Git-Zusammenfassung

Michael Merz *

Version vom 7. Juni 2020

Zu diesem Dokument

Shell-Befehle sind in `Maschinenschrift` geschrieben. Meiner Auffassung nach zunächst weniger wichtige Passagen sind grau hinterlegt. Verpflichtende Parameter sind in `<>` annotiert, optionale Parameter in `[]`. Diese Klammern beim Eintippen der konkreten Parameter weglassen!

In diesem Dokument wird `example.com` als Platzhalter-Domain verwendet. Diese muss durch die Domain des jeweiligen Servers ersetzt werden, den man verwendet.

Inhaltsverzeichnis

*`www.telekobold.de`

[1] `https://creativecommons.org/licenses/by-sa/4.0/deed.de`

1 Einführung

Git ist eine *freie*[2] Software zur *verteilten Versionsverwaltung*[3], die ursprünglich zur Entwicklung des Linux-Kernels geschrieben wurde. Git wird vor allem in der Softwareentwicklung genutzt, kann aber grundsätzlich für die Versionsverwaltung beliebiger Dateien verwendet werden (sollte allerdings nur für textbasierte Dateien verwendet werden, also beispielsweise nicht für Fotos oder Videos). Insbesondere ermöglicht es Git, dass viele Leute gleichzeitig an einem großen Softwareprojekt arbeiten können. Mehr dazu findet sich etwa auf der offiziellen Website der Git-Entwickler[4] oder im Git-Artikel der deutschsprachigen Wikipedia[5].

Dieses Dokument soll die wichtigsten Befehle der Git-Kommandozeilen-Schnittstelle für die alltägliche Arbeit mit Git zusammenfassen und ihre Funktionalität kurz und präzise erläutern. Es richtet sich an Git-Einsteiger und soll die Lücke zwischen einer einzeiligen Erklärung wie in einem Sheat Sheet und einer sehr detaillierten und technischen Erklärung wie in einer Manpage[6] zu einem einzelnen Git-Befehl schließen. Insbesondere soll es Git-Einsteigern dabei helfen, sich schnell mit der Git-Kommandozeilen-Schnittstelle zurechtzufinden. Es ist aus der Perspektive eines Linux-Nutzers geschrieben, der gleichzeitig mit der *GitLab Community Edition*[7] arbeitet, die Befehle und ihre Erklärungen funktionieren aber fast immer genauso beispielsweise auch unter der Git-Bash in Windows. Bei *GitLab* handelt es sich um eine Browser-Oberfläche auf Basis von Git, welche viele Funktionalitäten wie einen *Diff-Log*, eine baumartige Darstellung von *Commits* und Kollaborationsfunktionen wie ein *Wiki* und einen *Issue-Tracker* zur Verfügung stellt und in Kombination mit Git verwendet wird.

2 SSH-Schlüsselpaar generieren

Die Generierung eines SSH-Schlüsselpaars ist optional, aber empfehlenswert, da auf diese Weise eine solide und sichere Verbindung zu Remote-Repositories verwendet werden kann.

Zur Interaktion mit einem *Remote-Repository* (ein mit dem lokalen Git-Repository verknüpftes Git-Repository, welches auf einem entfernten Server liegt) arbeitet Git mit der *Secure Shell* (*SSH*)[8] zusammen. Falls man Git nicht nur lokal verwenden möchte (was in den meisten Fällen der Fall sein wird) und noch kein SSH-Schlüsselpaar hat und/oder ein neues generieren möchte, tut man dies mit `ssh-keygen -t rsa -b 4096`. Dieser Befehl legt das neu generierte Schlüsselpaar unter Linux im Home-Verzeichnis des Nutzers unter `~/.ssh` ab. Näheres dazu findet sich etwa hier[9] oder nach Eingabe von `man ssh-keygen`.

`-t` spezifiziert das Krypto-Verfahren (in diesem Fall *RSA*), `-b` spezifiziert die Schlüssellänge (in diesem Fall 4096 Bit). Bei der Schlüsselerzeugung sollte eine gute™ Passphrase definiert werden. Ohne die Verwendung einer Passphrase hat jede(r), der/die Zugriff auf das entsprechende Nutzerkonto hat, in dem das Schlüsselpaar gespeichert ist, auch Zugriff auf alle Dienste, für die das Schlüsselpaar zur Authentifizierung verwendet wird. Anschließend den Public Key (Ausgabe z.B. mittels `cat ~/.ssh/id_rsa.pub`) im GitLab-Webinterface hinterlegen.

[2] `https://fsfe.org/freesoftware/basics/summary.html`
[3] `https://de.wikipedia.org/wiki/Versionsverwaltung`
[4] `https://git-scm.com/`
[5] `https://de.wikipedia.org/wiki/Git`
[6] `https://de.wikipedia.org/wiki/Manpage`
[7] `https://about.gitlab.com/` , `https://de.wikipedia.org/wiki/GitLab`
[8] `https://de.wikipedia.org/wiki/Secure_Shell`
[9] `https://wiki.ubuntuusers.de/SSH/`

3 Kurze Auflistung und knappe Erklärung der initial wichtigsten Git-Befehle

In der GitLab Community Edition kann man über die grafische Oberfläche neue Projekte erzeugen, welche dann als Git-Projekte geklont werden können. Die jeweilige URL eines Projektes kann jederzeit wahlweise für SSH und für HTTPS in der Projektübersicht eingesehen und durch Drücken auf einen Button in den lokalen Zwischenspeicher kopiert werden. Beispielsweise in GitHub[10] funktioniert die grafische Erzeugung von Projekten ähnlich und es gibt ebensolche Buttons zum Kopieren der URLs, welche zum Klonen des jeweiligen Projektes verwendet werden können. Die URLs sehen in etwa so aus:

- Falls SSH verwendet wird (Public-Key-Authentifizierung, bevorzugte Methode): `git@gitlab.example.com:nutzer/nameDesGitProjektes.git`
- Falls Passwort-Authentifizierung verwendet wird:
  `https://gitlab.example.com/nutzer/nameDesGitProjektes.git`

Die im Folgenden gelisteten Befehle können direkt so und in dieser Reihenfolge ausgeführt werden (Ausführung auf einer lokalen Shell):

1. Mit `cd <Verzeichnisname>` oder grafisch im Dateinamager in das Verzeichnis wechseln, in welches man das Remote-Repository klonen möchte. Git erstellt beim Klonen (`git clone`) ein neues Verzeichnis, es ist daher nicht notwendig, dies (mit `mkdir` oder grafisch im Dateimanager) selbst zu tun. Man sollte nur darauf achten, dass kein lokales Verzeichnis existiert, welches denselben Namen hat wie das Verzeichnis, welches der `git clone`-Befehl erstellen wird.
2. Repository klonen (wird nur einmal gemacht):
   `git clone git@gitlab.example.com:nutzer/nameDesGitProjektes.git`
3. Ggf. Änderungen am lokalen, geklonten Repository vornehmen.
4. Einzelne geänderte Datei mittels
   `git add <DateipfadDerGeaendertenDatei>`
   oder alle geänderten Dateien mittels
   `git add -A` (bzw. `git add --all`)
   zum Commit vormerken.
5. Änderungen ins lokale Repository übernehmen ("committen"):
   `git commit -m ‘‘<Aenderungsnachricht>’’`
6. Vor dem Übertragen neueste Änderungen vom Remote-Repository runterladen, um Konflikte zu vermeiden (falls man sich nicht sicher ist, dass es auf dem Remote-Repository keine solchen Änderungen gegeben hat, z.B. weil man selbst der einzige Contributor ist):
   `git pull`
7. Änderungen ins Remote-Repository schreiben:
   `git push`

Änderungen am Remote-Repository (von anderen Projektteilnehmern oder von einem selbst von einem anderen Rechner dorthin gepusht) werden nun jederzeit mit `git pull` ins lokale Repository übernommen.

Nach Ausführen von `git clone` (oder `git init`) solle man, falls noch nicht passiert, Nutzername und E-Mail-Adresse konfigurieren:

[10] `https://github.com/`, `https://de.wikipedia.org/wiki/GitHub`

`git config user.name ‘‘<Name>’’`
`git config user.email ‘‘<E-Mail-Adresse>’’`

Die hier angegebenen Daten werden mit Commits verknüpft, die man durchführt und sind anschließend z.B. mittels `git log` nachvollziehbar (siehe weiter unten). Dies dient v.A. dazu, dass in Projekten mit mehreren Contributoren jeder Commit einer Person zugeordnet werden kann (und die Person ggf. über ihre E-Mail-Adresse kontaktiert werden kann).

Der Aufruf dieser Befehle ohne weitere Optionen sorgt für eine *lokale* Konfiguration. Mittels der Option `--global` (`git config --global user.name ‘‘<Name>’’` bzw. `git config --global user.email ‘‘<E-Mail-Adresse>’’`) kann man Name und E-Mail *global* (also für alle Git-Projekte auf dem Rechner) konfigurieren. Eine Eingabe ohne Parameter (`git config user.name` bzw. `git config user.email`) führt zu einer Ausgabe von Name und E-Mail-Adresse. Lokale Konfigurationen überschreiben für das lokale Git-Repository die globale Konfiguration, d.h. es ist möglich, für verschiedene Git-Projekte verschiedene Namen und E-Mail-Adressen (und weitere Einstellungen) zu definieren.

4 Auflistung und Erklärung einzelner Git-Befehle

4.1 Befehle, die nur auf lokale Repositories angewendet werden

- `git add <Datei>` stellt `Datei` bzw. die Änderung von `Datei` unter Versionsverwaltung (fügt sie zur *Staging-Area* hinzu) und merkt sie somit zum Committen vor.
  `git add -A` tut dies mit *allen* seit der letzten Ausführung von `git add` vorgenommen Änderungen im lokalen Repository.
  `git add` kann vor einem `git commit` beliebig oft ausgeführt werden.

- `git rm <Datei>` löscht `Datei` und entfernt sie aus dem Index von Git. Diese Änderung muss anschließend noch mit `git add` und `git commit` auf das lokale Repository und ggf. mit `git push` auf das Remote-Repository angewendet werden.
  `git rm --cached <DateiName>` entfernt die zuvor mittels `git add <DateiName>` hinzugefügte `Datei` (bevor `git commit` ausgeführt wurde).

- `git commit -m ‘‘<Aenderungsnachricht>’’` schreibt mittels `git add` hinzugefügte Änderungen ins lokale Repository auf den Branch, auf dem man sich gerade befindet (fügt sie zum *Arbeitsverzeichnis* hinzu). Für `Aenderungsnachricht` (als *Commit-Message* bezeichnet) sollte man eine aussagekräftige Nachricht schreiben, sodass jede(r), der/die sich diese Nachrichten etwa mit `git log` ansieht (insbesondere man selbst zu einem späteren Zeitpunkt), auf einen Blick (grob) erkennt, welche Änderungen durch diesen Commit am Projekt vorgenommen wurden.

- Jeder Commit erhält von Git eine eindeutige ID. Es gibt verschiedene Arten, Commits zu referenzieren:
  - *absolut*:
    - über die *vollständige* Commit-ID (Ausgabe durch `git log`)
    - über die *gekürzte* Commit-ID (Ausgabe durch `git log --oneline`)
  - *relativ*:
    - `HEAD` referenziert den letzten Commit
    - z.B. `HEAD~2` referenziert den vorletzten Commit

    - z.B. `master~2` referenziert den vorletzten Commit auf dem Master-Branch
    - z.B. `master@{4}` referenziert den viertletzten Commit vom aktuellen HEAD aus auf dem Master-Branch

- `git commit --amend -m ''neue Commit-Message''` überschreibt die Commit-Message des letzten Commits auf dem aktuellen Branch, sofern dieser Commit noch nicht auf ein Remote-Repository gepusht wurde. Rückname eines Commits, falls dieser bereits auf ein Remote-Repository gepusht wurde: Siehe `git reset`.

- `git log` zeigt alle bisherigen Commits (die *Commit-Historie*) an, incl. Commit-ID, Commit-Message, Autoren-Name, Autoren-E-Mail-Adresse und Zeitstempel.
  `git log -n` zeigt die letzten n commits an (n muss ein positiver Integer-Wert sein).

- `git log --oneline` gibt einen Präfix der Commit-ID und zusätzlich die Commit-Message in einer Zeile aus. Dieser Befehl eignet sich insbesondere, wenn man zur Referenzierung von Commits z.B. für `git diff` die gekürzten Commit-IDs dieser beiden Commits benötigt.

- `git status` zeigt den aktuellen Status des Arbeitsverzeichnisses an, darunter auf welchem Branch man sich befindet, ob sich der aktuelle Branch vom letzten bekannten Stand des verknüpften Remote-Branches unterscheidet, ob etwas noch nicht versioniert ist oder ob es etwas zu committen gibt. Falls etwas von diesen genannten Zuständen oder ein anderer Zustand gilt, schlägt `git status` einen Befehl vor, wie man möglicherweise weiter verfahren sollte (falls etwas noch nicht versioniert ist z.B., dass man `git add` anwenden könnte).

- `git diff` zeigt Unterschiede zwischen den Versionen von Dateien im Git-Repository und noch nicht unter Versionsverwaltung gestellten Änderungen an diesen Dateien an (bevor diese mit `git add` zur Staging-Area hinzugefügt werden).

- `git diff --cached` zeigt Unterschiede zwischen den Versionen von Dateien im Git-Repository und den geänderten Versionen dieser Dateien in der Staging-Area an (nachdem diese mit `git add` zur Staging-Area hinzugefügt wurden und bevor diese mit `git commit` zum Arbeitsverzeichnis hinzugefügt werden).

- `git diff <Commit-ID1> <Commit-ID2>` zeigt zeilenweise Unterschiede zwischen `Commit-ID1` und `Commit-ID2` an (häufig als *"die diffs"* bzw. *"der diff-Log" zwischen den beiden Commits* bezeichnet). Wie unter `git log --oneline` bereits erwähnt, kann für `Commit-ID1` und `Commit-ID2` die Ausgabe von `git log --oneline` verwendet werden.
  Es hängt von der Reihenfolge der Parameter ab, was in dieser Ausgabe als hinzugefügt (am Beginn einer Zeile vorangestelltes +) und was als entfernt (am Beginn einer Zeile vorangestelltes -) dargestellt wird. Üblicherweise verwendet man `git diff <aeltererCommit> <neuererCommit>`, um in `neuererCommit` *hinzugefügte* Dinge mittels + und in `neuererCommit` *entfernte* Dinge mittels - dargestellt zu bekommen.

- `git mv <AlterDateiname> <NeuerDateiname>` – Versionierte Datei oder versioniertes Verzeichnis verschieben oder umbenennen (anschließend ist die Ausführung von `git commit` notwendig). (Hinweis für Unix-Nutzer: ist dasselbe

wie `mv <AlterDateiname> <NeuerDateiname>`
`&& git add <NeuerDateiname> && git rm <AlterDateiname>`.)

Wiederherstellen von früheren Versionen bzw. Zurücknahme von Änderungen

- `git revert <Commit-ID>` nimmt im aktuellen Branch die Änderungen des durch `Commit-ID` referenzierten Commits zurück (also alles, was durch diesen Commit am Projekt verändert wurde), ohne diesen Commit aus der Historie zu löschen. Die Rücknahme des durch `Commit-ID` referenzierten Commits wird als neuer Commit dargestellt. Sowohl der zurückgenomme Commit als auch der Commit, der die Rücknahme referenziert, sind anschließend mittels `git log` bzw. `git log --oneline` nachvollziehbar.
  `git revert <Commit-ID>` funktioniert in der Regel nur dann ohne Merge-Konflikte, wenn `Commit-ID` den letzten Commit referenziert.
  Die Verwendung von `git revert` ist bei der alltäglichen Arbeit mit Git eher unpraktikabel.

- `git reset <modus> <Commit-ID>` bringt den aktuellen Branch auf den Stand des durch `Commit-ID` referenzierten Commits und löscht alle Commits davor aus der Historie. Kennt die drei Modi *soft* (`--soft`), *mixed* (`--mixed`) und *hard* (`--hard`).
  Beispiele:

  - `git reset --soft <Commit-ID>` nimmt alle Commits vor `Commit-ID` zurück, behält die Änderungen aber im Arbeitsverzeichnis und in der Staging-Area (die zurückgenommenen Änderungen können also mit `git add` erneut zur Staging-Area und mit `git commit` als neuer Commit erneut zum Arbeitsverzeichnis hinzugefügt werden).
  - `git reset --hard <Commit-ID>` nimmt alle Commits vor `Commit-ID` zurück und löscht sie komplett und unwiderruflich.

  `Commit-ID` kann entweder eine absolute Angabe durch ein Commit-ID-Präfix sein, welches man mittels `git log --oneline` (siehe oben) erhält, oder eine relative Angabe wie z.B. `HEAD~2` (siehe oben).
  Merke: Für das Löschen von Dateien und Entfernen dieser Dateien aus der Versionierung verwendet man nicht `git revert` oder `git reset`, sondern `git rm` (siehe weiter oben).

- `git reset HEAD <Datei>` entfernt die zuvor mittels `git add` zur Staging-Area hinzugefügte `Datei` aus der Staging-Area.

- `git checkout -- <Datei>` löscht noch nicht versionierte Änderungen (vor Ausführung von `git add`) an *Datei*. Mittels `git checkout -- *` kann man alle noch nicht versionierten Änderungen auf dem aktuellen Branch des Git-Archives löschen.

- `git clean` entfernt alle nicht versteckten Dateien des Git-Archivs, welche nicht unter Versionsverwaltung stehen. `git clean -x` entfernt alle versteckten und nicht-versteckten Dateien, welche nicht unter Versionsverwaltung stehen. Diese Befehle sind v.A. zur Entfernung von build-Dateien nützlich.

- Man kann wiefolgt die Commit-Message des letzten Commits verändern, falls diese bereits auf einen Remote-Branch gepusht wurde:
  `git reset --soft <praefixDerVorletztenCommitID>`
  `git commit -m ‘‘Neue Commit-Message’’`

`git push -f`
Merke: In GitLab ist es standardmäßig nicht möglich, Commits zu resetten, welche auf den master-Branch gepusht wurde. Daher empfiehlt es sich, beim Arbeiten einen branch *development* anzulegen (und auch ins Remote-Repository zu pushen). Dieser kann dann regelmäßig in den master-Branch gemerged werden.

Weitere Git-Befehle

- `git init` initialisiert ein leeres Git-Repository oder initialisiert ein bestehendes Git-Repository neu. `git init` ist nützlich, wenn man ein bestehendes Verzeichnis als Git-Repository anlegen möchte. Falls GitLab verwendet wird, legt man dazu zunächst über die grafische Nutzeroberfläche von GitLab ein neues Projekt an. Dann geht man mit seiner Shell in das lokale Verzeichnis, welches man als Git-Projekt anlegen möchte (`cd /pfad/zum/Verzeichnis`) und führt folgende Schritte aus:

  ```
  git init
  git remote add origin git@gitlab.example.com:nutzer/NameDesGitRepositories.git
  ```
  (die URL erhält man, wie oben beschrieben, durch Kopieren im GitLab-Webinterface)
  ```
  git add -A
  git commit -m ‘‘initial commit’’
  git push --set-upstream origin master
  ```

  Falls es bereits ein Remote-Repository gibt (welches nicht zum Zweck des Anlegens eines bestehenden Verzeichnisses als Git-Repository neu angelegt wurde), verwendet man nicht `git init`, sondern `git clone`.

- `git stash` bzw. `git stash push` speichert alle seit dem letzten Commit vorgenommenen Änderungen zwischen und nimmt alle Änderungen seit dem letzten Commit zurück (Zustand der Variablen `HEAD`). `git stash` kann beliebig oft aufgerufen werden, auf diese Weise können beliebig viele sogenannte *Stashes* auf dem Stash-Stack gespeichert werden. Verschiedene Stashes können mit `git stash list` aufgelistet und mit `git stash show [<stash>]` begutachtet werden.
  `git stash pop [<stash>]` stellt die Änderungen des angegebenen Stashes wieder her und löscht diesen aus dem Zwischenspeicher (das Gegenteil von `git stash` bzw. `git stash push`. `stash` ist dabei eine natürliche Zahl, welche einen der Stashes auf dem Stash-Stack referenziert (welcher mit `git stash list` ausgegeben werden kann). Falls für `stash` kein Argument übergeben wurde, wird (wie bei einem Stack zu erwarten) der zuletzt gepushte Stash wiederhergestellt.

- Falls auf einem mit einem Remote-Repository verknüpften Branch (von anderen Projektteilnehmern) Änderungen vorgenommen wurden und man auf demselben Branch selbst lokal Änderungen vorgenommen hat, produziert Git automatisch einen Merge-Commit. Dieser lässt sich vermeiden, indem die lokalen Änderungen zunächst zwischengespeichert und erst nach einem `git pull` hinzugefügt werden:

  ```
  git stash
  git pull
  git stash pop
  ```

  Falls die Änderungen auf dem Remote-Repository und dem lokalen Klon sich

widersprechen, kommt es zu Merge-Konflikten, welche dann manuell behoben werden müssen (mehr dazu weiter unten).

Bezogen auf Branches

Ein *Branch* ist quasi eine interne Abspaltung eines Projektes. Branches ermöglichen es, Änderungen am Projekt vornehmen zu können, ohne dass dies unmittelbare Auswirkungen auf andere Branches hat. Branches können jederzeit lokal angelegt, auf Remote-Repositories gepusht und gelöscht werden. Zwischen verschiedenen Branches kann jederzeit hin und her gewechselt werden. Änderungen eines Branches A können auf einen anderen Branch B übernommen werden, in diesem Fall spricht man davon, dass A nach B *gemerged* wird. Beim Anlegen eines neuen Git-Repositories gibt es nur einen einzigen, von Git angelegten Branch, welcher den Namen *master* hat.

Beim Arbeiten an komplexeren Projekten ist es häufig sinnvoll, für Teilaufgaben sogenannte *Feature-Branches* zu verwenden. Dabei handelt es sich um normale Branches, die zur Realisierung eines bestimmten Features eines Projektes (oder z.B. auch zur Behebung eines Bugs) angelegt werden. Dies dient dazu, nicht alle (möglicherweise experimentellen) Änderungen sofort auf den Haupt-Entwicklungs-Branch zu comitten und auf dem Haupt-Entwicklungs-Branch immer eine lauffähige Version des Projektes zu haben. Feature-Branches sollten mit einem vorangestellten "feature/" benannt werden.

- `git branch` listet alle vorhandenen Branches des lokalen Git-Repositories auf, in dessen Verzeichnis man sich mit seiner Shell gerade befindet. Falls nicht explizit Branches angelegt wurden, so erscheint lediglich die Ausgabe `*master`.
  - `git branch -r` listet alle Remote-Branches auf
  - `git branch -a` listet alle lokalen und alle Remote-Branches auf (ist dasselbe wie `git branch && git branch -r`). Remote-Branches werden dabei mittels `origin/branchname` referenziert.
- `git branch bernddasbrot` erzeugt den neuen Branch mit dem Namen "bernddasbrot".
- `git checkout bernddasbrot` wechselt zum Branch bernddasbrot ("checkt zum Branch bernddasbrot aus"). Der Befehl setzt die Variable `HEAD` (falls er vorher auf dem chilidassschaf-Branch war von chilidasschaf) auf diesen neuen Branch.
- Die beiden obigen Befehle lassen sich in einem Befehl ausführen:
  `git checkout -b bernddasbrot` (legt den neuen Branch "bernddasbrot" an und wechselt zu ihm).

Wie bereits erwähnt, können Branches *gemerged* werden, um in einem Branch vorgenommene Änderungen in (mindestens) einen anderen Branch zu übernehmen. Im Folgenden soll der bernddasbrot-Branch in den master-Branch gemerged werden. Dazu wechselt man zunächst mit `git checkout master` in den master-Branch und holt dann mit `git merge bernddasbrot` alle Änderungen aus dem bernddasbrot-Branch in den master-Branch. Der bernddasbrot-Branch bleibt dabei unverändert.

- `git branch -d <branchName>` löscht den gemergten Branch `branchName`.
- `git branch -D <branchName>` löscht den nicht gemergten Branch `branchName`. Dies ist z.B. sinnvoll, falls man auf `branchName` etwas ausprobiert hat, was man wieder rückgängig machen möchte, ohne es auf einen anderen Branch zu übernehmen.

Bei einem Merge können *Merge-Konflikte* auftreten. Ein Merge-Konflikt entsteht, falls ein Branch A in einen Branch B gemerged wird, wobei auf A Änderungen vorgenommen wurden, die mit Änderungen auf B in Konflikt stehen (z.B. verschiedene Edits derselben Zeile in einer Quellcode-Datei). Außerdem können Merge-Konflikte auftreten, wenn mehrere Leute parallel auf demselben Branch Änderungen durchführen, die sich widersprechen (z.B. das Editieren derselben Zeile in einer Quellcode-Datei) und diese anschließend auf den Remote-Branch pushen bzw. vom Remote-Branch pullen (zu Remote-Branches siehe nächster Abschnitt - das Pushen auf einen Remote-Branch ist technisch ein *Merge* des lokalen Branches, auf dem man sich gerade befindet, in den verknüpften Remote-Branch, das Pullen ein Merge des verknüpften Remote-Branches in den lokalen Branch, auf dem man sich gerade befindet).
Wenn ein Merge-Konflikt auftritt, erhält Git beide in Konflikt stehende Änderungen und informiert darüber, dass ein Merge-Konflikt aufgetreten ist. Falls es sich bei den Konflikt-Dateien um textbasierte Dateien handelt, makiert Git die entsprechenden Konflikt-Stellen in der jeweiligen textbasierten Datei inclusive Commit-Referenzierungen. Solche Merge-Konflikte lassen sich dann durch Editieren der entsprechenden Dateien auflösen, indem man die jeweilige Änderung übernimmt, welche man übernehmen möchte und die Konflikt-Markierungen entfernt werden. (Anschließend wie gewohnt `git add`, `git commit` und ggf. `git push` ausführen.)

- Mit `git branch -m oldBranchName newBranchName` benennt man den Branch *oldBranchname* in *newBranchName* um.
  Umbenennen von Remote-Branches: Siehe weiter unten. Umbenennen von versionierten Dateien: siehe `git mv`.
- `git branch --merged [branchName]` gibt alle Branches zurück, die in den Branch `branchName` gemerged wurden. Falls `git branch --merged` ohne Argumente aufgerufen wird, werden alle Branches zurückgegeben, die in den Branch gemerged wurden, in dem man sich gerade befindet. Gibt nur den Branch selbst zurück, falls kein anderer Branch in diesen Branch gemerged wurde.
- `git branch --no-merged` ist das Gegenteil von `git branch --merged`

4.2 Befehle, die Auswirkungen auf ein Remote-Repository haben

Bezogen auf Branches

Die bisher aufgeführten Branch-Befehle haben nur *lokale* Auswirkungen. In den meisten Fällen arbeitet man aber auch mit einem *Remote*-Repository, um eine Sicherungskopie von seiner Arbeit zu haben und/oder um auf mehreren Geräten arbeiten zu können und/oder weil man mit mehreren Leuten gemeinsam an einem Projekt arbeiten möchte.

- Sowohl einen lokalen als auch einen Remote-Branch *briegelderbusch* anlegen und beide Branches miteinander verknüpfen:
  1. `git checkout -b briegelderbusch` legt den lokalen Branch briegelderbusch an und wechselt zu ihm (wie oben bereits erwähnt Kurzform von
     `git branch briegelderbusch && git checkout briegelderbusch`)
  2. Veränderungen vornehmen (z.B. Dateien editieren und die Änderungen mittles `git add` und `git commit` zum lokalen Repository hinzufügen)

3. `git push --set-upstream origin briegelderbusch` erstellt den Branch briegelderbusch im Remote-Repository und verknüpft diesen mit dem lokalen Branch briegelderbusch (setzt den Remote-Branch briegelderbusch als *Upstream-Branch* für den lokalen Branch briegelderbusch).

Nach Ausführung dieser drei Befehle kann auf dem Branch briegelderbusch einfach `git pull` und `git push` ausgeführt werden.

- `git checkout --track origin/fritzhoppenstedt` legt den lokalen Branch fritzhoppenstedt an, welchselt zu ihm und verknüpft ihn mit dem bereits existierenden Remote-Branch fritzhoppenstedt. Der neu erstellte lokale Branch fritzhoppenstedt ist auf diese Weise sofort mit dem Remote-Branch fritzhoppenstedt verknüpft und auf dem aktuellen Stand dieses Branches. Im Folgenden können auf dem lokalen Branch fritzhoppenstedt also einfach `git pull` und `git push` ausgeführt werden.
  Falls der Branch fritzhoppensted bereits im Remote-Repository existiert, reicht es, `git checkout fritzhoppensted` (ohne `-b`) einzugeben, um den Branch fritzhoppenstedt lokal anzulegen und ihn mit dem Remote-Branch fritzhoppenstedt zu verknüpfen.

- `git branch --unset-upstream` hebt die Verknüpfung des lokalen Branches, auf dem man sich gerade befindet, mit dem Remote-Branch auf, mit dem der lokale Branch verknüpft ist. Dies funktioniert auch dann, wenn der Remote-Branch gelöscht wurde.

- `git push --set-upstream origin fritzhoppenstedt` verknüpft den lokalen Branch fritzhoppenstedt mit dem Remote-Branch fritzhoppenstedt.

- `git push origin --delete fritzhoppenstedt` löscht den Remote-Branch fritzhoppenstedt. Löschen von lokalen Branches: Siehe `git branch -d` bzw. `git branch -D`.

- Einen Remote-Branch kann man folgendermaßen umbenennen:

  1. Mittels `git checkout` zum lokalen Branch wechseln, welcher mit dem unzubenennenden Remote-Branch verknüpft ist. Falls noch kein solcher Branch existiert, legt `git checkout` diesen (wie bereis erwähnt) an (falls der entsprechende Remote-Branch existiert).
  2. Remote-Branch löschen
     (`git push origin --delete <branchName>`).
  3. `git branch --unset-upstream` ausführen.
  4. Lokalen Branch wie gewünscht umbenennen
     (`git branch -m oldBranchName newBranchName`).
  5. `git push --set-upstream origin newBranchName` ausführen (erstellt, wie oben bereits beschrieben, den Remote-Branch newBranchName und verknüpft ihn mit dem lokalen Branch newBranchName).

  Umbenennen von lokalen Branches: Siehe `git branch -m`. Umbenennen von versionierten Dateien: Siehe `git mv`.

Weitere Befehle auf Remote-Repositories

- `git clone <url>` klont ein komplettes Git-Remote-Repository in ein dabei neu erstelltes Verzeichnis mit demselben Verzeichnis-Namen, den auch das

Remote-Repository hat (ohne .git) und erstellt auf diese Weise ein lokales Repository (zum Zeitpunkt der Ausführung von `git clone` ein Klon des Remote-Repositories). Es wird automatisch jeweils einen Remote-Tracking-Branch für jeden auf dem Remote-Repository vorhandenen Branch erzeugt. Zusätzlich wird ein lokaler Branch erstellt, welcher denselben Namen erhält wie der Haupt-Branch auf dem Remote-Projekt (dies ist häufig der master-Branch). Nach Aufruf von `git clone` sind `git pull` und `git push` so konfiguriert, dass sie ohne weitere Optionen direkt funktionieren.

- `git pull` holt alle Änderungen auf allen Branches aus dem Remote-Repository in das lokale Repository. Ist in seiner Standardkonfiguration dasselbe wie `git fetch` mit anschließendem `git merge FETCH_HEAD`.
- `git push` schreibt lokale Änderungen (Commits) auf dem Branch, auf dem man sich gerade befindet, auf das Remote-Repository.
- `git remote` handhabt Verknüpfungen zwischen lokalem Repository und Remote-Repository (dem entfernten Repository). Verwendungs-Beispiel: siehe `git init`.

5 Die Datei *.gitignore*

Fügt man einem Verzeichnis im Git-Repository eine Datei mit dem Namen *.gitignore* hinzu, so werden alle darin aufgeführten Dateien bzw. Verzeichnisse von der Versionsverwaltung ausgenommen. Aus Gründen der Übersichtlichkeit und Wartbarkeit empfiehlt es sich, *eine* zentrale .gitignore-Datei für das gesamte Git-Repository zu pflegen, welche im Basis-Verzeichnis des Git-Repositories abgelegt wird (indem auch das .git-Verzeichnis liegt)[11].

[11] (Theoretisch lässt sich in jedem Verzeichnis und Unterverzeichnis des Git-Repositories jeweils eine .gitignore-Datei erstellen, was allerdings extrem unübersichtlich wäre.)

Beispiele:

- Möchte man von einem LaTeX-Compiler erzeugte LaTeX-Systemdateien nicht versionieren (insbesondere also nicht zum Remote-Repository hinzufügen), so befüllt man die Datei .gitignore mit folgendem Inhalt:
```
*.aux
*.lof
*.log
*.lot
*.fls
*.out
*.toc
*.fmt
*.fot
*.cb
*.cb2
.*.lb
*.dvi
*.xdv
*.bbl
*.bcf
*.blg
*-blx.aux
*-blx.bib
*.run.xml
*.nav
*.pre
*.snm
*.vrb
*.synctex.gz
*.tex.backup
```

- Der folgende Eintrag ignoriert das Verzeichnis *build*:
```
build/
```

- Der folgende Eintrag ignoriert das Verzeichnis *build* mit Ausnahme der Datei *version.h* innerhalb dieses Verzeichnisses:
```
build/*
!build/version.h
```

- Der folgende Eintrag ignoriert PDF-Dateien im Verzeichnis *blub* und rekursiv in allen Unterverzeichnissen des Verzeichnisses blub:
```
blub/**/*.pdf
```

- Kommentare werden mit `#` eingeleitet.

Git schließt in der .gitignore-Datei aufgeführte Dateien und Verzeichnisse nur dann von der Versionierung aus, wenn diese nicht vorher (mittels `git add`) bereits unter Versionsverwaltung gestellt wurden. Hat man versehentlich Dateien bereits unter Versionsverwaltung gestellt bzw. möchte man nachträglich Dateien von der Versionsverwaltung ausnehmen, geht man folgendermaßen vor:

1. Von der Versionsverwaltung auszunehmende Dateien temporär außerhalb des Git-Archives kopieren (bzw. falls sie z.B. beim nächsten Kompilieren z.B. von einem LaTeX-Compiler eh wieder erzeugt werden, einfach löschen).

2. Diese Dateien zur .gitignore-Datei hinzufügen.
3. Änderungen (wie oben beschrieben) adden, committen und pushen.
4. Von der Versionsverwaltung auszunehmende Dateien wieder ins Git-Archiv zurückkopieren. Diese Dateien stehen nun nicht mehr unter Versionsverwaltung und sind im aktuellen Commit nicht mehr im Remote-Repository, aber auf dem lokalen Dateisystem weiterhin vorhanden.

Weitere Informationen: `https://git-scm.com/docs/gitignore`

6 Schreiben guter Commit-Messages

Regeln für das Schreiben guter Commit-Messages[12]:

- Commit-Messages sollten grundsätzlich in englischer Sprache verfasst werden, um sie für möglichst viele verständlich zu machen.
- Die Commit-Message sollte in Betreff und Body aufgeteilt werden.
- Betreff und Body sollten durch eine Leerzeile getrennt werden.
- Die Betreff-Zeile sollte:
  - maximal 50 Zeichen lang sein.
  - nicht mit einem Punkt enden.
  - im Imperativ geschrieben sein. (Hintergrund: Die Betreff-Zeile sollte in den Satz *If applied, this commit will <Commit-Message>* eingesetzt werden können.)
- Jede andere Zeile der Commit-Message sollte maximal 72 Zeichen lang sein.
- Der Body sollte zur Erklärung des *Was* und *Warum* verwendet werden, nicht des *Wie* (dazu gibt es z.B. `git diff`).
- Falls Issues verwendet werden, sollten diese am Ende der Commit-Message erwähnt werden (z.B. *Resolves #42* oder *See also: #42*).

Ein Beispiel dazu findet sich z.B. unter `https://chris.beams.io/posts/git-commit/#seven-rules`.

Automatischer Zeilenumbruch

Falls man Git in der Bash unter Linux verwendet und den Texteditor *Nano* als Standard-Texteditor für Git nutzt (automatischer Aufruf von Nano nach Eingabe von `git commit`), kann man Nano so konfigurieren, dass Zeilen ausschließlich beim Aufruf des Editors durch Git nach 72 Zeichen umgebrochen werden[13]:

- Entweder in der Datei `~/.bashrc` die Umgebungsvariable *GIT_EDITOR* setzen: `export GIT_EDITOR='nano -r 72'`
- oder durch Eingabe des folgenden Konsolenbefehls die Konfigurationsdatei `~/.gitconfig` editieren:
  `git config --global core.editor ''nano -r 72''`

Weitere Informationen zur Konfiguration des Standard-Texteditors finden sich unter `https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration`.

[12] Quelle: `https://chris.beams.io/posts/git-commit/`
[13] Quelle: `https://stackoverflow.com/questions/42201438/how-to-set-nano-up-for-git-commit-messages-with-line-length-limits`

7 Nützliche Hinweise

- `git --help` gibt die wichtigsten Git-Befehle mit kurzen Erklärungen aus.
- Unter Linux gibt es für jeden Git-Befehl eine eigene *Manpage*[14]. Diese wird mit `man git-<Unterbefehl>` oder `man git <Unterbefehl>` aufgerufen.
- Es ist mittels Git nicht (bzw. nicht ohne weiteres) möglich, leere Verzeichnisse zu versionieren[15] (`mkdir <LeeresVerzeichnis> && git add <LeeresVerzeichnis>`). Das Hinzufügen von neuen Verzeichnissen ist standardmäßig nur implizit durch das Hinzufügen von Dateien in diese neuen Verzeichnisse möglich. Initial eignet sich hierzu z.B. eine Datei Readme.txt oder Readme.md (letzteres ist eine Markdown-Datei[16] – Markdown wird im Zusammenhang mit Git-Projekten häufig zur Dokumentation des Systems verwendet). Ist dies nicht sinnvoll und möchte man trotzdem unbedingt ein leeres Verzeichnis hinzufügen, kann man dazu z.B. eine leere Datei mit dem Namen *.gitkeep* verwenden (dieser Name ist reine Konvention).
- Die Namen von Git-Repositories enden häufig mit *.git*, um sie als Git-Repository zu kennzeichnen. (Bei *meinTollesGitRepository.git* handelt es sich also (in der Regel) nicht um eine Datei mit der Dateinamenserweiterung .git, sondern um ein Git-Verzeichnis.)
- Beim Arbeiten mit mehreren Branches ist es möglich, in ein und demselben lokalen Klon eines Projektes für jeden Remote-Branch einen lokalen Branch anzulegen und diesen mit dem jeweiligen Remote-Branch zu verknüpfen (was man nach dem Klonen mit `git clone` einfach mit `git checkout` tut - siehe oben). Falls man parallel an mehreren Feature-Branches arbeitet, ist es jedoch sinnvoller, dasselbe Projekt lokal mehrfach zu klonen und auf jedem dieser Klone einen der Feature-Branches auszuchecken, auf dem man arbeitet. Auf diese Weise kann man beispielsweise in Eclipse mithilfe von mehreren Java-Projekten, in welche jeweils ein anderer dieser Projekt-Klone importiert wurde, parallel an verschiedenen Feature-Branches arbeiten.
- Git unterstützt kryptografische Signaturen beispielsweise von Commits mithilfe von GPG[17]. Näheres dazu erfährt man unter `https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work`.

8 Weiterführende Links

- `https://git-scm.com` – offizielle Website des Git-Projektes
- `https://git-scm.com/doc` – offizielle Git-Dokumentation mit Benutzerhandbuch, ausführlichem Handbuch, Videos, Tutorials und mehr
- `https://git.wiki.kernel.org` – das Git-Wiki der Linux-Kernel-Entwickler

9 Fehlerbehebung

Falls unter Linux Fehler bei der SSH-Verbindung auftreten: Debugging mittels `ssh -Tv git@gitlab.example.com` durchführen. Dieser Befehl sollte, wenn alles funktioniert, nach Eingabe der Passphrase (nach einer Menge anderer Ausgaben) mit *Exit*

[14] `https://de.wikipedia.org/wiki/Manpage`
[15] `https://git.wiki.kernel.org/index.php/GitFaq#Can_I_add_empty_directories.3F`
[16] `https://de.wikipedia.org/wiki/Markdown`
[17] `https://gnupg.org/`, `https://de.wikipedia.org/wiki/GNU_Privacy_Guard`

status 0 enden. Ansonsten enthalten die Fehlermeldungen in der Ausgabe eventuell nützliche Hinweise darauf, was schief läuft.

10 Kurze Definition von verwendeten Unix-Befehlen

(Haben nichts unmittelbar mit Git zu tun, sind aber (meines Wissens) auch in der Git-Shell unter Windows nutzbar.)

- `cat <Datei>` schreibt den Inhalt von `Datei` in die Standardausgabe (gibt den Inhalt von `Datei` in der aktuellen Shell aus).
- `cd <Dateipfad>` wechselt in den (absoluten oder relativen) `Dateipfad` (*cd = change directory*).
- `mkdir <Verzeichnisname>` erstellt ein leeres Verzeichnis (*mkdir = make directory*).
- `mv <Quelle> <Ziel>` benennt `Quelle` in `Ziel` um oder verschiebt `Quelle` nach `Ziel` (*mv = move*).
- `touch <Dateiname>` ändert unter Unix-Systemen den Zeitstempel der Datei `Dateiname` auf die aktuelle Systemzeit. Falls die Datei `Dateiname` nicht existiert, wird sie neu (leer) angelegt.
- Mittels `<Befehl1> && <Befehl2>` werden zwei Befehle verkettet. `Befehl2` wird dabei nur dann ausgeführt, wenn `Befehl1` keinen Fehler geworfen hat.