fournova

GIT CHEAT SHEET

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

CREATE

Clone an existing repository

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

Create a new local repository

```
$ git init
```

LOCAL CHANGES

Changed files in your working directory

```
$ git status
```

Changes to tracked files

```
$ git diff
```

Add all current changes to the next commit

```
$ git add .
```

Add some changes in <file> to the next commit

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

Commit all local changes in tracked files

```
$ git commit -a
```

Commit previously staged changes

```
$ git commit
```

Change the last commit

Don‘t amend published commits!

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

COMMIT HISTORY

Show all commits, starting with newest

```
$ git log
```

Show changes over time for a specific file

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

Who changed what and when in <file>

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

BRANCHES & TAGS

List all existing branches

```
$ git branch -av
```

Switch HEAD branch

```
$ git checkout <branch>
```

Create a new branch based on your current HEAD

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

Create a new tracking branch based on a remote branch

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

Delete a local branch

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

Mark the current commit with a tag

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

UPDATE & PUBLISH

List all currently configured remotes

```
$ git remote -v
```

Show information about a remote

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

Add new remote repository, named <remote>

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

Download all changes from <remote>, but don‘t integrate into HEAD

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

Download changes and directly merge/integrate into HEAD

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

Publish local changes on a remote

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

Delete a branch on the remote

```
$ git branch -dr <remote/branch>
```

Publish your tags

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

MERGE & REBASE

Merge <branch> into your current HEAD

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

Rebase your current HEAD onto <branch>

Don‘t rebase published commits!

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

Abort a rebase

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

Continue a rebase after resolving conflicts

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

Use your configured merge tool to solve conflicts

```
$ git mergetool
```

Use your editor to manually solve conflicts and (after resolving) mark file as resolved

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

UNDO

Discard all local changes in your working directory

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

Discard local changes in a specific file

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

Revert a commit (by producing a new commit with contrary changes)

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

Reset your HEAD pointer to a previous commit ...and discard all changes since then

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

...and preserve all changes as unstaged changes

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

...and preserve uncommitted local changes

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

fournova

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.

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.

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.

HÄUFIG COMMITTEN

Oft zu committen hilft dabei, die einzelnen Commits klein und thematisch passend zu halten. Zusätzlich können dadurch die Änderungen öfter veröffentlicht 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 veröffentlicht, sind Konflikte vorprogrammiert.

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».

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.

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.

VERSIONSKONTROLLE IST KEIN BACKUP SYSTEM

Ein schöner Nebeneffekt 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.

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/