

## Git Aufgaben

Hierbei geht es um das Beantworten 41 Fragen zum Versionsverwaltungssystem Git.

- 1.) → Bei einem zentralen Versionsverwaltungssystem arbeiten alle Entwickler an einem Projekt, welches auf einem Server liegt, damit alle gleichzeitig Zugriff darauf haben. Ein Beispiel hierfür wäre „Apache Subversion“, kurz auch SVN genannt. Hierbei kann es vermehrt zu Problemen kommen, wenn man sich nicht im Team abspricht bevor man seine Änderungen für alle übernimmt. Verbindung zum Server erforderlich.
- Bei einem dezentralen System ist das nicht der Fall. Jeder Entwickler hat seine eigene Kopie des Projekts und überträgt nach dem Abschließen eines Auftrags seine Änderungen auf das Hauptprojekt. Dieses liegt auf einem Server. „Gitorious“, bzw. GIT, wäre ein Beispiel für ein dezentrales System. Es wird mehr Sicherheit für das Projekt geboten, da jeder Entwickler eine eigene Kopie hat, die er vor dem Übernehmen ausgiebig testen kann. Es sind auch offline Arbeiten daran möglich.

- 2.) Für ein einzelnes Repository:

```
$ git config user.name "M. Lengl"
$ git config user.email "mle@...de "

$ git config user.name
Marcel Lengl
$ git config user.email
mle@...de
```

Für alle Repositories:

```
$ git config --global user.name "M. Lengl"
$ git config --global user.email "mle@...de"

$ git config --global user.name
M. Lengl
$ git config --global user.email
mle@...de
```

- 3.) Erstellen eines neuen und leeren Repositories:

```
$ git init --bare <directory>
```

- Das Angeben eines Verzeichnisses ist optional. Macht man an dieser Stelle keine Angabe, so wird das aktuelle Verzeichnis dafür verwendet.
- --bare wird verwendet, wenn man ein Repository anlegen möchte, welches später mit anderen Entwicklern, in Form von Ablegern (Kopien), geteilt werden soll.

## 4.) Eine Arbeitskopie erstellen:

```
$ git clone <Pfad zum Ziel>
$ git remote add origin <server-URL> // Wenn repo kein Klon ist aber zusammengeführt werden soll.
```

## 5.) GIT definiert 3 Hauptzustände um Fehler zu vermeiden.

- ➔ modified: Eine Datei wurde verändert/angelegt/gelöscht.
- ➔ staged: Eine Datei wurde geändert, aber noch nicht übernommen. Sie ist aber für den nächsten commit vorgemerkt.
- ➔ committed: Datei wurde lokal gespeichert.

## 6.) Die Staging-Area ist eine Datei im GIT-Verzeichnis, in der vorgemerkt wird welche Änderungen der nächste commit umfasst. Zum Staging Area wird oft auch Index gesagt. → Versioniert wird erst beim commit

## 7.) Den Status von Dateien im aktuellen Branch kann ich mit dem folgenden Befehl anzeigen lassen. Die Art der Änderung wird durch eine farbige Markierung hervorgehoben und durch ein voran gestelltes Wort gekennzeichnet.

Diese sind: deleted, added, modified, new File, ...

```
$ git status
```

## 8.) Dateien oder Verzeichnisse können hinzugefügt werden, wenn man zB in Eclipse eine neue Datei/Klasse anlegt oder man direct einen add-Befehl für GIT anwendet. Die Datei/das Verzeichnis ist nach dem add-Befehl nur auf dem momentanen Branch und muss erst noch commitet werden.

```
cd <directory>
$ git add <filename>

$git commit -m "Neue Datei/Verzeichnis hinzugefügt."
```

## 9.) Mehrere Dateien können dem Index mit folgendem Befehl hinzugefügt werden:

```
$ git add -I // Adden mit Hilfe eines Menüs.
$ git add .
$ git add --all

#Nur bestimmte Dateien (Filter: nur Texte aus einem Ordner)
$ git add <directory>/*.txt #Mit WildCard-Nutzung
```

Alle Dateien werden dem Index mit der aktuellen Änderung hinzugefügt.

10.) Datei adden und dann einen Commit absetzen.

```
$ git add <file>
$ git commit -m „TEXT“
$ git push
```

11.) → Dieser Befehl übernimmt alle Dateien, die schon im getrackt werden.

```
$ git add -am "Meine commit-Nachricht"
```

→ Dieser Befehl trägt alle Dateien in den Index ein, die noch nicht vorhanden sind und commitet dann alle. Bereits getrackte Dateien werden ohne in die Staging-Are zu kommen commitet und somit direkt übernommen.

```
$ git add -u .
$ git commit -a
#geänderte Dateien automatisch aufnehmen
$ git add . -A
```

12.) → commit: Ein commit aktualisiert das lokale Repository. Die Änderungen werden aus dem Index heraus übernommen. (Nur dort eingetragene Änderungen.)

→ push: Mit dem push-Befehl werden alle commits auf das Server-Repository übertragen. Somit ist auch dieser wieder aktuell und andere können sich dann auch die Änderungen auf ihr lokales Repository herunterladen.

## 13.) Durch den Befehl

```
$ git log
```

Option	Beschreibung
<code>-p</code>	Zeigt die Veränderungen durch jeden commit.
<code>--stat</code>	Zeigt die Statistik von jedem commit.
<code>--shortstat</code>	Zeigt in kurzer form die Anzahl der Neuerungen auf. +,-,changed.
<code>--name-only</code>	Zeigt eine Liste der Namen von Dateien die Verändert wurden.
<code>--name-status</code>	Zeigt eine Liste der Veränderten Dateien an mit vorangestellten Statusbuchstaben.
<code>--abbrev-commit</code>	Zeigt nur die ersten paar Stellen der SHA-1 Checksumme statt alle 40.
<code>--relative-date</code>	Zeigt ein relatives Änderungsdatum an (zum Beispiel, "2 weeks ago"), anstatt des genauen zeitpunkts.
<code>--graph</code>	Zeigt durch ASCII-Zeichen einen Graphen zum Commit- und Mergeverlauf an.
<code>--pretty</code>	Zeigt die commits in einer anderen Form. Beinhaltet Optionen: oneline, short, full, fuller, and format (eigenes Format festlegen).

## 14.) Änderungen oder Unterschiede von local und remote lassen sich wie folgt anzeigen:

```
$ git diff <object 1> <object 2>
```

```
#Das Repository in dem man sich befindet muss nicht angegeben werden.
```

```
$ git diff master blessed/master
```

## 15.) 2 Möglichkeiten Änderungen zu holen:

```
#Änderungen von Dateien in den Index holen
```

```
$ git fetch <repo_name>
```

```
#Änderungen holen und mit branch zusammenführen (Neue Dateien herunterladen und auto. adden)
```

```
cd <branchname>
```

```
$ git pull <branchname>
```

## 16.) Änderungen werden durch einen Commit auf dem lokalen Repository gespeichert.

Man kann diese Daten nun per `push` oder `merge` auf das remote Repository übertragen.

17.) Eine commit-Message kann man ändern, in dem man auf den Branch wechselt wo dieser commit gemacht wurde und diesen Befehl abgibt:

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

18.) Es gibt 2 Möglichkeiten.

→Setzt einen neuen Commit mit dem Stand des gewünschten Commits ganz oben in die Reihe.

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

→Dateien/Verzeichnisse aus der Staging-Area nehmen und vor neuem hinzufügen editierbar. 2 Möglichkeiten:

```
$ git reset HEAD <file>
```

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

19.) Auch hier wieder 2 Möglichkeiten:

→Einfach nur die Änderung aus der Staging-Area holen und bearbeitete Datei dann noch einmal mit dem gleichen Befehl ganz verwerfen.

→Datei im Repo lassen aber komplett aus dem Index entfernen.

```
#Aus dem Index holen
```

```
$ git checkout <filename>
```

```
#Aus dem Index herausnehmen aber im Repo lassen
```

```
$ git rm --cached <filename>
```

20.) Beide Befehle zeigen alle commits dazu an. Autor steht jeweils dabei.

```
#Commit-Message durchsuchen
```

```
$ git log -S <String in "">
```

```
#Commit-Message und Dateinamen suchen
```

```
$ git log -G <String in "">
```

```
$git blame <filename>
```

21.) Man benötigt die commit-ID und dann kann man folgendes eingeben, um nur den Namen der Dateien oder auch die kompletten Änderungen anzeigen zu lassen.

```
#Zeigt alle Änderungen in dem Commit mit Status.
```

```
$ git show --pretty="format:" <commit-ID>
```

```
#Zeigt nur den Namen der geänderten Dateien.
```

```
$ git show --pretty="format:" --name-only <commit-ID>
```

## 22.) Erstellen eines neuen Branches

```
$ git checkout -b <branchname> #Kurzform - Wechselt direkt auf den neuen Branch
$ git branch <branchname> #Erstellt einen neuen Branch
$ git checkout <branchname> #Wechselt auf den neuen Branch
```

23.) Ein Branch der noch nicht gelöscht ist kann mit der Option -D gelöscht werden. Er ist dann endgültig weg. Bei -d muss man noch einmal bestätigen, dass man den Branch wirklich löschen will.

```
$ git <branchname> -d #Mit Nachfrage und unmöglich wenn Änderungen nicht zurückgeführt sind.
$ git <branchname> --D #Endgültiges löschen, auch wenn Fehler vorhanden - forced
```

24.) Um jeden Branch lokal oder remote anzeigen zu lassen muss nur 1 Parameter geändert werden.

```
$ git branch -a #Alle
$ git branch -r #Alle Remote
$ git branch #Nur lokal
```

25.) Beim Löschen eines Branches auf dem Remote Repo, wird der lokale Branch auch gelöscht.

```
#Gleiche Funktion
$ git push origin --delete <branchName>
ODER
$ git push origin :<branchName>
```

26.) Es gibt 3 Möglichkeiten einen commit zu übernehmen/sichern.

```
#Commits zurückgehen und in neuen Branch packen.
$ git branch <Neuer Branchname>
$ git reset --hard HEAD~3 #3 commits zurück gehen. Änderungen gehen verloren!
$ git checkout <Neuer Branchname>

#Zu einem ausgewählten Branch springen und in neuen Branch sichern.
$ git branch <Neuer Branchname>
$ git reset --hard <BranchID> #Zu Branch mit bestimmter ID gehen
$ git checkout <Neuer Branchname>

#In einen existierenden Branch sichern.
$ git checkout <existierender Branch>
$ git merge master
$ git checkout master
$ git reset --hard HEAD~3 #SIEHE OBEN
$ git checkout <existierender Branch>
```

- 27.) Ein Branch ist immer die Kopie des Hauptzweiges in der man dann etwas verändert. Will man diese Änderungen übernehmen, so muss man folgenden Befehl anwenden:

```
$ git push origin <lokaler branch> : <in den remote Branch> #Quelle : Ziel  
$ git push <remote> <locale Quelle>:<remote Ziel> #Generelle Form
```

- 28.) Der merge-Befehl führt einfach Branchs zusammen und wird dabei wie ein commit gehandhabt. Rebase hingegen übernimmt eine Änderung an einer beliebigen Stelle und kann an jedem Stand des Projektes angesetzt werden. Es wird zB vor den aktuellen Stand gesetzt um die Grundlage des aktuellen Branches zu verändern.
- 29.) Merge-Konflikte entstehen, wenn sich nicht seit dem ziehen eines Branche die Basisdateien geändert haben. So ist nicht mehr bekannt ob die gemachten Änderungen wirklich überall wirklich übernommen werden sollen, da der Entwickler eventuell auch nicht weiß, dass jemand etwas geändert hat.  
→ Man behebt die Fehler manuell im SourceCode wo eine Änderung gekennzeichnet wurde und löscht alles außer dem gewünschten Code, der bleiben soll.
- 30.) Es wird einfach entlang der commit-Historie gemerged. Haben sich aber im aktuellen Branch Daten geändert, so muss man in einzelnen Schritten durch den merge gehen und die Fehler beheben. Man muss zusätzlich darauf achten, dass man einen solchen Merge in der Historie nicht als merge sieht.
- 31.) Wenn man die Änderungen in einem Branch auf einen anwendet, so kann es passieren, dass dieser von einer anderen Person bearbeitet wurde und nicht mit den Änderungen von Person A kompatibel ist. Dies kann dann trotz richtigem Löschen doppelt geänderter Codeteile die Funktion des Codes stören.
- 32.) Ein Hashwert in GIT ist 40 stellen lang. Meist wird er als Kurzform mit 7 Stellen genutzt, jedoch kann dies manchmal dazu führen, dass die ersten 7 Stellen nicht eindeutig sind, wie sie sein sollten. Der Hashwert ist eine ID/Versionsnummer die nicht manipuliert werden kann, da diese Zahl sich aus vielen Faktoren berechnet. Kleinste Änderungen haben schon einen Einfluss auf diesen Wert.
- 33.) Die Hashes bei den einzelnen Commits nach dem Rebasing ändern sich, da nun Änderungen mit dabei sind und die Zeitstempel/Reihenfolge anders. Dies sind Faktoren die bei der Berechnung des Wertes eine Rolle spielen.

- 34.) Dies ist durch die Nutzung eines interaktiven Rebasings möglich. Man kann die Reihenfolge der Commits in einem Script ändern oder die komplett löschen.

```
#Zuvor
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

#Dannach
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

- 35.) Durch Tags lassen sich besondere Versionen/Stände in der Historie markieren. Dies wird in der Praxis meist dazu benutzt Releases zu bestimmten. Ein Tag ist eine eigene Anordnung von Zeichen/Zahlen. Es gibt 2 Arten von Tags. Einfache tags und kommentierte.

Man kann nun viel einfacher mit Daten umgehen, da man immer nur den Tag nennen muss und man darüber immer den richtigen Commit/Branch erwischt.

```
#Mit -a einen kommentierten Tag anlegen
$ git tag -a v1.0 -m "Mein Text"

#Einfacher Tag
$ git tag v1.0

#Tag verifizieren
$ git tag -v v1.0

#Tag löschen
$ git tag -d v1.0
$ git push origin :refs/tags/v1.0
```

- 36.) Beim Stashing schreibt man seine aktuellen Änderungen (im Index oder auch nicht), die noch nicht commitet sind, in den Stack. Von dort kann man sie jeder Zeit wieder holen und anwenden.

Man tut dies vor dem wechseln eines Branches, wenn man die Änderungen so noch nicht commiten will oder noch einmal testen möchte, wie es ohne diese Änderungen wäre.

```
$ git stash #ähnlich wie $git add .

#Ansehen der gestashten Dateien
$ git shash list

#Stash rückgängig machen
$ git stash show -p stash@{0} | git apply -R #Mit Stashnamen

#Wenn man es öfter braucht kann man sich einen Shortcut machen
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash apply
...Arbeiten und so
$ git stash-unapply
```

37.) Zu Beginn der Ausbildung waren mir beide Systeme unbekannt. Nach den Tests und dem Arbeiten mit beiden Systemen habe ich meine Entscheidung auf GIT, da dieses Tool innerhalb unseres Projekts und der NT weitestgehend verwendet wird. Nachteile konnte ich durch mangelnde Erfahrung nicht ausmachen und entschied somit nur, was sich *für mich* in Zukunft mehr rentieren wird.

→git-Atom evtl. interessant.

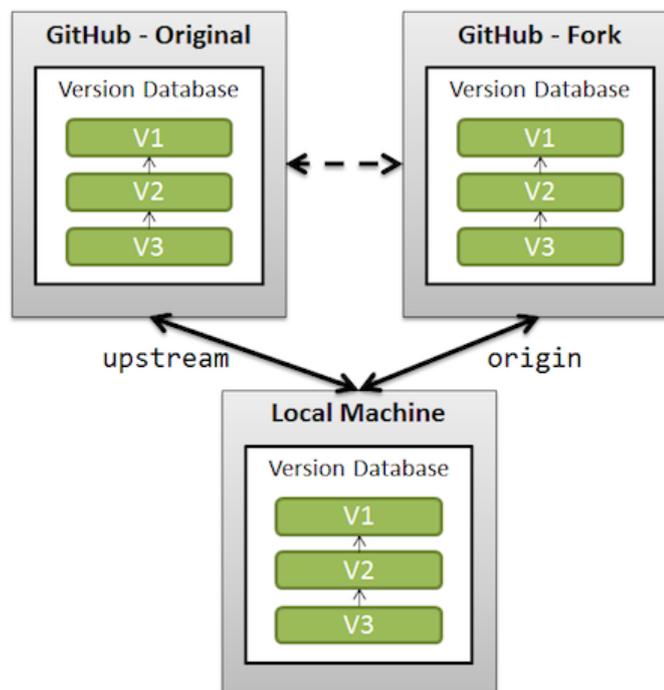
38.) Ein bare-Repo enthält keinen extra Ordner mit allen git Historiendaten. Weiterhin enthält dieser auch keine funktionsfähigen Kopien oder andere Daten zu einem Projekt.

Ein bare-Repo wird nicht zum Arbeiten, sondern zum Teilen verwendet und so kann jeder Nutzer seine Änderungen dort zentral speichern und allen zugänglich machen.

```
$ git init --bare // Erstellt ein bare-Repo
```

39.) Ein Fork-Repo ist eine besondere Kopie eines Repos. Diese Kopie hat einen Bezug zu seinem Ursprung und kann also als Ableger eines Repos betrachtet werden. Man kann von dort aus Änderungen auf das ursprüngliche Original übertragen.

BSP: Eine Software ist bereits im Betrieb, soll aber weiter verbessert werden. Man erstellt ein Fork-Repo und entwickelt nun an diesem Ableger weiter, bis alles für Release 2.0 fertig ist und funktioniert. Nun kann man aus dem sicheren Bereich der Entwicklungskopie alles direkt in das Original-Repo pushen.



40.) Ein Merge-Request ist eine Anfrage auf einen Merge. Diese stellt man einem anderen Entwickler, nach dem man ein Ticket erfolgreich bearbeitet hat und dieses selbst schon getestet hat.

Ziel einer solchen Anfrage ist das Wiederholen der Tests um wirklich sicher zu gehen, dass alles funktionsfähig ist. Ist dies der Fall, so werden die Änderungen dann in das Projekt übernommen.

41.) Ablauf in Textform:

- aktuellen blessed-master auf lokalen master holen.

- Branch erstellen um etwas zu ändern.

- Änderungen adden und commiten.

- mergerequest stellen.

- Entwickler des Teams holt sich den Branch.

- Testet alles.

- Push auf blessed-master.

- Alle aktualisieren ihren lokalen master

