sucht nach Textmustern (Patterns) in Dateien oder Ausgaben von Programmen.

Typische Verwendung:

- Text in Dateien suchen
- Logs durchsuchen
- Ausgabe von Befehlen filtern

Syntax:

```
grep [OPTIONEN] PATTERN DATEI
```

Beispiele:

```
grep "error" logfile.txt
ps aux | grep firefox
grep -i "test" datei.txt
```

Wichtige Optionen

Option	Bedeutung
`-i`	ignoriert Groß-/Kleinschreibung
`-v`	zeigt Zeilen ohne Treffer
`-n`	zeigt Zeilennummer
`-r`	rekursive Suche in Verzeichnissen
`-l`	zeigt nur Dateinamen mit Treffer
`-c`	zählt Treffer
`-w`	sucht nur ganze Wörter
`-E`	erweitert Regex (egrep)
`-F`	feste Strings (kein Regex)
`-o`	zeigt nur den Treffer
`--color=auto`	markiert Treffer

Beispiele:

Suche ohne Groß-/Kleinschreibung

```
grep -i "linux" datei.txt
```

Rekursive Suche

```
grep -r "main" .
```

Nur Dateinamen anzeigen

```
grep -rl "TODO" .
```

9.2. sed

Zweck:
Stream Editor zum **Bearbeiten von Textströmen** (Ersetzen, Löschen, Einfügen).

Typische Verwendung:

- Text ersetzen
- bestimmte Zeilen löschen
- Dateien automatisch bearbeiten

Syntax:

```
sed [OPTIONEN] 'BEFEHL' DATEI
```

Beispiele:

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

Wichtige Optionen

Option	Bedeutung
`-e`	mehrere Befehle
`-n`	keine automatische Ausgabe
`-i`	Datei direkt ändern (in-place)
`-r`	erweitertes Regex

Häufige Befehle

Ersetzen

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

Nur erstes Vorkommen pro Zeile.

Alle Vorkommen ersetzen

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

Direkt Datei ändern

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

Zeilen löschen

Zeile löschen

```
sed '3d' datei.txt
```

Zeilenbereich löschen

```
sed '3,5d' datei.txt
```

Zeilen mit Pattern löschen

```
sed '/error/d' datei.txt
```

Zeilen anzeigen

Nur bestimmte Zeile

```
sed -n '5p' datei.txt
```

Text einfügen

Vor Zeile einfügen

```
sed '3i TEXT' datei.txt
```

Nach Zeile einfügen

```
sed '3a TEXT' datei.txt
```

9.3. find

Zweck:
Dateien und Verzeichnisse im Dateisystem suchen.

Typische Verwendung:

- Dateien nach Namen finden
- Dateien nach Größe oder Datum suchen
- Aktionen auf Dateien ausführen

Syntax:

```
find PFAD [OPTIONEN] [AKTION]
```

Beispiele:

```
find . -name "test.txt"
find /home -type f
```

Wichtige Optionen

Option	Bedeutung
`-name`	Name der Datei
`-iname`	Name ohne Groß-/Kleinschreibung
`-type`	Dateityp
`-size`	Dateigröße
`-mtime`	Änderungszeit
`-user`	Besitzer

`-perm`	Rechte
`-maxdepth`	maximale Tiefe
`-mindepth`	minimale Tiefe

Dateitypen

Typ	Bedeutung
`-type f`	normale Datei
`-type d`	Verzeichnis
`-type l`	symbolischer Link

Größe

Beispiel	Bedeutung
`-size +10M`	größer als 10 MB
`-size -1M`	kleiner als 1 MB
`-size 100k`	genau 100 KB

Zeit

Option	Bedeutung
`-mtime`	Tage seit Änderung
`-atime`	letzter Zugriff
`-ctime`	Statusänderung

Beispiel:

```
find . -mtime -7
```

Dateien der letzten 7 Tage.

Aktionen

Dateien löschen

```
find . -name "*.tmp" -delete
```

Befehl ausführen

```
find . -name "*.log" -exec rm {} \;
```

9.4. awk

Zweck:
Werkzeug zur **Textanalyse und Datenverarbeitung** (besonders Tabellen).

Typische Verwendung:

- Spalten ausgeben
- Daten filtern
- Berechnungen durchführen

Syntax:

```
awk 'BEDINGUNG {AKTION}' DATEI
```

Grundprinzip

Jede Zeile wird automatisch verarbeitet.

Standard-Trennzeichen: **Whitespace**

Spalten:

Variable	Bedeutung
`$1`	erste Spalte
`$2`	zweite Spalte
`$0`	ganze Zeile
`NF`	Anzahl Felder
`NR`	Zeilennummer

Beispiele

Erste Spalte anzeigen

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

Mehrere Spalten

```
awk '{print $1, $3}' datei.txt
```

Bedingungen

Nur Zeilen mit Bedingung

```
awk '$3 > 100 {print $1}' daten.txt
```

Trennzeichen ändern

```
awk -F ":" '{print $1}' /etc/passwd
```

Berechnungen

Summe berechnen

```
awk '{sum += $1} END {print sum}' zahlen.txt
```

Durchschnitt

```
awk '{sum += $1} END {print sum/NR}' zahlen.txt
```

Wann benutzt man welches Tool?

Tool	Verwendung
`grep`	Text suchen
`sed`	Text automatisch bearbeiten
`find`	Dateien im Dateisystem suchen
`awk`	Daten analysieren und Spalten verarbeiten

Typische Kombinationen

Mit Pipe:

```
cat logfile | grep error
ps aux | grep firefox
find . -name "*.log" | grep error
```

Kurzvergleich

Tool	Stärke
`grep`	schnelles Suchen
`sed`	automatisches Bearbeiten
`find`	Dateisystem durchsuchen
`awk`	strukturierte Daten analysieren

Anhang "Juli Nützlich"

Analyse.sh

```
#!/bin/bash

anzahl() {
    dateien=$(find "$pfad" -maxdepth 1 -type f | wc -l)
    verzeichnisse=$(find "$pfad" -maxdepth 1 -type d | wc -l)
    echo "Anzahl Verzeichnisse: $verzeichnisse"
    echo "Anzahl Dateien: $dateien"
}

groesse() {
    # stat -c zeigt Dateiinformationen an,
    # %s gibt die Dateigröße in Bytes zurück, %n den Dateinamen
    biggest=$(stat -c "%s %n" * | sort -n | tail -n 1)
    smallest=$(stat -c "%s %n" * | sort -n | head -n 1)
    echo "Kleinste Datei: $smallest"
    echo "Größte Datei: $biggest"
}

kategorie() {
    klein=0
    mittel=0
    gross=0

    for datei in "$pfad"/*
    do
        if [ -f "$datei" ]
        then
            size=$(stat -c %s "$datei")

            if [ "$size" -lt 102400 ]
            then
                ((klein++))
            elif [ "$size" -le 1048576 ]
            then
                ((mittel++))
            else
                ((gross++))
            fi
        fi
    done

    echo "kleine Dateien < 100KB: $klein"
    echo "mittlere Dateien < 1MB: $mittel"
    echo "grosse Dateien > 1MB: $gross"
}

if [ $# != 1 ]
then
    echo "Keine Paramenter übergeben"
    exit 1
fi

pfad="$1"

while [ "$auswahl" != 4 ]
do
    echo " bitte wählen:"
```

```
    echo "1) Anzahl der Dateien und Unterverzeichnisse"
    echo "2) Größte und kleinste Datei finden"
    echo "3) Dateien nach größe klassifizieren"
    echo "4) Beenden"

    read auswahl

    case "$auswahl" in
        1)
            anzahl
            ;;
        2)
            groesse
            ;;
        3)
            kategorie
            ;;
        4)
            echo "Programm wird beendet"
            break
            ;;
        *)
            echo "ungueltige Auswahl!"
            ;;
    esac
done
```

Backup.sh

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

BACKUP_SRC=("$1")
BACKUP_DEST="/home/juli/Schreibtisch/backup"
RETENTION_DAYS=7
LOGFILE="/home/juli/Schreibtisch/backup.log"
LOCKFILE="/tmp/backup.lock"
DRY_RUN=false

log() {
    # Tee macht es möglich, die Ausgabe sowohl auf der Konsole
    # als auch in einer Logdatei zu speichern
    echo "$(date '+%F %T') $1" | tee -a "$LOGFILE"
}

cleanup() {
    rm -f "$LOCKFILE"
}

# trap fängt Signale ab, in diesem Fall EXIT, und führt die Funktion
cleanup aus,
# um sicherzustellen, dass die Lockdatei entfernt wird, wenn das
Skript beendet wird,
# egal ob es erfolgreich war oder durch einen Fehler unterbrochen
wurde.
trap cleanup EXIT

[[ -f "$LOCKFILE" ]] && { echo "Already running"; exit 1; }
touch "$LOCKFILE"

# $@ enthält alle Argumente, die an das Skript übergeben wurden.
for arg in "$@"; do
    [[ "$arg" == "--dry-run" ]] && DRY_RUN=true
done

TIMESTAMP=$(date +%F_%H-%M-%S)
TARGET="$BACKUP_DEST/backup_$TIMESTAMP.tar.gz"

log "Starting backup..."

if ! $DRY_RUN; then
    tar -czf "$TARGET" "${BACKUP_SRC[@]}"
else
    log "Dry run: tar -czf $TARGET ${BACKUP_SRC[*]}"
fi

log "Cleaning old backups..."
# find sucht nach Dateien im Backup-Verzeichnis,
# die älter als die angegebene Anzahl von Tagen sind und löscht sie,
# um Speicherplatz freizugeben und die Anzahl der Backups zu
begrenzen.
find "$BACKUP_DEST" -type f -mtime +$RETENTION_DAYS -name "*.tar.gz"
-delete

log "Backup finished."
```

Count.sh

```
#!/bin/bash

if [ $# != 1 ]
then
    echo "Kein Argument übergeben"
    exit 1
fi

DATEI="$1"

# überprüfen, ob die angegebene Datei existiert
if [ -e "$DATEI" ];
then
    echo "Die Datei $DATEI wurde gefunden"
else
    echo "Die Datei $DATEI wurde nicht gefunden"
fi
```

Envinfo.sh

```
#!/bin/bash

# Umgebungsvariablen
echo "User: $USER"
echo "Homeverzeichnis: $HOME"
echo "Skript: $0"
echo "Datum: $(date)"
echo "PC Name: $(hostname)"
echo "Aktuelles Verzeichnis: $(pwd)"
```

List_info.sh

```
#!/bin/bash

# Dateien eines bestimmten Typs auflisten, hier: alle .sh
for file in "$1"/*.sh
do
    [ -e "$file" ] || continue

    if [ -s "$file" ]
    then
        echo "$file"
    fi
done
```