HowTo: Linux

Diese Datei gibt dir einen Überblick über die Grundlagen des Terminals und über die wichtigsten Befehle, die man am ehesten mal braucht.

Inhalt:

1. Begrifflichkeiten und Grundlagen
2. Navigation
3. Dateien und Ordner
    i. Dateien
    ii. Ordner
    iii. Dateien & Ordner suchen
    iv. Text in Dateien suchen
4. Berechtigungen
5. Netzwerk und Internet
    i. Dateien herunter- & hochladen
    ii. SSH
    iii. SCP
6. Prozesse
7. Texteditoren

1. Begrifflichkeiten und Grundlagen

Begrifflichkeiten

Kommandozeile: Ort (→ Kommando*zeile*) der Eingabe von Befehlen.

CLI: Steht für "command line interface" und ist mit der Kommandozeile gleichzusetzen.

Shell: Interpreter-Programm der eingegebenen Befehle. Anwendungen werden im Kontext dieses Interpreters ausgeführt, befinden sich also in einer Hülle (→ engl. *shell*). Innerhalb der Shell kann man z.B. Umgebungsvariablen ändern und zwar ohne, dass andere Programme/Shells das merken. Die erste Shell gabs in UNIX und ist heute meist unter `/bin/sh` zu finden.

Bash: Weiterentwicklung der Ur-Shell. Alternativen sind z.B. ZSH, KSH, Dash, Fish aber auch die PowerShell unter Windows kann man als Shell bezeichnen (hat "Shell" ja schließlich auch im Namen).

Terminal: Gaaaanz früher ein elektronisches Gerät an dem jemand saß und Befehle eingegeben hat. Wird heute als Gerätedatei (z.B. `/dev/tty...`) dargestellt, welche eingehende Daten an die entsprechende Shell weiterleiten.

Terminal-Emulator: Programm, welches das Terminal darstellt. Es kümmert sich also um Schriftart, Farben, Styles, Effekte, etc. Meist sagt man einfach "Terminal" und meint damit einen Terminal Emulator.

Hinweis: Im Folgenden der Datei werden all diese Begriffe für das gleiche verwendet, da der Unterschied im Alltag für die meisten kaum vorhanden ist.

Grundlagen

Genau wie der normale Datei-Explorer befindet sich auch ein Terminal stets in einem Ordner, sodass man z.B. beim öffnen von Dateien darauf achten muss, ob man dazu im richtigen Ordner ist.

Im Terminal gibt man in einer Zeile (daher auch der Begriff Kommando*zeile*) einen oder mehrere Befehle ein.

Von Befehlen und Programmen

Ein Befehl ist dabei entweder ein sogenannter *built-in* Befehl oder der Name eines normalen Programms. Built-in Befehle sind diejenige, welche vom Terminal direkt interpretiert werden, z.B. gibt es auch in der Bash Konstrukte wie `if` oder `while`, was Sprachelemente der Scriptsprache der Bash sind. Man kann aber auch normale Programme starten, ich könnte also den Befehl `firefox` eingeben um den Firefox-Browser zu starten. Manchmal sagt man zu Befehlen auch *command* oder einfach *programm*.

Flags, Parameter und Argumente

Befehle können Parameter/Argumente übergeben bekommen. Üblicherweise wird unterschieden zwischen *flags*, *parametern* und *argumenten*, wobei es keine einheitliche Definition dafür gibt, hier aber eine grobe Kategorisierung: * *flags*: Einfacher Parameter ohne weitere Daten. Beispiel: `-v` gibt bei vielen Tools die Version des tools aus. * *parameter*: Wie Flags nur mit zusätzlichen Daten. Beispiel: `--output=/pfad/zu/datei.txt` spezifiziert bei manchen Programmen die Ausgabe Datei in die etwas geschrieben werden soll. * *argument*: Ist wie ein Parameter, aber ohne Prefix. Beispiel: `wget https://mein-server/datei.txt`, hier ist `wget` der Befehl und `https://mein-server/datei.txt` das argument (in diesem Fall eine URL die heruntergeladen werden soll).

Da es keine wirkliche Definition gibt, sagen viele einfach zu allem "Parameter", weils einfacher ist und technisch unter der Haube eh alles das gleiche ist.

Eingaben und Ausgaben

Viele Programme geben Text aus (man spricht auch von *logging*), welcher direkt im Terminal sichtbar wird. Manche Programme schreiben sowas aber auch in Dateien, das ist ganz unterschiedlich.

Einige Programme können neben Parametern auch Eingaben entgegen nehmen. Der Unterschied zwischen Eingaben und Parametern ist sehr technisch:

Parameter werden von einem Programm als feste Strings verarbeitet, sie ändern sich also nicht während das Programm läuft und werden auch nicht mehr. Eingaben aber werden üblicherweise erst dann getätigt, während das Programm läuft.

Man startet also ein Programm (ggf. mit irgendwelchen Parametern) und dieses erwartet dann eine Eingabe. Diese muss man entweder manuell eingeben, manche können aber auch Eingaben aus anderen Programmen entgegen nehmen.

Technische Aspekte:

Es gibt drei sogenannte *streams*, die hier relevant sind:

- `/dev/stdout` ("standard out"): Über diesen stream werden Ausgaben vom Programm an das Terminal weitergeleitet. Das Tewrminal zeigt einfach alles Stumpf an, was es über diesen Stream bekommt.
- `/dev/stdin` ("standard in"): Dieser stream geht in die Gegenrichtung: Fordert mich ein Programm zu einer Eingabe aus, gebe ich also Text ein, drücke ggf. Enter und der Text wird vom Terminal in diesen stream geschrieben. Das Programm liest dann auf der anderen Seite diesen stream aus und verarbeitet dann meine Eingabe.
- `/dev/stderr` ("standard error"): Ähnlich wie `/dev/stdout`, aber für Fehlermeldungen gedacht und auch diese werden vom Terminal unverändert angezeigt

Beispiel: `cat datei.txt | grep moin`

Zunächst `cat datei.txt`: Wie wir später erfahren liest dieser Befehl den kompletten Inhalt der Datei `datei.txt` und schreibt ihn in `/dev/stdout` oder in anderen Worten: Zeigt den kompletten Inhalt von `datei.txt` im Terminal an.

Der Zweite Befehl ist `grep moin`: Hier wird jede eingelesene Zeile nach `moin` durchsucht. Taucht `moin` nicht auf, wird die Zeile ignoiert. Man filtert also nach dem String `moin`.

Den `grep` Befehl schauen wir uns aber später noch an. Spannend bei `grep` ist, dass die zu durchsuchenden Daten als Eingabe erwartet werden. Man muss also den Inhalt von `datei.txt` von `/dev/stdout` vom `cat` Befehl irgendwie umleiten nach `/dev/stdin` vom `grep` Befehl. Das ist mit einer sogenannten *pipe* möglich, also dem senkrechten Strich `|`. Dieser sagt so viel wie: Leite die Ausgabe vom linken Befehl (also `cat`) um in die Eingabe vom rechten Befehl (also `grep`).

Historie und Autovervollständigung

Die meisten Terminals bieten eine Historie an zuletzt eingegebenen Befehlen an. Einfach die Pfeiltaste nach oben drücken um durch die Historie zu gehen.

Meistens wird auch Autovervollständigung unterstützt. Dazu einfach den Beginn eines Befehls eingeben und dann

- einmal die Tab-Taste: Versucht eine Autovervollständigung des Befehls. Gibt es eine mögliche Vervollständigung wird diese eingefügt, ansonsten passiert erst mal nichts.
- ein zweites mal Tab-Taste (nur wenn es zuvor kein eindeutiges Ergebnis gab): Zeigt alle möglichen Autovervollständigungen an.

2. Navigation

Die Shell befindet sich immer in *einem* Ordner (wie man es vom normalen Datei-Explorer kennt).

- `cd pfad` oder `cd pfad/zu/einem/ordner` wechselt in den eingegebenen Ordner
- `cd` oder `cd ~` wechselt in das eigene Home-Verzeichnis (quasi der "Eigene Dateien" Ordner unter Linux).
  - `cd ~/ordner` wechselt in den Ordner `ordner` im Home-Verzeichnis
- `cd /` wechselt in den Root-Ordner (den aller obersten Ordner; vergleichbar mit `C:\` unter Windows)
- `pwd` steht für "print working directory" und zeigt den Ordner an in dem man sich gerade befindet

3. Dateien und Ordner

Dateien

Erstellen

- `touch datei.txt` erstellt man eine leere Datei `datei.txt`. Details: Die neue Datei hat die Rechte `rw-r--r--` (s.u. Beispiel zu `ls` und Abschnitt über Berechtigungen).

Lesen

- `cat datei.txt` : Zeigt den kompletten Inhalt von `datei.txt` auf einmal an. Will man vielleicht bei größeren Dateien nicht machen.
- `less datei.txt` : Zeigt den Inhalt an aber man kann blättern (Pfeiltasten oder Bild-rauf/Bild-runter, mit "q" beenden). Bei großen Dateien empfohlen.

Filtern

Das tool `grep` wird häufig zum Filtern von Ausgaben genutzt. Es arbeitet Zeilenweise, schaut sich also jede Testzeile an und prüft, ob die Zeile ausgeschlossen werden soll oder nicht.

- `cat datei.txt | grep -i -n moin` : Gibt alle Zeilen von `datei.txt` aus in denen "moin" vorkommen.
  - `-i` : Case-**i**nsensitive, behandlte also "moin", "Moin", "MOIN", etc. gleich
  - `-n` : Gebe die Zeilen**n**ummer mit aus

Hinweis: Das `|` Zeichen wFird auch *Pipe* genannt, weil die Ausgabe von `cat` gleichzeitig die Eingabe von `grep` ist. Die Ausgabe von `cat` wird also an `grep` weitergeleitet, man sagt auch "reingepiped".

Schreiben

- Operator `>` : Mit `echo "foo" > datei.txt` schreibt man den string `foo` in die Datei `datei.txt` und löscht dabei alles bestehende in der Datei
- Operator `>>` : Mit `echo "foo" >> datei.txt` hängt man den string `foo` am Ende der Datei an

Man kann Befehle verbinden: `cat datei.txt >> andere-datei.txt` hängt den Inhalt von `datei.txt` am Ende von `andere-datei.txt` an.

Auch `grep` funktioniert damit: `cat datei.txt | grep moin > moin.txt` schreibt alle Zeilen von `datei.txt` in `moin.txt` , die den String "moin" enthalten.

Kopieren

- `cp foo.txt anderer/ordner/bar.txt` kopiert die Datei `foo.txt` in den Ordner `anderer/Ordner/` wo sie dann `bar.txt` heißt
- `cp foo/* bar` kopiert alle Dateien aus dem Ordner `foo` in den Ordner `bar`
- `cp foo/*.pdf bar` kopiert nur PDFs von `foo` nach `bar`

Umbenennen & verschieben

- `mv foo.txt bar.txt` benennt `foo.txt` in `bar.txt` um
- `mv foo.txt anderer/ordner` verschiebt `foo.txt` nach `anderer/ordner/foo.txt`

Löschen

- `rm datei.txt` löscht die Datei `datei.txt`, wenn der Nutzer das Schreibrecht (`w`) auf der Datei hat

Ordner

Erstellen

- `mkdir mein-ordner` erstellt man einen leeren Ordner namens `mein-ordner` mit den Rechten `drwxr-xr-x`.

Inhalt auflisten

- `ls` listet alle Dateien und Ordner auf
  - `ls -a` listet auch versteckte Dateien und Ordner auf (diese beginnen mit einem Punkt)
  - `ls -l` stellt das dann auch wirklich als Liste mit Metainfos dar
  - `ls -alh` verbindet beides und zeigt Größen in lesbarem Format an (z.B. `2,3G` statt `2368641604`)

Beispiele siehe oben.

Kopieren

- `cp -r foo anderer/ordner/` kopiert den Ordner `foo` samt Inhalt nach `anderer/ordner`. Der Pfad der Kopie ist also `anderer/ordner/foo`.

Umbenennen & verschieben

- `mv ordner anderer-ordner` benennt den Ordner `ordner` in `anderer-ordner` um
- `mv ordner anderer/pfad` verschiebt den Ordner `ordner` samt Inhalt nach `anderer/pfad/ordner` (kein `-r` oder so nötig)

Löschen

- `rmdir ordner` und `rm -r ordner` löschen den Ordner `ordner` samt darin enthaltenen Dateien und Unterordnern. Nutzer muss Schreibrecht auf `ordner` haben.

Dateien & Ordner suchen

Tool der Wahl um Dateien und Ordner zu suchen/finden: `find`

- Wie finde ist Datei `foo.txt` ? → `find -name foo.txt`

- Wie finde ich Datei `<irgendwas>wodafon-rechnung<irgendwas>` ? → `find -name "*wodafon-rechnung*"`
- Wie finde ich alle PDF-Dateien im Ordner `foo` ? → `find foo -name "*.pdf*`
- Wie finde ich Ordner `bar` ? → `find -type d -name bar`

Text in Dateien suchen

- In welchen Dateien kommt der Text `wodafon ist doof` vor? → `grep -Hirn "wodafon ist doof"`
- In welchen Dateien im Ordner `foo` kommt der Text `wodafon ist doof` vor? → `grep -Hirn "wodafon ist doof" foo`
- In welchen Dateien kommt das Muster `<irgendwas>bar<vier-Ziffern>` also vor? → `grep -HirnP "bar[0-9]{4}"`

4. Berechtigungen

Hinweis: Dieser Abschnitt ist recht technisch und für Anfänger nicht zwingend notwenig. Er passt hier thematisch aber ganz gut hin.

Linux kennt drei Berechtigungen: * Leserechte (`r`): Bestimmt, ob man Datei- oder Ordnerinhalt lesen darf * Schreibrechte (`w`): Bestimmt, ob man in Datei oder Ordner schreiben darf * Ausführungsrechte (`x` wie in e**x**ecute): * Bei Dateien: Bestimmt, ob diese Datei als Programm ausgeführt werden darf * Bei Ordnern: Bestimmt, ob man den Ordner betreten darf

Darstellung

Normalerweie schreibt man eine Berechtigung als `rwx` oder `r-x` oder ähnliches.

Manchmal sieht man auch Zahlenfolgen wie 604 oder ähnliches. Jede Ziffer steht für ein Dreierbündel aus `rwx` :

Zahl	`rwx` -Pendant
0	`---`
1	`--x`
2	`-w-`
3	`-wx`
4	`r--`
5	`r-x`
6	`rw-`
7	`rwx`

Also steht `754` für `rwxr-xr--` oder in Worten "Ich darf alles, Gruppenmitglieder dürfen lesen und ausführen und der Rest nur lesen".

Beispiele

```
$> ls -alh datei.txt
-rwxrw-r-- 1 foo bar 3,9K 13. Nov 13:58 datei.txt
```

- gehört Nutzer `foo` und Gruppe `bar`
- ist 3,9 KB groß
- wurde am 13.11.2021 um 13:58 das letzte mal geändert
- das `-` ganz am Anfang sagt "dies ist eine Datei" (bei Ordnern steht da `d` für **d**irectory)
- das `rwxrw-r--` sagt folgendes:
  - Der Nutzer `foo` hat Lese- (`r`), Schreib- (`w`) und Ausführungsrechte (`x` wie e**x**ecute)
  - Mitglieder der Gruppe `bar` haben Lese- und Schreibrechte (`rw-`)
  - Alle anderen haben nur Leserechte (`r--`)

```
$> ls -alh ordner
insgesamt 12K
drwxr-xr-x  2 foo bar 4,0K 27. Nov 23:11 .
drwxr-xr-x 65 foo bar 4,0K 27. Nov 23:11 ..
-rwxrw-r--  1 foo bar 3,9K 13. Nov 13:58 datei.txt
```

Puh, das sieht komplizierter aus. Also gut:

- Insgesamt ist der Ordner (ohne Unterordner!) run 12 KB groß
- Der erste Eintrag gehört zum Ordner `.`. Dieser Ordner ist *immer* der Ordner in dem sich die Shell gerade befindet. Die Datei `./foo.txt` ist also die Datei `foo.txt` im aktuellen Ordner.
- Der zweite Eintrag gehört `..`, was der übergeordnete Ordner ist. Sagen wir der Ordner in diesem Beispiel hat den Pfad `/home/foo/ordner`, dann entspricht `..` dem Ordner `foo`.
- Der dritte Eintrag zeigt unsere Datei vom vorigen Beispiel.
- Das `d` ganz am Anfang der beiden ersten Zeilen heißt "dies ist ein Ordner" (weil Ordner im englischen = **d**irectory)
- Berechtigungen der Ordner (also `drwxr-xr-x`)
  - Benutzer `foo` darf Dateien im Ordner auflisten (`r`), Dateien erstellen (`w`) und den Ordner betreten (`x`)
  - Alle anderen dürfen Dateien auflisten und den Ordner betreten (`r-x`)

Berechtigungen ändern

- `chmod +w datei.txt` bzw. `chmod u+w datei.txt` (`u` = user) erteilt Schreibrechte für den aktuellen Nutzer auf der Datei `datei.txt`
- `chmod g+w datei.txt` (`g` = group) erteilt Schreibrechte für Gruppenmitglieder
- `chmod o+w datei.txt` (`o` = other) erteilt Schreibrechte für alle anderen
- `chmod -R ug+r ordner` erteilt Leserechte für mich und alle Gruppenmitglieder für alle Dateien im Ordner `ordner`
- `chmod 751 datei.txt` verteilt die Rechte `rwxr-xr--` auf der Datei `datei.txt`

5. Netzwerk und Internet

Dateien herunter- & hochladen

- `curl https://meine-website.de/datei.txt` : Lädt die Datei `datei.txt` herunter und gibts den Inhalt in der Konsole aus
- `curl -o foo.txt https://meine-website.de/datei.txt` : Lädt die Datei `datei.txt` herunter und speichert sie als `foo.txt` ab
- `wget https://meine-website.de/datei.txt` : Lädt die Datei `datei.txt` herunter und speichert sie auch als `datei.txt` ab
- `wget -o foo.txt https://meine-website.de/datei.txt` : Lädt die Datei `datei.txt` herunter und speichert sie als `foo.txt` ab

Man kann mit `curl` nicht nur Dinge herunterladen, sondern auch andere HTTP Anfragen stellen:

- `curl -X POST -H "Content-Type: application/json" -d '{"name": "foo"}' https://example.com/api/user` : Stellt einen POST-request und lädt JSON an den angegebenen API Endpunkt hoch

SSH

- `ssh foo@mein-server` : Verbindet sich als Nutzer `foo` per SSH-Protokoll auf den Server `mein-server` (Standardport ist 22) und fragt dich dann dem Passwort. Bei hinterlegtem SSH-Key ist das etwas anderes, das muss man aber manuell konfigurieren.
- `ssh -p 123 foo@mein-server` : Nutzt statt dem Standardport 22 den Port 123

SSH-Keys nutzen

Man kann sich mit einem Kryptografischen Schlüssel an Servern authentifizieren (also statt der klassischen Nutzername/Passwort Kombination). Dazu muss man 1. ein Schlüsselpaar generieren (sofern noch nicht vorhanden) und 2. den öffentlichen Schlüssel auf dem Server hinterlegen.

1. Schlüsselpaar generieren

1. `ssh-keygen -b 4096`

Hinweis: Definitiv sollte hier eine Passphrase (also ein Passwort für die Schlüssel) vergeben werden!

Die Schlüssel liegen nun in `~/.ssh` , also im Home-Ordner im `.ssh` Ordner. Testen ob alles da ist, es sollte ca. so aussehen:

```
$> ls ~/.ssh
id_rsa  id_rsa.pub  key_rsa  key_rsa.pub  known_hosts
```

2. Schlüssel auf Server hinterlegen

Der **öffentliche** Schlüssel wird nun auf dem Server hinterlegt. Ich melde mich später per SSH quasi via des privaten Schlüssels an und der Server weiß dann, ob wirklich ich das bin oder nicht, da nur mein privater

Schlüssel zu meinem öffentlichen gehört.

1. `ssh-copy-id -i .ssh/key_rsa.pub foo@mein-server`: Wie oben ist `foo` der Nutzername auf dem Server `mein-server`. Wichtig hier, dass man die `.pub` Datei nimmt!

3. Testen

1. `ssh foo@mein-server`: Hier sollte man nach der Passphrase des Schlüsselt gefragt werden, nicht nach dem Passwort vom Benutzer `foo`!

SCP

- `scp foo@mein-server:/pfad/zu/datei.txt /lokaler/ordner/`: Lädt via SSH (s.o. mit Nuzter `foo` und Standardport) die Datei `datei.txt` vom Server herunter und speichert sie unter `/lokaler/ornder/datei.txt` ab.
- `scp -r /lokaler/ordner/ foo@mein-server:123/toller/pfad/`: Lädt dem kompletten Ordnerinhalt (`-r` = recursive) von `/lokaler/ordner/` hoch auf den Server nach `/toller/pfad/`. Als Port wird hier 123 genutzt.

6. Prozesse

Prozesse auflisten und System-Auslastung anschauen

`htop`: Quasi wie der Task-Manager in Windows. Bei den größeren Distributionen (z.B. Ubuntu und Ubuntu-Verwandten) direkt mit dabei, ansonsten muss man `htop` nachinstallieren.

Prozess beenden

- `kill 12345` beendet Prozess mit der ID (PID) 12345
- `killall firefox` beendet alle Prozesse des Programms `firefox`

Eventuell mit man dem Befehl ein `sudo` voranstellen (also `sudo kill ...`), je nachdem ob der eigene Nutzer oder jemand anderes (z.B. das Betriebssystem) den Prozess gestartet hat.

7. Texteditoren

`nano`

Mit `nano foo.txt` barbeitet man Datei `foo.txt`. Shortcuts stehen am unteren Bildschirmrand (z.B. `^X` heißt STRG-x).

`vim`

Uff .... ja ne, das erkläre ich jetzt nur grob. Also `vim` arbeitet mit verschiedenen Modi/Zuständen und bedarf daher etwas Übung.

Startet man `vim` ist man im "normal mode", VIM nimmt in diesem Modus nur Befehle entgegen, man kann aber nicht normal in der Datei schreiben. Befehle fangen mit `:` an (also Doppelpunkt, also halt die Shift- und Punkt-Taste drücken) und nützliche sind z.B. folgende: * `:w` schreibt (**w**rite) Änderungen der gerade geöffneten Datei * `:q` beendet VIM und kann nur benutzt werden, wenn keine ungespeicherten Änderungen vorliegen * `:q!` beendet VIM und verwirft ungespeicherte Änderungen * `:wq` speichert alles und beendet VIM * `:e foo.txt` öffnet (**e**dit) die Datei `foo.txt` im aktuellen Fenster.

Drückt man im "normal mode" `i` (also die normale i-Taste) ist man im "insert mode" und kann normale schreiben und mit den Pfeiltasten navigieren. Drückt man dann "ESC" (also die Escape-Taste), wechselt man zurück in den "normal mode".

Vim kann noch den ganzen geilen Scheiß (Makros, Syntax-Highlighting, Autovervollständigung, File-Browser, Tabs, Split-Screen, abgefahren mächtige Standard-Shortcuts und Befehle die so kompliziert sind, dass einem der Kopf explodiert, etc. pp.). Wusstest du, dass man mit `:%s/foo/bar/gc` den string `foo` durch `bar` ersetzen kann und VIM dich bei jeder Ersetzung fragt? Oder, dass man mit `:%s/^\(.*\)\n\1$/\1/` easy peasy alle doppelten Zeilen löschen kann?! ODER, dass man sich mit `map <C-g> :! bash -c "set -e xtrace; gcc -M % \| tr '\\\\ ' '\n' \| sed -e '/^$/d' -e '/\.o:[ \t]*$/d' \| ctags -L - --c++-kinds=+p --fields=+iaS --extras=+q"<CR><CR>` einen Shortcut auf STRG-g definieren kann, der ... der äh ... ähm ... EGAL! Man kann zudem ne MILLIARDE Plugins installieren und VIM damit noch umfangreicher und komplexer machen solange bis es mächtiger ist als ALLE IntelliJ-IDEs ZUSAMMEN!!1! *bösewicht-lache*

Konfiguration der Bash

Die Konfiguration der Bash findet in der Datei `.bashrc` im Home-Verzeichnis statt. Bei Alternativen ist es ähnlich (z.B: bei ZSH ist es `.zshrc`).