Git – ein ~~kurzes~~ Tutorial

Dieses Tutorial bietet eine Einführung in die grundlegenden Funktionen von Git.

Was ist VCS?

- Version Control System
- VCS kann Änderungen an Software erfassen und verwalten
- es gibt drei Arten von Versionsverwaltung
    - Lokale Versionsverwaltung
    - Zentrale Versionsverwaltung
    - Verteilte Versionsverwaltung

Was ist Git?

- Git ist eine freie Software zur *verteilten Versionsverwaltung*
- Git wurde vom *Linus Torvalds*, dem Erfinder von Linux initiiert

Was kann Git?

- gemeinsames Arbeiten an Software-Projekten
- Änderungen können verfolgt, überprüft und ggf. rückgängig gemacht werden
- Jede*r Entwickler*in hat ein vollständiges "Backup" des Codes auf dem eigenen Rechner
- verschiedene Entwicklungsrichtungen können voneinander unabhängig in *Branches* organisiert werden
    - z.B.: eine *Master* und eine *Dev Branch*.
- *Repositories* können als *Remotes* auf Servern liegen
    - z.B. bei *GitHub*, *GitLab* oder einem eigenen Server

Wie kann ich Git nutzen?

- auf der Kommandozeile / im Terminal
- mit einer IDE oder einem Editor:
    - Atom (kostenlos)
    - IntelliJ (kostenlos mit kostenpflichtiger Pro-Version)
    - NetBeans (kostenlos)
- mit grafischen Tools:
    - GitHub Desktop (kostenlos)
    - GitKraken (kostenlos mit kostenpflichtiger Pro-Version)

Git auf der Kommandozeile / im Terminal

Warum soll ich Git auf der Kommandozeile oder im Terminal lernen?

Die Möglichkeiten von Git auf der Kommandozeile oder dem Terminal, unterscheiden sich nicht von denen, die grafische Anwendungen bieten, da dieselben Befehle verwendet werden.

Grafische Anwendungen unterscheiden sich jedoch in der Menüführung stark voneinander, sodass man keine allgemeingültige Einführung über ihre Verwendung geben kann. Außerdem fassen sie einige der einfachen Git-Befehle zu komplexeren Operationen zusammen, sodass die Versionsverwaltung zwar bequemer, aber anfangs auch ein bisschen unübersichtlich ist.

Um die Funktionalitäten von Git zu verstehen, ist es daher hilfreich, die Git-Befehle zu kennen. So gewinnt man auch eine Vorstellung davon, wie grafische Anwendungen mit Git arbeiten.

Git einrichten

Um Git verwenden zu können, sollte man mindestens den eigenen Namen und eine E-Mail-Adresse hinterlegen.

```
git config --global user.name "Max Mustermann"
```

```
git config --global user.email max@mustermann.de
```

Ein Repository initiieren

Ein neues *Repository* kann man in einem neuen oder bereits vorhandenen Projektordner anlegen.

```
git init
```

Ein Repository kopieren

Von einem Server (z.B. GitHub) kann man ein Repository auf den eigenen PC kopieren.

```
git clone https://gitlab.marc-michalsky.de/gfn/git_tutorial.git
```

Status des Repositories

Ob Änderungen vorgemerkt wurden und welche Dateien bereits getrackt werden (oder auch nicht), steht im Status des Repositories.

```
git status
```

Dateien hinzufügen

Dateien, die noch nicht von Git getrackte werden, können mit `git add` hinzugefügt werden.

```
git add -A  # Fügt alle Dateien im Projektordner hinzu (auch Dateien in
Unterordnern)
```

```
git add .   # Fügt alle Dateien im aktuellen Ordner hinzu (auch Dateien in
Unterordnern)
```

```
git add -p <datei>      # Fügt nur die angegebene Datei hinzu
```

Bestimmte Dateien ignorieren

Möchte man bestimmte Dateien oder Verzeichnisse im Projektordner nicht im Repository verwalten, können sie in einer Datei namens *.gitignore* aufgeführt werden.

/projektordner/.gitignore

```
/out            # Ignoriert das gesamte Verzeichnis "/out"
/.idea          # Ignoriert das gesamte versteckte Verzeichnis "/.idea"
*.xml           # Ignoriert alle Dateien mit der Endung ".xml"
```

Änderungen vormerken

Hat man Dateien verändert, **können** die Änderungen für den nächsten *Commit* vorgemerkt werden. Das nennt man auch *stagen*.

Änderungen werden genau so *gestaged*, wie Dateien am Anfang einen Repository hinzugefügt werden.

```
git add -A  # Fügt alle Änderungen im gesamten Projektordner hinzu (auch in
Unterordnern)
```

```
git add .   # Fügt alle Änderungen im aktuellen Ordner hinzu (auch in
Unterordnern)
```

```
git add -p  <datei>     # Fügt nur die Änderungen in der angegebenen Datei hinzu
```

Änderungen committen

Ein *Commit* benötigt immer eine *Message*, in der kurz zusammengefasst steht, was der *Commit* bewirkt.

Um alle vorgemerkten Änderungen zu *committen*, verwendet man `git commit`. Anschließend wird man aufgefordert, eine *Message* für den *Commit* einzugeben.

```
git commit
```

Die *Message* kann mit `-m` auch direkt an den Befehl angehängt werden.

```
git commit -m "Initial Commit"      # verwendet man häufig für den ersten Commit
```

Man kann auch noch nicht vorgemerkte Änderungen sofort committen. Dafür müssen die Dateien aber bereits von Git getrackt werden (siehe: Dateien hinzufügen).

```
git commit -a
```

Lokale Änderungen anzeigen

Um alle Änderungen anzuzeigen, die seit dem letzten *Commit* vorgenommen wurden, verwendet man:

```
git diff
```

Logs anzeigen

Alle *Commits* werden von Git geloggt. In den Logs steht:

- Wer die Datei geändert hat (Name, E-Mail)
- Wann die Datei geändert wurde
- ein Hash-Wert des Commits

```
git log
```

oder für Änderungen an einzelnen Dateien

```
git log -p <datei>
```

Branches

Die vorhandenen *Branches* anzeigen:

```
git branch -a
```

Eine neue *Branch* hinzuzufügen:

```
git branch <branch-name>
```

Die Branch wechseln:

```
git checkout <branch-name>
```

ACHTUNG: Möchte man zu einer anderen Branch wechseln, dürfen in der aktuellen Branch keine Änderungen mehr gestaged sein! Möchte man Änderungen noch nicht *committen*, die Branch jedoch trotzdem welchseln ist der Stash eine gute Lösung.

Ein Remote hinzufügen

Hat man ein *Repositorie* angelegt, kann man es mit einem *Remote* mit einem Projekt auf einer Plattform (z.B. auf GitHub) verknüpfen. Dazu muss man in der Regel zuerst ein neues Projekt auf der Plattform anlegen.

```
git remote add git_tutorial ssh://git@gitlab.marc-
michalsky.de:10022/gfn/git_tutorial.git
```

Vorhandene Remotes anzeigen:

```
git remote -v
```

Push & Pull

Um *Commits* per *Remote* an einen Server zu übertragen, verwendet man `git push`.

```
git push <remote-name> <branch-name>
```

Möchte man sein lokales Repository auf den aktuellen Stand des Repositories auf dem Server bringen, gibt es zwei Möglichkeiten:

1. Mit `git fetch` werden die Änderungen zwar heruntergeladen, aber noch nicht in die aktuelle *Branch* übertragen. Den *Commit* muss man anschließend selbst vornehmen.

```
git fetch <remote-name>
```

2. Mit `git pull` wird das lokale Repository direkt auf den aktuellsten Stand gebracht.

```
git pull <remote-name> <branch-name>
```

Reset

Es gibt viele Möglichkeiten, um ein Repository auf einen früheren Stand zu bringen. Sie unterscheiden sich vor allem darin, wie weit zurück man gehen möchte und wie mit den in der Zwischenzeit vorgenommenen Änderungen umgegangen wird.

Alle lokalen Änderungen seit dem letzten *Commit* verwerfen:

```
git reset
```

Den Zustand nach einem bestimmten *Commit* wiederherstellen:

Lokale Änderungen werden behalten!

```
git reset <commit-hash>             # Den Hash findet man in den Logs
```

Den Zustand nach einem bestimmten *Commit* wiederherstellen:

Lokale Änderungen werden verworfen!

```
git reset --hard <commit-hash>      # Den Hash findet man in den Logs
```

ACHTUNG: Ein *Reset* verändert die Historie des Repositories und sollte deshalb niemals verwendet werden, wenn die Änderungen bereits mit anderen Entwickler*innen geteilt wurden! Stattdessen sollte Revert verwendet werden.

Revert

Revert bietet die Möglichkeit, Änderungen rückgängig zu machen, ohne die Historie des Repositories zu verändern. *Revert* erstellt einen neuen Commit, der dem gewünschten früheren Zustand entspricht. Die in der Zwischenzeit gemachten Änderungen werden zwar rückgängig gemacht, bleiben jedoch im *Log* erhalten und können so nachvollzogen werden.

```
git revert <commit-hash>
```

Stash

Der *Stash* ist ein Staurraum, in dem Änderungen verstaut werden können, die noch nicht soweit sind, um *commited* zu werden. Dies ist nützlich, wenn man einen bestimmten Bearbeitungsstand speichern möchte oder man die *Branch* wechseln will, aber noch Änderungen vorliegen, die noch nicht *committed* wurden.

Der *Stash* ist unabhängig von den *Branches*!

Der einfache Befehl `git stash push` verstaut alle Änderungen, die seit dem letzten *Commit* vorgenommen wurden im *Stash* und stellt den Bearbeitungsstand des letzten *Commits* wieder her. Mit `-m` kann eine Message angehängt werden.

```
git stash push -m "neue Klasse erstellt"          # Die Message ist optional
```

Um gespeicherte Bearbeitungsstände im *Stash* anzuzeigen verwendet man:

```
git stash list
```

Der Bearbeitungszustand kann aus dem *Stash* geladen werden, indem der gewünschte Eintrag aus der Liste `git stash list` ausgewählt wird.

Der Eintrag im *Stash* bleibt erhalten!

```
git stash apply stash@{0}           # Stellt den Zustand von stash@{0} wieder
her
```

Um den letzten Bearbeitungszustand wiederherzustellen und ihn gleichzeitig aus dem *Stash* zu entfernen, verwendet man:

Der neuste Eintrag im *Stash* wird verworfen!

```
git stash pop
```

Um einen Bearbeitungsstand zu löschen, verwendet man:

```
git stash drop stash@{0}            # Löscht den Eintrag stash@{0}
```

Um alle Einträge im *Stash* zu löschen, verwendet man:

```
git stash clear
```

Merge

Es ist möglich, die Zustände verschiedener *Branches* ineinander zu integrieren.

Eine andere *Branch* in die aktuelle *Branch* integrieren:

```
git merge <branch-name>
```

Wurden in den beiden *Branches*, die *gemerged* werden sollten Dateien an den gleichen Stellen unterschiedlich bearbeitet, kommt es zu einem *Merge-Konflikt*. Die miteinander in Konflikt stehenden Dateien werden unter `git status` als `unmerged` angezeigt. Git hat die Konflikte in den Dateien markiert. Man kann diese nun bearbeiten und auf den gewünschten Stand bringen. Anschließend muss die korrigierte Datei wieder mit `git add` hinzugefügt werden.

Man kann auch ein grafisches Tool verwenden, um die Konflikte aufzulösen:

```
git mergetool
```

Weiterführende Informationen

Dieses Tutorial bietet nur einen groben Überblick über die wichtigesten Funktionen von Git. Viele der genannten Befehle haben zudem noch einige zusätzliche Parameter.

Seiten mit weiterführenden Informationen:

- Offizielle Git Dokumentation
- Git-Cheat-Sheet von Github
- YouTube Tutorials von *Corey Schafer*