TOWER


# GIT CHEAT SHEET

presented by Tower - the best Git client for Mac and Windows

## REPOS ERSTELLEN

Clonen eines bestehenden Repos

```
$ git clone ssh://user@domain.com/repo.git
```

Neues, lokales Repo erstellen

```
$ git init
```

## LOKALE ÄNDERUNGEN

Veränderte Files in der Working Copy

```
$ git status
```

Änderungen an versionierten Files

```
$ git diff
```

Alle lokalen Änderungen zum nächsten Commit hinzufügen

```
$ git add .
```

Teile der Änderungen in <file> zum nächsten Commit hinzufügen

```
$ git add -p <file>
```

Alle lokalen Änderungen an bereits versionierten Files direkt committen

```
$ git commit -a
```

Zur Staging Area hinzugefügte Änderungen committen

```
$ git commit
```

Den letzten / neuesten Commit ändern
*Keine bereits veröffentlichten Commits ändern!*

```
$ git commit --amend
```

## COMMIT LOG

Alle Commits chronologisch anzeigen

```
$ git log
```

Änderungen an einem speziellen <file> über versch. Versionen hinweg anzeigen

```
$ git log -p <file>
```

Wer veränderte was & wann in <file>

```
$ git blame <file>
```

## BRANCHES & TAGS

Alle Branches auflisten

```
$ git branch -av
```

Aktuellen HEAD-Branch wechseln

```
$ git switch <branch>
```

Neuen lokalen Branch erstellen (basierend auf dem aktuellen HEAD)

```
$ git branch <new-branch>
```

Neuen lokalen Branch auf Basis eines <remote/branch> erstellen

```
$ git checkout --track <remote/branch>
```

Lokalen Branch löschen

```
$ git branch -d <branch>
```

Tag auf aktuellem Commit erstellen

```
$ git tag <tag-name>
```

## UPDATE & PUBLISH

Alle konfigurierten remote Repos listen

```
$ git remote -v
```

Infos über ein <remote> anzeigen

```
$ git remote show <remote>
```

Weiteres remote Repo anbinden

```
$ git remote add <shortname> <url>
```

Alle Änderungen von <remote> runterladen, aber nicht in HEAD integrieren

```
$ git fetch <remote>
```

Änderungen herunterladen und direkt in HEAD integrieren / mergen

```
$ git pull <remote> <branch>
```

Lokale Änderungen auf Remote pushen

```
$ git push <remote> <branch>
```

Remote Branch löschen

```
$ git push <remote> --delete <branch>
```

Tags veröffentlichen

```
$ git push --tags
```

## MERGE & REBASE

Merge von <branch> in aktuellen HEAD

```
$ git merge <branch>
```

Aktuellen HEAD auf <branch> rebasen
*Kein Rebase von Commits, die published sind!*

```
$ git rebase <branch>
```

Rebase abbrechen

```
$ git rebase --abort
```

Rebase nach Konfliktlösung fortsetzen

```
$ git rebase --continue
```

Konflikt in externem Mergetool lösen

```
$ git mergetool
```

Konflikt als ‹resolved› markieren nach manueller Konfliktlösung im Editor

```
$ git add <resolved-file>
$ git rm <resolved-file>
```

## UNDO

Alle lokalen Änderungen in der Working Copy verwerfen

```
$ git reset --hard HEAD
```

Lokale Änderungen in <file> verwerfen

```
$ git checkout HEAD <file>
```

Commit «rückgängig» machen durch neuen Commit mit gegens. Änderungen

```
$ git revert <commit>
```

HEAD-Zeiger auf einen früheren Commit setzen und ...alle Änderungen seither verwerfen

```
$ git reset --hard <commit>
```

...alle Änderungen behalten

```
$ git reset <commit>
```

...noch nicht committete lokale Änderungen behalten

```
$ git reset --keep <commit>
```

30-day free trial available at
www.git-tower.com

TOWER

TOWER

# VERSION CONTROL

BEST PRACTICES

## THEMATISCH PASSEND COMMITTEN

Ein Commit sollte zusammengehörige Änderungen sammeln. Z.B. sollte das Beheben von zwei unterschiedlichen Bugs in zwei getrennten Commits passieren. Kleine Commits helfen anderen Entwicklern, die Änderungen zu verstehen und sie ggf. rückgängig zu machen falls etwas fehlschlug.
Mit Tools wie der Staging Area und der Möglichkeit, einzelne Teile einer geänderten Datei zu stagen, lassen sich in Git sehr einfach granulare Commits erstellen.

## HÄUFIG COMMITTEN

Oft zu committen hilft dabei, die einzelnen Commits klein und thematisch passend zu halten. Zusätzlich können dadurch die Änderungen öfter veröentlicht werden. Dadurch wird es für alle im Team einfacher, Änderungen regelmäßig zu integrieren und Merge-Konflikte zu vermeiden.
Wenn man hingegen wenige große Commits nur selten verö entlicht, sind Konflikte vorprogrammiert.

## NICHTS HALBFERTIGES COMMITTEN

Man sollte nur fertigen Code committen. Das bedeutet nicht, dass man wochenlang nicht committen sollte, bis ein großes Feature fertig ist. Vielmehr sollte die Implementierung in logische Häppchen aufgeteilt und committet werden. Allerdings sollte man nicht committen, nur um vor dem Feierabend noch etwas im Repo zu haben.
Auch muss man nicht committen, nur um eine saubere Working Copy zu bekommen (um einen Branch zu wechseln etc.). Hierfür ist der «Stash» in Git ideal.

## TESTEN VOR DEM COMMITTEN

Bevor etwas nicht getestet ist, sollte es nicht committen werden. Risiken und Nebenwirkungen sollte man ausführlich testen, um sicherzustellen, dass das Feature wirklich sauber abgeschlossen ist. Vor allem bevor man die eigenen Änderungen mit Teamkollegen teilt, sollte man sicher sein, keinen halbgaren Code committet zu haben.

## GUTE COMMIT MESSAGES

Eine Commit-Message sollte mit einer kurzen Zusammenfassung der Änderungen beginnen (nicht länger als 50 Zeichen). Nach einer Leerzeile sollten dann folgende Fragen durch die Message beantwortet werden:
› Was war der Grund für die Änderung?
› Wie unterscheidet sie sich von der früheren Implementierung?
Mit Imperativ und Gegenwartsform («change» anstatt «changed» oder «changes») bleibt man konform mit automatisch generierten Messages wie z.B. nach «git merge».

## VERSIONSKONTROLLE IST KEIN BACKUP SYSTEM

Ein schöner Nebenefekt der Versionskontrolle ist, dass man ein Backup seiner Files auf einem Remote-Server hat.
Dennoch sollte man sein VCS nicht benutzen als sei es ein Backup-System.
Bei der Versionskontrolle sollte man darauf achten, thematisch passende Änderungen zusammenzufassen (s.o.) - und nicht einfach nur irgendwelche Dateistände zusammen zu würfeln.

## BRANCHES VERWENDEN

Einfaches und schnelles Branching war von Anfang an eine zentrale Anforderung an Git. Und tatsächlich sind Branches eines der besten Features in Git: sie sind das perfekte Tool, um verschiedene Kontexte im Entwicklungsalltag sauber getrennt zu halten. Moderne Workflows sollten Branches intensiv nutzen: für neue Features, Bugfixes, Ideen oder Experimente.

## EINHEITLICHER WORKFLOW

Git ermöglicht die verschiedensten Workflows: langlebige Branches, themenbasierte Branches, Merge oder Rebase, git-flow, ...Wofür man sich entscheidet, hängt von verschiedenen Faktoren ab: dem Projekt, den generellen Entwicklungs und Deployment-Workflows und (vielleicht am wichtigsten) auch von den persönlichen Präferenzen und denen des Teams. Aber egal, für welche Arbeitsweise man sich entscheidet, gilt immer: alle Teammitglieder sollten sich auf einen gemeinsamen Workflow einigen und ihn einhalten.

## HILFE & DOKUMENTATION

Hilfe auf der Kommandozeile

```
$ git help <command>
```

## KOSTENLOSE INFOS IM WEB

http://www.git-tower.com/learn
http://rogerdudler.github.io/git-guide/
http://www.git-scm.org/

30-day free trial available at
www.git-tower.com

TOWER