Allgemeine Befehle

pwd	in welchem Ordner man sich befindet
ls	Ordnerinhalt ausgeben
ls -a	um auch versteckte Ordner anzugeben
cd Ordnername/	change directory - Ordner wechseln
cd ..	geht eine Ebene nach oben (Ordner)
mkdir Name	erstellt einen neuen Ordner
touch datei.txt	Datei erstellen
rm datei.txt	Datei löschen
mv dateiii.txt datei.txt	Datei umbenennen dateiii-txt -> datei.txt
mv datei.txt ../ Auch möglich: **mv datei.txt ../d.txt**	verschiebt die Datei in den Desktop verschoben und umbenannt
cp datei.txt d.txt	erstellt Kopie der Datei
clear	Konsole leeren
cat datei.txt	gibt die Datei aus

neues Repository anlegen

git init	neues Repository im aktuellen Verzeichnis im Unterordner .git gespeichert ist eventuell ein versteckter Ordner
git status	zeigt, welche Dateien geändert wurden

Commit wiederherstellen

git checkout <commit hash>	stellt Zustand von einem Commit wieder her
git checkout master	springt zurück zum master (stellt den zuletzt gespeicherten Zustand in dem repository wieder her)

Wird benutzt, um sich einen älteren Zustand anzuschauen (obwohl man weiß, wie der Code weiter verarbeitet wurde)

Repository-History ausgeben

git log	zeigt Commit-Historie
git log -p	zeigt mehr Details in der Commit-Historie (was hat sich geändert)
git log --oneline	gibt nur einen kurzen Kommentar an

Commit abändern

git commit --amend	überschreibt letzten Commit Vorsicht: Am besten nur bei lokalen Änderungen

Änderungen anzeigen

git diff	zeigt Änderungen zur staging area (z. working directory <-> staging area)
git diff --cached	zeigt gestagete Änderungen an (zw. staging area <-> Reposiory
git diff <hash1> <hash2>	zeigt Änderungen zw. zwei Commits

Änderungen anzeigen (cont)

git diff <hash1> <hash2> --Date-i.txt	fügt man " -- Datei.txt" dann wird der "diff" Befehl oder andere wie "log" Befehl nur aus dieser Datei gezeigt
git log -- datei.css	mit **-- datei.*** werden alle Dateien genommen, die Datei. heißen (Beispiel datei.txt, datei.css etc.)

git diff <hash1> <hash2> Beispiel:
Mit "git log --oneline" kann man die Historie sehen und nimmt sich dann zwei der linken ID's und fügt diese an die Stelle von hash1 und hash2 (Hash1 sollte am besten der Commit sein, der weiter in der Vergan-genheit liegt)

Änderungen durchführen

git add <file>	Datei in staging area übernehmen
git add <folder>	ganzer Ordner in staging area übernehmen

Änderungen durchführen (cont)

git commit	Commit aus allen
**git commit -m "<message>"	getätigten Änderungen Alternativ auch mit kurzer Commit-Nachricht

Mit "Commit" wird der Zustand des Projektes abgespeichert (quasi "Version")
In Staging Are werden Commits vorbereitet
Änderungen werden zunächst im working directors gelegt und müssen mit "git add" in die staging are gelegt werden
In der staging area sammeln wir alle Änderungen die wir vornehmen möchten
Mit "git commit" speichern wir diese Änderungen auch in das Repository
(Best Practice: nur eine Änderung pro commit (ein Bug, ein neues Feature...)

Änderungen zurücknehmen

Git Reset

git reset	nimmt **alle** Änderungen aus der staging-area
**git reset <datei-name>	nimmt diese Änderung aus staging-area

Git Reset mit Commit

git reset <commit hash>	Löscht alle Commits nach <commit hash>. Alle Änderungen bleiben im working directory bestehen
git reset --hard <commit hash>	Löscht alle Commits nach <commit hash>. Änderungen werden verworfen

HEAD

HEAD	ist der gerade ausgecheckte Commit
HEAD~1	ist der Parent vom gerade ausgecheckten Commit (sprich eins vorher)
HEAD~2	Parent vom Parent vom gerade ausgecheckten Commit

genutzt um z.B.
statt: *git diff 239f87 98237dc*
schreib man: *git diff HEAD~2 HEAD*

Änderungen zwischenspeichern

Git Stash	aktuellen Zustand vom Working directory und der Staging Area zwischenzuspeichern

Beispiel-Workflow: Du speicherst den Zustand im Stash, schaust dir einen alten Commit an, wechselst
wieder zum aktuellen Stand, und liest den alten Zustand aus dem Stash aus

Git Stash ist ein *Last In First Out* (Stack)
git stash (sichert den aktuellen Stand vom working directory)
git stash pop (stellt den letzten Stand vom working directory wieder her)
git stash list (zeigt alle Stashes an)
git diff stash (zeigt den diff des letzten Stashes an)

Änderungen rückgängig machen

git revert <commit hash>	Erstellt einen neuen Commit, der die Änderungen von <commit hash> rückgängig macht

Würde man eine Sache löschen, auf den die anderen Commits aufbauen, dann würde es zu einem Konflikt führen => werden gefragt, ob wir sie wirklich löschen wollen

(mit *git reset --hard* könnte man zu dem letzten commit springen und den aktuellen Löschvorgang beenden)

Datei ignorieren

.gitignore	Liste an Pfaden, die von git ignoriert werden

Änderungen an diesen Dateien haben dann keine Auswirkung auf *git diff, git add*, etc.
.gitignore sollte mit commitet werden

Bsp:
- Datei mit Passwörter oder Liste an Pfaden die ignoriert werden sollen.
- **file/*.text** würde bedeuten, dass alle .text Dateien in dem Ordner file ignoriert werden sollen

https://github.com/github/gitignore ist eine Sammlung von möglichen .gitignore Dateien. (Aufpassen die Datei in.gitignore umbenennen)

Datei löschen und umbenennen

git rm <datei>	löscht die Datei aus working directory und staging area
git mv <alter Dateiname> <neuer Dateiname>	bewegt die Datei

um Datei richtig zu löschen (working directory und staging area) -> *git rm datei.txt*

Datei umbenennen:
- *mv index.html index2.html* würde dazu führen, dass die Datei nur kopiert wird und damit die ganze Historie von index.html ebenfalls verschwindet
- richtig: *git mv index.html index2.html

Git blame

git blame <filename>	zeigt Zeile für Zeile an, in welchem Commit diese zuletzt geändert wurde
git blame --color-lines <filename>	gleiche Commits farblich hervorheben

Git blame (cont)

git log -p --<Dateiname>	ist eine Alternative die alles im log Stil zeigt

git blame zeigt, wer welche Zeile geschrieben hat
--color-lines führt dazu, dass gezeigt wird, welche Zeilen in einem commit hinzugefügt/geändert wurden

Branches

Git Branches

git branch	zeigt alle vorhandenen Branches an
git branch <branch name>	legt einen neuen Branch an
git checkout <branch name>	wechselt zu einem Branch
Shortcut: **git checkout -b <branch name>**	legt einen neuen Branch an, und wechselt direkt dorthin
git log --online --graph --branches	gibt auch die Branches aus

"Branches" meint eine Verzweigung. Einer arbeitet an feature1 der andere an feature2, oder es wird in einer neuen branch ein Bug gefixt
Wenn dann ein feature fertig ist, wird es an der master-branch gelegt

Der * bei git branch zeigt an, in welchem branch man sich gerade befindet
Genauso **git log --online --graph --branches** zeigt HEAD -> auf welchem Zweig man sich im Moment befindet

Branches mergen (Fast-Forward-Merge)

git merge <branch>	Versucht den Branch branch mit dem aktuellen Branch zusammenzuführen

fast-forward Merge:
1) *git checkout master*, da wir von master aus das neue feature holen
2) *git merge feature*, damit holen wir uns das feature
3) *git log --oneline --branches*, müsste jetzt (HEAD -> master, feature) ausgeben
3) git branch -d feature, das feature am besten nach dem Merge löschen
4) *git log --online --branches*, müsste jetzt (HEAD -> master) ausgeben (feature wurde hinzugefügt, jedoch sieht es aus, als hätte es ihn nie gegeben)

Branches mergen (Three-Way-Merge)

Befehle sind die selben wie bei Fast-Forward-Merge

Wenn kein Fast-Forward-Merge möglich ist, da in Master ebenfalls Änderungen statt fanden

Three-Way-Merge
1) *git log --oneline --branches --graph*, Linien zeigen, was letzter gemeinsame Commit war & wie sich Pfad aufgeteilt hat
2) *git checkout master*, um in master zu wechseln
3) *git merge <feature>*, git erkennt, dass Three-Way-Merge gemacht werden muss und merged automatisch (Best Practice: Die Zeile "Merge branch 'feature' " so lassen, Zusatzkommentar ist möglich

Merge-Konflikte beheben

selben Befehle wie bei den Merge
1) Meldung, dass man den Konflikt manuell prüfen soll
2) wird gezeigt, welche Zeilen von welchem Commit kam
3) Code anpassen, Vorsicht!: Die Zeilen >>> HEAD, ... müssen gelöscht werden

Branch aktuell halten

git rebase <branch>	baut Commit-Historie basierend auf anderen Branch auf
git rebase -i <branch>	erlaubt es, dabei: • Commits anzupassen (edit) • Commit-Messages zu editieren (fixup) • mehrere Commits zusammenzufassen (squash)

Die features werden nicht an dem älteren Branch, sondern an dem aktuellsten Branch gesetzt

Merge-Strategien

git merge --no-ff	Verhindert, dass git einen fast-forward-Merge durchführt. Es wird also immer ein Merge-Commit erstellt.

Merge-Strategien (cont)

git merge --ff-only	Erzwingt einen fast-forward Merge

NUR FAST-FORWARD-MERGE
• Repository ist „in einem Strang“
• Dadurch einfach zu sehen, wie Änderungen aufeinander aufbauen

NUR THREE-WAY-MERGE
• Jedes Feature in eigenen Branch entwickelt
• einfach zu sehen, wie welches Feature entwickelt wurde

Git tag

git tag -a v<x.y>	neuen Tag für Version x.y erstellen
git tag	alle bestehenden Tags anzeigen
**git tag v<x.y> <commit>	altem Commit eine Versionsnummer geben

Mit *git tag* kann man einen Release einer neuen Version markieren

bei *git log* sieht man dann den *tag*

GitHub

Remote-Repository clonen

git clone <url>	erstelltlokale Kopie des master-Branches eines Repositories
git remote -v Ausführlicher: **git remote -vv**	zeigt, welche remote Repositories getrackt

clone:
1) git clone https://...
master ist dann unser lokales Repository
origin/master ist unser remote Repository

Remote-Workflow: fetch & pull

git fetch	lädt Commits und Branches von remote Repository herunter
git log origin/<branch> **git merge origin/<branch>**	lädt Commits und Branches von remote Repository herunter
git pull	Shortcut beider Befehle (man muss in master sein)

zeigt, ob im Repository Änderungen vorgenommen wurden
- *git fetch*
oder:
- *git log -- oneline --graph --branches* zeigt den aktuellen Stand des lok Repository
git log -- oneline --graph --branches origin/master zeigt den akt Stand des remote Repositories an

um es zu mergen: *git merge origin/master*, Shortcut: *git pull*

Einstieg in Github - Code auf GitHub laden

1) Repository erstellen
2) *git remote add origin https:// link zum repository*
3) *git push -u origin master

Änderungen zu GitHub übertragen

1) *git status* => Ausgabe sollte sein:
Auf Branch master
Ihr Branch ist auf demselben Stand wie 'origin/master'
2) Bearbeite die Datei
3) *git add datei.txt*
4) *git commit*
5) *git push* (wenn jemand aber den origin/master ändert, dann kommt es zu Konflikten -> Pull-Requests)

Cheatography

Git und GitHub Cheat Sheet

by Ghost 21 via cheatography.com/151928/cs/38489/

Änderungen übernehmen (Pull-Requests)

1) *git status* wenn nicht auf akt Stand: *git checkout master* & *git pull*
2) *git branch more-content* & *git checkout more-content* (erstellt neue Branch und wechselt rein)
3) Datei bearbeiten & *git add datei.txt* & *git commit*
4) *git checkout master* & *git pull*
5) *git checkout more-content*

...

6) *git rebase master*
7) *git push* -> Fehler, wenn more-content keinen Upstream-Branch => *git push --set--upstream origin more-content* => Jetzt Änderungen in **neuen** Branch
8) oben rechts auf *Compare & pull request* -> Änderungen von more-content in master gemerged
9) *git checkout master* & *git pull*(Aktuelle auch Lokal)

GitHub Workflow: Readme & Markdown--Dateiformat

README.md
Überschrift 1.Ordnung
Überschrift 2.Ordnung (bis 6)

man kann die Wörter dick, kursiv, etc. formatieren

Listen erstellen: mit Spiegelstrichen, oder nummeriert

[link] (https:// wo führt der Link hin)

GitHub Workflow: Issues & Pull-Requests

Genutzt, um zu schreiben, was geändert werden soll oder welcher Bug gefixt werden sollen

Assignes und Labels können angepasst werden

wie löse ich issues:
git branch bg-fix, *git checkout bg-fix*, Datei

GitHub Workflow: Collaborator hinzufügen

Wie füge ich jemanden zum Repository hinzu:
Settings - Manage access - Invite a collaborator

Um auf Master nicht mehr direkt commiten zu dürfen:
Settings - Branches - (Branch protection rules) Add rule - Branch name pattern: "-master" - Require pull request reviews before merging ...

Fork & Pull-Requests; Fork Rebasen

Git Bisect

Verwendung:
git bisect start
git bisect bad [commit-id]
git bisect good [commit-id]
git bisect reset

Häufigsten Befehle

Häufigsten Befehle (cont)

7) `git commit -m " Nac hri - cht "`	commite die geände Dateien
8) `git push` eventuell: `git push origin br anchname`	pusht in Reposit

ändern, *git add datei.txt*, *git commit (mit "- Fixes #Nummer" angeben, welches Issues gefixt*, *git push* / *... --set-upstream...*, Link kopieren um den Pull Befehl zu machen

Wörter wie Fixes, Resloves, etc können benutzt werden

1) `git clone https://` 1) `git pull`	klont das git Repository in den lokalen Ordner falls git clone schon ausgeführt
2) `git status`	zeigt ob geändert
3) `git add Datein ame n. txt` 3) `git add .`	geänderte Datei zur Staging Area alle geänderten Datei zur Staging Area
4) `git status`	nochmal überprüfen
5 `git branch branchname`	erstellt Branch mit dem Namen "- Branch-name"
6) `git checkout branchna me`	Wechsel zur Branch