Technisches Gymnasium Ravensburg

Verteilte Versionskontrolle mit Git

Gleichwertige Feststellung von Schülerleistungen

Johannes Sonn
28.11.2017
TG 12/4
Computertechnik bei Frau Fichtner

Inhaltsverzeichnis

Was ist ein Versionskontrollsystem?

Problemstellung

Bei der Entwicklung eines Computerprogramms, ist es wichtig den Überblick über den geschriebenen Quellcode zu behalten. Treten beispielsweise Fehler auf, ist es wichtig nachvollziehen zu können wer oder was diesen Fehler verursacht hat. Hierbei sollte man als Entwickler in der Lage sein auf ältere funktionierende Versionen des Quellcodes zurückgreifen zu können.

Durch das Anlegen von Kopien mit unterschiedlichen Namen (z.B. mit fortlaufender Versionsnummer oder mit Zeitstempel), kann auf älteren Quellcode wieder zugegriffen werden. Durch diese Methode ist allerdings die Übersichtlichkeit nicht gegeben und es kann leicht passieren, dass man die falsche Version einer Datei verwendet, kopiert, überschreibt oder womöglich löscht. Außerdem wird, wenn man die gesamte Datei kopiert, um eine neue Version anzulegen, unnötig viel Speicher belegt. Dies liegt daran, dass der unveränderte Inhalt einer Datei ebenso, wie die Änderungen kopiert wird und man so dieselben Daten öfter, als benötigt vorliegen hat. Ein weiteres Problem kann beim Kollaborieren entstehen, da möglicherweise nicht jeder mit der letzten Version, und somit auf einer anderen Grundlage, arbeitet.

Einleitung

Um den oben genannten Problemen aus dem Weg zu gehen, wurden Versionskontrollsysteme, kurz VCS (Version Control Systems) erfunden. Versionskontrollsysteme speichern Änderungen einer oder mehrerer Dateien (meistens mit einem Zeitstempel und einer Checksumme) über die Zeit hinweg in ein sogenanntes Repository (englisch für Lager, Depot oder auch Quelle). Dies erlaubt es zu jeder Zeit auf eine beliebige Version oder Änderung zugreifen zu können und auf diese zurückzusetzen. Außerdem kann man mithilfe eines VCS erfahren wer wann einen Teil oder eine ganze Datei geändert hat, indem man die Logdatei des VCS einsieht, welche neben dem Zeitstempel auch den Autor und weitere Informationen zu einer Änderung beinhalten kann. Dadurch entsteht die Möglichkeit beispielsweise den Autor einer Änderung, die ein Problem verursacht oder fehlerhaft ist, auszumachen, zu kontaktieren und um Verbesserung oder Erläuterung des Quellcodes zu bitten.

Um eine Version einer Datei aus einem Repository zu bearbeiten, wird zuerst in einem Arbeitsverzeichnis eine *Arbeitskopie* der Datei erstellt. Dies wird als *Check-out* bezeichnet. Nachdem Abschluss der Bearbeitung wird diese Prozedur umgekehrt ausgeführt und die Änderungen in einer neuen Version gespeichert. Dies wird wiederum als *Check-in* (bei Git *Commit*) bezeichnet. Im Fall von Git wird beim Check-in die geänderte Datei zunächst „staged“ (d.h. für den nächsten Commit vorgemerkt) und anschließend „commited“ und damit dauerhaft in das Repository gespeichert. (siehe Abbildung 1)

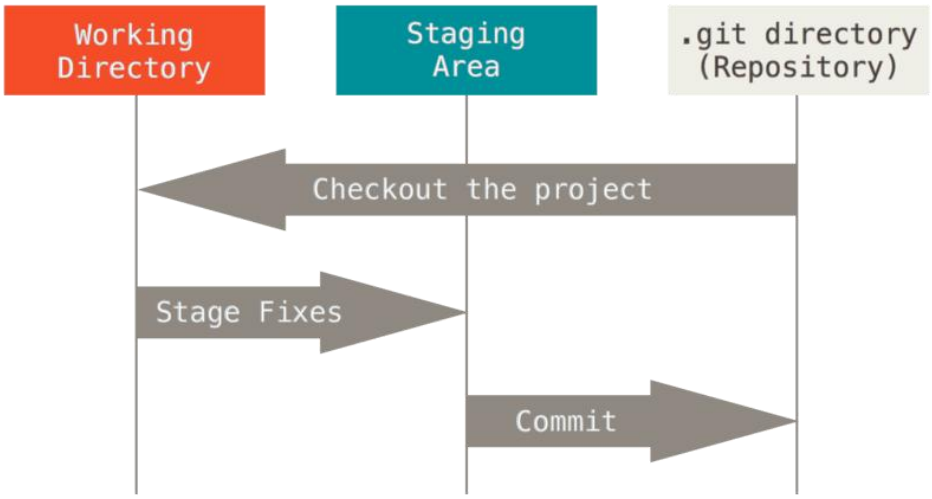

Abbildung 1 Arbeitsverzeichnis, Staging Area und Git Verzeichnis

Lokale Versionskontrollsysteme

Lokale Versionskontrollsysteme sind Versionskontrollsysteme, die Dateien nur lokal verwalten. Allerdings wird bei lokalen Versionskontrollsystemen oft für jede versionierte Datei eine separate Datei, die den Änderungsverlauf beinhaltet, angelegt. Deshalb kann jede Datei nur einzeln und nicht ein komplettes Projekt versioniert werden.
Beispiele für lokale VCS sind SCCS (Source Code Control System) von 1972 von Marc J. Rochkind und RCS (Revision Control System) von Anfang der 1980er Jahre von Walter F. Tichy. RCS wird auch heute noch mit vielen Computern verkauft. Es funktioniert, indem es für jede versionierte Datei eine Änderungsdatei (patch set) in einem speziellen eigenen Format anlegt (siehe Abbildung 2). Durch Aufsummieren von Änderungen (patches) kann eine Datei so wiederhergestellt werden, wie sie zu einem bestimmten Zeitpunkt aussah. Ein Unterschied zwischen RCS und SCCS ist, dass RCS keine Checksumme in der Änderungsdatei speichert und so anfälliger für unbemerkte Defekte ist.

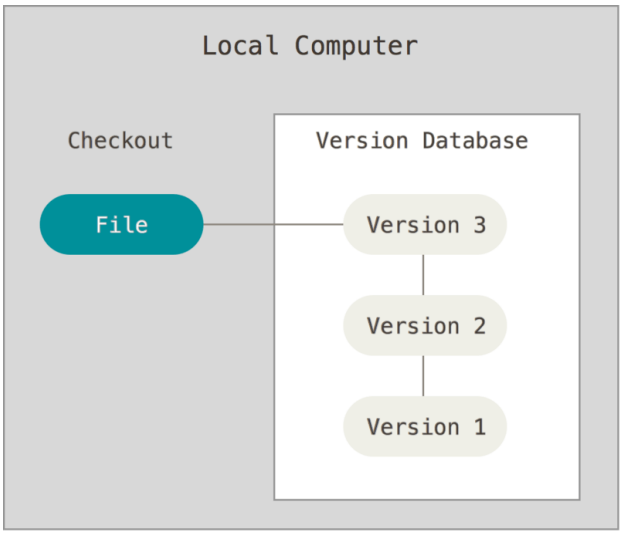

Abbildung 2 Struktur einer lokalen Versionskontrolle

Zentralisierte Versionskontrollsysteme (CVCS)

Zentralisierte Versionskontrollsysteme, kurz CVCS (Centralized Version Control Systems), wurden entwickelt um Entwicklern die Möglichkeit zu geben mit Ihresgleichen auf verschiedenen Computern mit verschiedenen Betriebssystemen zu kollaborieren. CVCS, wie beispielsweise CVS (Concurrent Versions System), Subversion oder Perforce, benötigen einen einzigen Server, auf welchem alle versionierten Dateien in einem Repository gehostet werden (siehe Abbildung 3). Dadurch können eine große Anzahl an Clients die Dateien von dem Server abholen (checkout), bearbeiten und wieder abliefern (checkin). Diese Struktur von Versionskontrollsystemen war über viele Jahre hinweg der Standard.

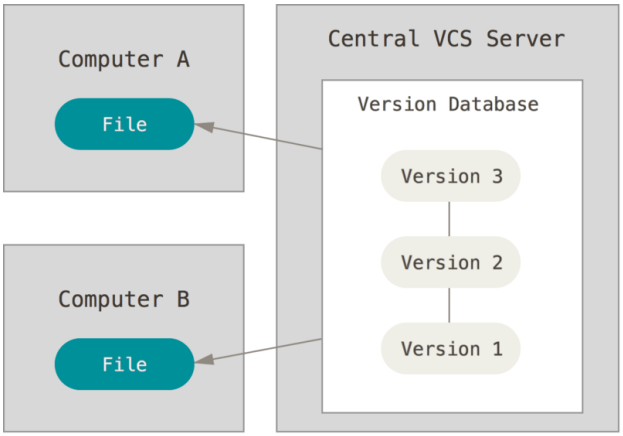

Abbildung 3 Struktur einer zentralisierten Versionskontrolle

Durch den Aufbau eines VCS mit einem zentralen Server entstehen, besonders gegenüber lokalen VCS, viele Vorteile. Beispielsweise wird die Arbeit eines Administrators stark erleichtert, da er nur an einer Stelle genau detailliert festlegen muss wer welche Dateien einsehen oder bearbeiten darf, statt lokale Repositorys auf jedem einzelnen Anwenderrechner zu verwalten. Benutzer eines solchen VCS haben auch den Vorteil, dass sie sich informieren können was andere Entwickler an einem Projekt zurzeit tun.

Zentralisierte Versionskontrollsysteme weisen ebenso wie Vorteile auch Nachteile auf. Der womöglich schwerwiegendste und offensichtlichste Nachteil ist der „Single Point of Failure“, welcher bei dem einzelnen zentralen Repository liegt. Sobald der Server ausfällt oder nicht erreichbar ist, kann für die Zeit, der Unerreichbarkeit, niemand kollaborieren oder neue Versionen einer Datei speichern, an der er oder sie gearbeitet hat. Ein schlimmeres Szenario könnte ein Defekt einer Festplatte, die die zentrale Datenbank beinhaltet, darstellen, denn ohne ordentliche Backups der Datenbank ist dann der gesamte Verlauf der Änderungen verloren, mit Ausnahme von zufällig auf Anwenderrechnern vorhandenen lokalen Kopien. Dasselbe Risiko des Datenverlusts besitzen auch lokale Versionskontrollsysteme.

Verteilte Versionskontrollsysteme (DVCS)

Verteilte Versionskontrollsysteme, kurz DVCS (Distributed Version Control Systems), beseitigen größtenteils das Risiko des Datenverlusts, welches lokale VCS und CVCS besitzen. Bei einem DVCS wird, statt mit einem zentralen Repository, mit mehrfachen Repositorys gearbeitet. Jeder Mitarbeiter, der an einem Projekt beteiligt ist, besitzt ein eigenes Repository, das mit jedem anderem abgeglichen werden kann. Außerdem können Änderungen an einem Repository lokal vorgenommen und eingesehen werden ohne eine Verbindung zu einem Server aufzubauen. Zur Verteilung eines Repository, wird es auf einen Server hochgeladen und Clients können das gesamte Repository von dort herunterladen (clone).

In solch einem Repository sind die eigentlichen Dateien und deren komplette Änderungshistorie enthalten. Sobald eine Änderung an einem lokalen Repository vorgenommen wurde, kann es mit dem ursprünglichen Repository oder jedem anderem abgeglichen werden. Um eine aktuelle Version eines Repositorys zu erhalten, muss ein Client das entfernte Repository (meistens das auf dem Server) mit seinem lokalem Repository abgleichen bzw. die neusten Änderungen herunterladen und in sein Repository einpflegen (merge).

Durch die Eigenschaft von DVCS den kompletten Datensatz auf mehreren verteilten Computern zu sichern entsteht eine Redundanz der Dateien. Aufgrund dieser Redundanz kann jedes Repository als Backup des Originals gesehen werden und dazu verwendet werden, beschädigte oder verlorengegangene Originale zu ersetzen.

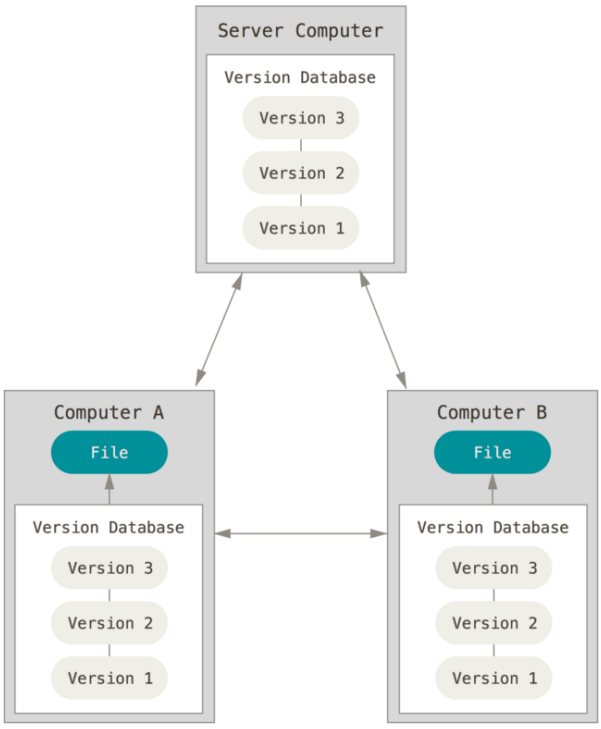

Abbildung 4 Struktur einer verteilten Versionskontrolle

Ein weiterer Vorteil verteilter Versionskontrollsystemen ist die Möglichkeit eine Datei verändern zu können, während jemand anders dieselbe Datei auch bearbeitet, ohne dass es zu einem Konflikt kommt. Sollten zwei Versionen sich widersprechen existieren sie zunächst parallel. Diese Versionen können weiterhin geändert werden und später in eine neue gemeinsame Version zusammengeführt werden. Vor allem in der Softwareentwicklung werden für einzelne Features meist separate Arbeitszweige (bei Git sog. Branches) angelegt und anschließend wieder mit dem Hauptarbeitszweig zusammengeführt. Bei größeren Projekten wird solch eine Version (bzw. Branch) oft von einer Person mit Integrator-Rolle geprüft und zusammengeführt.

Was ist Git?

Geschichte

Git ist ein weitverbreitetes verteiltes Versionskontrollsystem. Es wurde aus Notwendigkeit im Jahre 2005 von Linus Torvalds initiiert. Torvalds ist auch der Initiator und Hauptentwickler des Linux Kernels. Zwischen 1991 und 2002 wurden die Änderungen des Kernels in Form von Patches und archivierten Dateien verteilt. Nach dieser Zeit wurde das proprietäre Verteilte Versionskontrollsystem namens „*BitKeeper*“ verwendet. Da die Firma, die BitKeeper entwickelte, auf Profit ausgerichtet war, änderte sie die Lizenz. Durch diese Lizenzänderung, konnten die Entwickler des Linux-Kernels BitKeeper nicht mehr kostenlos verwenden. Aufgrund dessen blieb für viele Mitarbeiter des Open-Source-Projektes der Zugang verwehrt. Deshalb begann Torvalds im April des Jahres 2005 ein neues VCS zu entwickeln. Ziele dieses VCS waren:

- Hohe Geschwindigkeit
- Einfaches Design
- Vollständige Verteilung (DVCS)
- Fähigkeit große Projekte zu verwalten (beispielsweise den Linux-Kernel)
- Hohe Sicherheit gegen Verfälschung versionierter Dateien
- BitKeeper-ähnliche Abläufe

Der Name des DVCS ist „Git“ und bedeutet in der britischen Umgangssprache so viel wie „Blödmann“. Die Wahl dieses Namens begründet Torvalds mit:

> *“I’m an egotistical bastard, and I name all my projects after myself. First ‘Linux’, now ‘Git’.”*
>
> *„Ich bin ein egoistisches Arschloch und ich benenne all meine Projekte nach mir. Zuerst ‚Linux‘, jetzt eben ‚Git‘.“*
>
> - Linus Torvalds

> *“The joke ‘I name all my projects for myself, first Linux, then git’ was just too good to pass up. But it is also short, easy-to-say, and type on a standard keyboard. And reasonably unique and not any standard command, which is unusual.”*
>
> *„Der Witz ‚Ich benenne alle meine Projekte nach mir, zuerst Linux, nun eben Git‘ war einfach zu gut, um ihn nicht zu machen. Aber es (der Befehl) ist auch kurz, einfach auszusprechen und zu schreiben auf einer Standardtastatur, dazu einigermaßen einzigartig und kein gewöhnliches Standardkommando – sehr ungewöhnlich.“*
>
> - Linus Torvalds

Eigenschaften von Git

Git ist ein Verteiltes Versionskontrollsystem und benötigt deshalb auch nicht zwingend einen Server. Es kann kostenlos genutzt werden, ist Open-Source und der Quellcode ist auf GitHub (github.com/git/git) öffentlich einsehbar.

Datenverwaltung

Einer der größten Unterschiede zwischen Git und anderen VCS ist wie es die Dateien behandelt. Im Gegensatz zu anderen VCS (wie beispielsweise CVS, Subversion, Perforce, Bazaar usw.), die Informationen als eine fortlaufende Liste von Änderungen an Dateien („Diff“) speichern (siehe Abbildung 5), betrachtet Git die Dateien als Snapshots. Bei jedem Commit (also bei Erstellung einer neuen Version) sichert Git den Zustand aller Dateien zu diesem Zeitpunkt („Snapshot“) und speichert eine Referenz auf diesen Snapshot. Wenn eine Datei nicht verändert wurde, wird die Datei nicht erneut gespeichert, sondern lediglich eine Verknüpfung zu der vorherigen unveränderten Version gespeichert(siehe Abbildung 6).

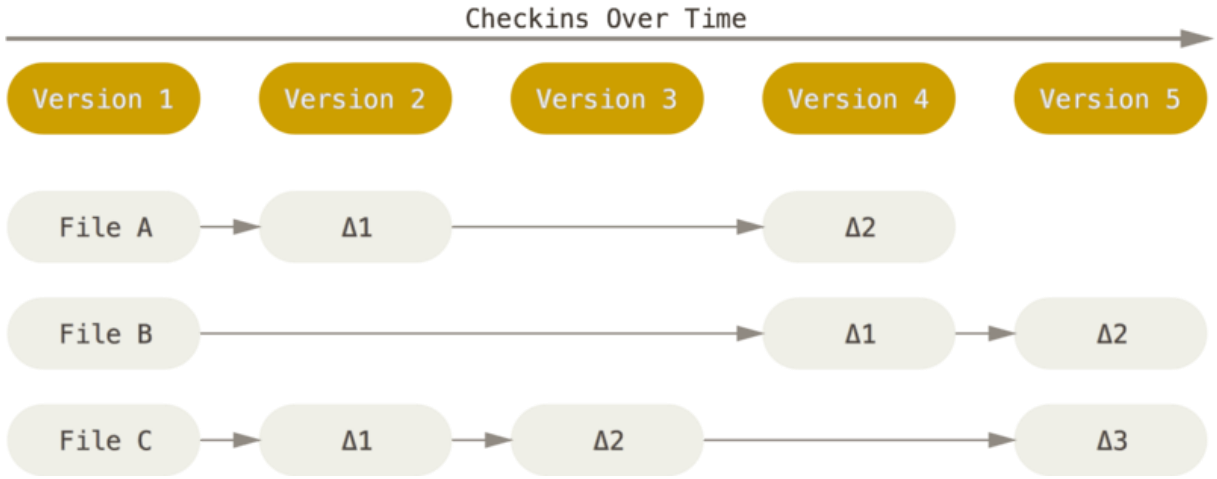

Abbildung 5 Andere VCSs speichern Daten als Änderungen (Delta) an einzelnen Dateien

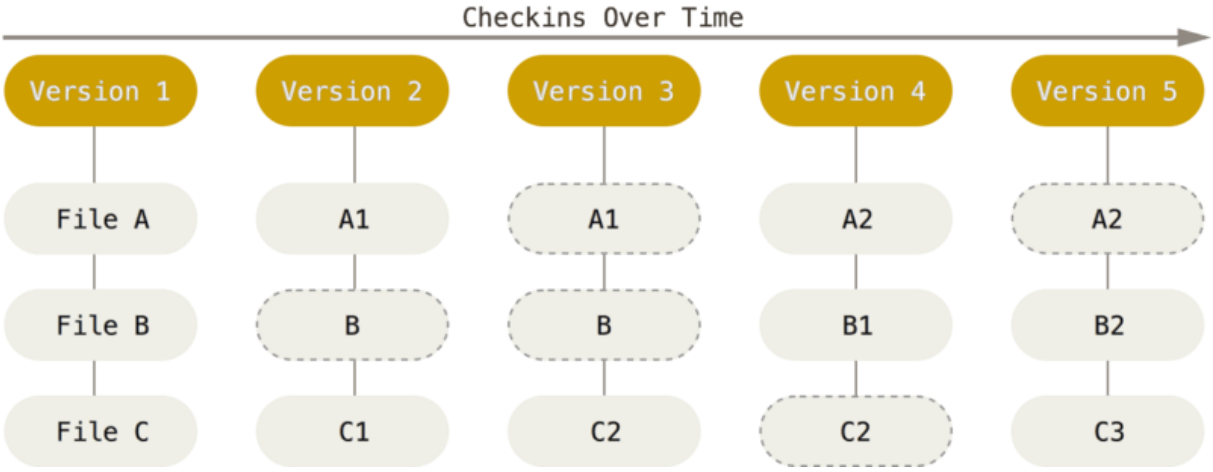

Abbildung 6 Git speichert Daten als komplette Snapshots eines Projekts

Unabhängigkeit von einem Server

Da Git sämtliche Dateien eines Projektes in einem Repository zusammenfasst und beim Kopieren von beispielsweise einem Server das komplette Repository herunterlädt (clone), ist für die meisten weiteren Operationen keine Verbindung zu einem Server vonnöten. Dadurch werden Operationen stark beschleunigt, da keine Daten von einem Server mit einer womöglich langsamen Verbindung geladen werden müssen, sondern nur das lokale Repository durchsucht werden muss. Einen weiteren Vorteil den dies mitbringt ist, dass man beispielsweise im Flugzeug oder Zug ohne Internetverbindung oder Zuhause ohne Verbindung zu einem VPN an dem Projekt arbeiten kann. Sobald wieder eine Verbindung gegeben ist, kann das lokale Repository wieder mit dem Server synchronisiert (push / pull) werden.

Integrität und Sicherheit der Daten

Git verwendet zur Sicherstellung der Integrität der Dateien den SHA-1 Hash-Algorithmus. Sobald eine Änderung vorgenommen wurde, berechnet Git den SHA-1 Hash (40 Zeichen lange hexadezimale Zeichenkette) der veränderten Dateien, bevor diese gespeichert werden. Jede Datei sollte also einen einzigartigen Hashwert besitzen[1], welcher von Git zum Referenzieren der Commits verwendet wird. Sobald eine Änderung an einer Datei vorgenommen oder sie beschädigt wurde, wird dies von Git registriert, da sich, bei nur einer kleinen Änderung, der komplette Hashwert ändert.
So sieht beispielsweise ein SHA-1 Hashwert von dem Wort „Git“ aus:

5819778898df55e3a762f0c5728b457970d72cae

Sobald das Wort nur leicht geändert wird entsteht ein komplett neuer Hashwert. Zum Beispiel der SHA-1 Hashwert von dem selben Wort in Großbuchstaben („GIT“) sieht so aus:

406750c04c6d320640c2a7c8d2c4b52d5b9965bf

[1] https://shattered.io/

Zustände einer Datei

Wie in der *Einleitung* schon erwähnt wurde, arbeitet Git mit drei Zuständen einer Datei. Sie kann *modified* ,*staged* oder *commited* sein. Dateien, die modified sind, werden oder wurden bearbeitet, aber die Änderungen sind noch nicht abgeschlossen. Dateien, die staged sind, sind geänderte Dateien, die für den nächsten Commit vorgesehen ist. Der dritte Zustand ist „commited“ und bedeutet, dass die Dateien in dem Repository gespeichert sind.

In dem Git Verzeichnis (siehe Abbildung 1) werden Git Metadaten und die lokale Datenbank für ein Projekt gespeichert. Beim Klonen (clone) eines Repositorys wird dieser Teil kopiert. Das Arbeitsverzeichnis ist ein Check-out einer Datei, in dem sie bearbeitet werden kann. Die Staging Area beschreibt eine Datei, die normalerweise im *„.git“*-Verzeichnis zu finden ist und beinhaltet, welche Änderungen beim nächsten Commit in das Git Verzeichnis gespeichert werden sollen.

Grundlegende Benutzung von Git

Die folgenden Beispiele basieren auf das Git CLI (Command Line Interface).

Repository erstellen

Um ein Git Repository zu erstellen gibt es zwei Möglichkeiten. Man kann entweder den Befehl *„git init“* oder *„git clone“* verwenden. Im Folgenden werden beide erklärt und ein Beispiel dafür angeführt.

Init

Mit dem folgenden Befehl kann ein lokales Git Repository initiert werden.

```
git init [project name]
```

Beim Initiieren eines Git Repositorys wird ein versteckter Ordner mit dem Namen *„.git“* angelegt, in welchem die Daten, die Git benötigt, gespeichert werden. Bevor man ein Repository erstellt, sollte man einen leeren Ordner erstellen oder optional den Projektnamen (*project name*) angeben, damit automatisch ein neuer Ordner mit dem angegebenen Namen erstellt wird. Der Projektname darf allerdings keine Leezeichen enthalten.

```
johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample
$ git init exampleRepo
Initialized empty Git repository in C:/Users/johan/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo/.git/

johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample
$ ls
exampleRepo/
```

Abbildung 7 Screenshot von 'git init'

Clone

Mit dem folgenden Befehl kann ein schon bestehendes Git Repository aus einem anderen Pfad, von einem anderen Rechner oder beispielsweise aus einem GitHub-Repository kopiert werden.

```
git clone /path/to/repository [optional new project name]
git clone <URL_of_a_Repository>
git clone username@host:/path/to/repository
```

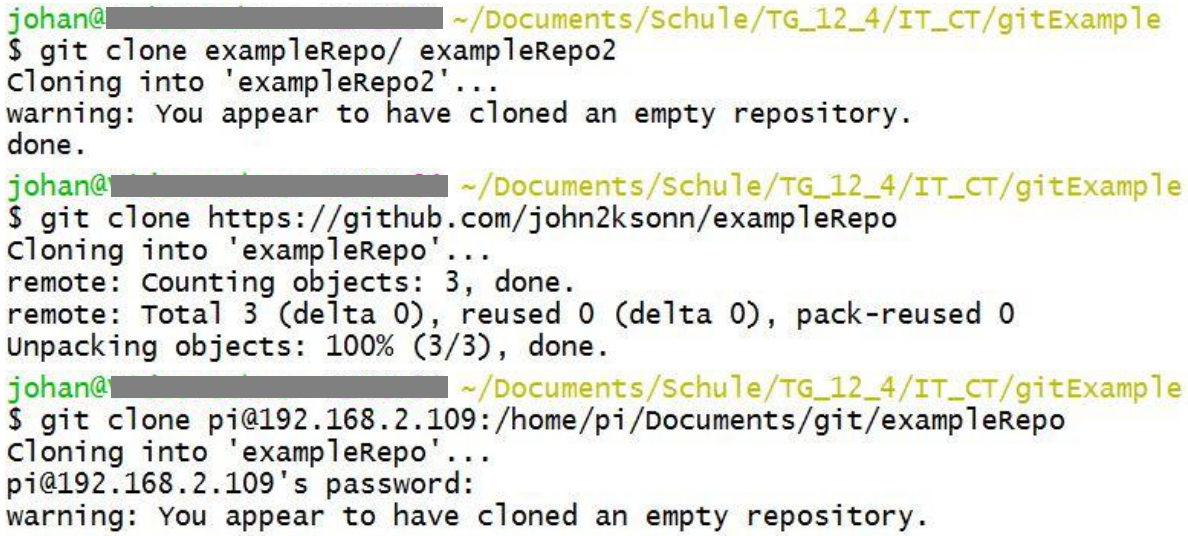

Abbildung 8 Screenshots von 'git clone'

Dateien hinzufügen und löschen

Add

Mit dem folgenden Befehl kann man eine oder mehrere Dateien zur Staging Area hinzufügen bzw. für den nächsten Commit merken.

```
git add <one or more filenames>   (only adds the specified files)
git add *                         (adds all changed files)
```

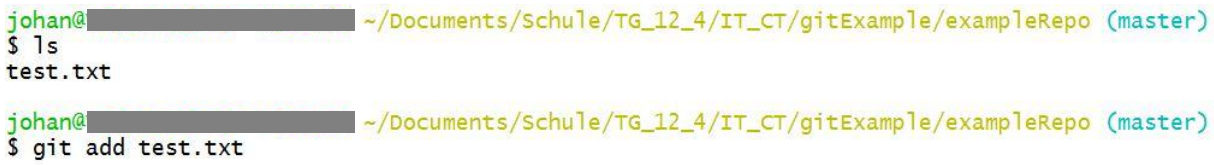

Abbildung 9 Screenshot von 'git add'

Status

Mit dem folgenden Befehl kann der Status des Repositorys angezeigt werden. Neben generellen Informationen zum Status des Repositorys werden auch alle *„staged“* Dateien ausgegeben.

```
git status
```

```
johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   test.txt
```

Abbildung 10 Screenshot von 'git status'

Remove (rm)

Mit dem folgenden Befehl kann eine *„staged“* Datei *„unstaged“* gemacht werden (--cached) oder eine Datei aus dem Repository komplett gelöscht werden.

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

```
git rm <filename>
```

```
johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo (master)
$ git rm --cached test.txt
rm 'test.txt'
```

Abbildung 11 Screenshot von 'git remove'

Committing und resetting

Commit

Mit dem folgenden Befehl kann man einzelne oder alle *„staged“* Dateien commiten. Beim Commiten muss eine *Commit message* angegeben werden, die die Änderungen beschreiben soll.

```
git commit –m “Commit message”
```
(commits all staged files)

```
git commit –am “Commit message”
```
(commits all changed files)

```
johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo (master)
$ git commit -am "Modified test.txt"
[master 5d53f67] Modified test.txt
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Abbildung 12 Screenshot von 'git commit'

Reset

Mit dem folgenden Befehl kann man eine *„staged“* Datei *unstagen* machen oder alles auf den letzten Commit zurücksetzen.

```
git reset /path/to/file          (unstages a staged file)
git reset --hard                 (reverts everything to last Commit)
```

```
johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo (master)
$ git reset test.txt
Unstaged changes after reset:
M       test.txt

johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo (master)
$ git reset --hard
HEAD is now at 5d53f67 Modified test.txt
```

Abbildung 13 Screenshot von "git reset"

Branching

Branches sind Arbeitszweige die neben der Hauptentwicklungslinie bzw. dem Hauptzweig bestehen können. Solche Branches werden angelegt um beispielsweise ein neues Feature zu implementieren. Nach der Entwicklung des Features, kann der Feature-Branch mit dem Haupt-Branch wieder zusammengeführt (merging) werden.

Branch und checkout

Mit den folgenden Befehlen kann man einen Branch erstellen und/oder zu einem Branch wechseln.

```
git branch <branchname>          (creates a new branch)
git checkout <branchname>        (switches to <branchname>)
```

Kurzverion:

```
git checkout -b <branchname>     (creates a new branch & switches to it)
```

```
johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo (master)
$ git checkout -b newBranch
Switched to a new branch 'newBranch'

johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo (newBranch)
$
```

Abbildung 14 Screenshot von 'git checkout'

Mit dem folgenden Befehl kann man einen Branch löschen.

```
git branch -d <branchname>
```

```
johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo (master)
$ git branch -d newBranch
Deleted branch newBranch (was 5d53f67).
```

Abbildung 15 Screenshot von 'git branch'

Merging Branches

Mit dem folgenden Befehl kann man einen Branch mit dem aktuellem zusammenführen.

```
git merge <branchname>
```

```
johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo (master)
$ git merge newBranch
Updating 5d53f67..1bff6b1
Fast-forward
 test.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Abbildung 16 Screenshot zu 'git merge'

Pull und push

Pull

Mit dem folgenden Befehl kann man die letzten Änderungen des ursprünglichen Repositorys herunterladen und mit dem aktuellen zusammenführen (merging).

```
git pull
```

```
johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo (master)
$ git pull
pi@192.168.2.109's password:
remote: Zähle Objekte: 3, Fertig.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From 192.168.2.109:/home/pi/Documents/git/exampleRepo
 * [new branch]      master     -> origin/master
```

Abbildung 17 Screenshot von 'git pull'

Push

Mit dem folgenden Befehl kann man die letzten Änderungen eines Branches des lokalen Repositorys in das ursprüngliche Repository hochladen und zusammenführen (merging).

```
git push origin <branchname>
```

```
johan@[redacted] ~/Documents/Schule/TG_12_4/IT_CT/gitExample/exampleRepo (newBranch)
$ git push origin newBranch
pi@192.168.2.109's password:
Permission denied, please try again.
pi@192.168.2.109's password:
Counting objects: 3, done.
Writing objects: 100% (3/3), 245 bytes | 245.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To 192.168.2.109:/home/pi/Documents/git/exampleRepo
 * [new branch]      newBranch -> newBranch
```

Abbildung 18 Screenshot von 'git push'

Sonstige Befehle

Log

Mit dem folgenden Befehl kann man alle Commits eines Repositorys auflisten.

```
git log
```

Help

Falls man Hilfe benötigt, kann man mit „git --help“ eine Übersicht aller Git Befehle erlangen. Mit „git <*command*> -h“ wird eine kurze Erklärung der möglichen Optionen in der Konsole ausgegeben und mit „git <*command*> --help“ wird eine Manual Page im Browser als HTML-Dokument geöffnet.

Git Cheat Sheet

Repository erstellen

- `git init [project-name]`
  - Initiiert ein lokales Repository mit gegebenem Namen
- `git clone [url]`
  - Lädt ein Repository von der gegebenen URL herunter

Änderungen vornehmen

- `git status`
  - Listet alle neuen oder bearbeitete Dateien auf
- `git diff`
  - Zeigt Änderungen der nicht-staged Dateien
- `git add *`
  - Stages alle geänderten Dateien
- `git reset [file]`
  - Setzt staged Dateien auf nicht-staged, aber behält deren Änderungen im Arbeitsverzeichnis bei
- `git commit -m "[commit message]"`
  - Alle staged Dateien werden mit der angegebenen *commit message* commited
- `git push`
  - Lädt lokale Änderungen in das ursprüngliche Repository
- `git pull`
  - Lädt die alle neuen Änderungen aus dem ursprünglichen Repository

Branching

- `git branch`
  - Listet alle lokalen Branches des aktuellen Repositorys auf
- `git branch [branch-name]`
  - Erstellt neuen Branch mit *branch-name* als Name
- `git checkout [branch-name]`
  - Wechselt zu dem angegebenen Branch
- `git merge [branch-name]`
  - Kombiniert den aktuellen Branch mit dem angegebenen
- `git branch -d [branch-name]`
  - Löscht den angegebenen Branch

Commits zurücksetzen

- `git reset [commit]`
  - Macht alle Commits nach dem gegebenen Commit rückgängig

Vergleich mit Alternativen Versionskontrollsysteme

VCS / Kriterium	VCS-Typ	Checksumme wird verwendet	funktioniert auch ohne Server	Kollaboration möglich	OS	Open-Source
RCS	Lokales VCS	✘	✔	✘	?	✔
SCCS	Lokales VCS	✔	?	?	Win/Mac/ Linux/…	✘
CVS	CVCS	?	✘/✔[1]	✔	Win/Mac/ Linux	✔
Subversion	CVCS	✔	✘	✔	Win/Mac/ Linux/…	✔
Perforce	CVCS	✔ (MD5)	?	✔	Win/Mac/ Linux/…	✘
Git	DVCS	✔ (SHA-1)	✔	✔	Win/Mac/ Linux	✔
Bazaar	DVCS	?	✔	✔	Win/Mac/ Linux/…	✔
Mercurial	DVCS	✔ (SHA-1)	✔	✔	Win/Mac/ Linux/…	✔
Darcs	DVCS	✔	✔	✔	Win/Mac/ Linux/…	✔

[1]Server und Client kann auf demselben Computer installiert sein.

GitHub

GitHub ist eine Internet-Plattform, auf welcher man Git Repositorys veröffentlichen kann und welche es erleichtert mit anderen Benutzern an Projekten zusammenzuarbeiten. Durch die Möglichkeit seine Repositorys durch das Internet zu erreichen, kann man relativ einfach von nahezu überall darauf zugreifen und ohne großen Aufwand über mehrere Computer verteilen. Sie ist unter www.github.com zu finden. Ohne einen GitHub-Account können alle öffentlichen Repositorys eingesehen und durchsucht werden. Nachdem man einen kostenlosen Account erstellt hat, gibt es die Möglichkeit zu bereits bestehenden Open Source Projekten beizutragen oder selber ein Projekt zu starten, indem man ein eigenes Repository veröffentlicht. Öffentliche Repositorys können von jedem Benutzer heruntergeladen & bearbeitet werden und bei Änderungen diese dem Besitzer des ursprünglichen Repositorys vorgeschlagen werden. Diese Änderungen können entweder in das ursprüngliche Repository übernommen oder abgelehnt werden. Um private Repositorys auf GitHub verwalten zu können, kann für unter zehn Euro im Monat (für Schüler und Student kostenlos) ein „Developer-Account“ erworben werden. Neben dem „Developer-Account“ können ebenso andere Accounts erworben werden, die eher für Teams oder Firmen gedacht sind.

Fazit/eigene Meinung

Abschließend möchte ich meine eigene Meinung zum Ausdruck bringen und kommentieren, ob es sinnvoll ist ein Versionskontrollsystem, wie Git, zu verwenden.

Ich persönlich benutze Git und GitHub seit Anfang 2016 selber und kann es einer Person, die ein Werkzeug zu Quellcode-Verwaltung und –Versionierung sucht, auf jeden Fall empfehlen. Anfangs mag es recht kompliziert wirken, aber mit ein bisschen Erfahrung kann man sich gut daran gewöhnen Git regelmäßig zu verwenden. Außerdem ist Git weitverbreitet und hat eine große Community, was zufolge hat, dass man bei Fragen durch kurze Internetrecherche viele Antworten finden kann. Wenn man sich nicht alle Befehle merken kann, ist das nicht schlimm, da man beispielsweise leicht auf einem Cheat Sheet, wie es auf Seite 16 oder massenhaft im Internet zu finden ist, nachschauen kann.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig angefertigt und alle verwendeten Quellen mit Datum des letzten Aufrufs aufgelistet habe.

Vogt, 28.11.2017, Johannes Sonn

Quellverzeichnis

Bücher & E-Books

https://git-scm.com/book/de/v1 (26.11.201)

https://git-scm.com/book/en/v2 (26.11.2017)

Websites

https://de.wikipedia.org/wiki/Versionsverwaltung (26.11.2017)

https://de.wikipedia.org/wiki/Git (26.11.2017)

https://en.wikipedia.org/wiki/Git (26.11.2017))

https://de.wikipedia.org/wiki/Source_Code_Control_System (26.11.2017)

https://en.wikipedia.org/wiki/Source_Code_Control_System (26.11.2017)

https://de.wikipedia.org/wiki/Revision_Control_System (26.11.2017)

https://en.wikipedia.org/wiki/Revision_Control_System (26.11.2017)

https://de.wikipedia.org/wiki/Concurrent_Versions_System (26.11.2017)

https://en.wikipedia.org/wiki/Concurrent_Versions_System (26.11.2017)

https://de.wikipedia.org/wiki/Apache_Subversion (26.11.2017)

https://en.wikipedia.org/wiki/Apache_Subversion (26.11.2017)

https://de.wikipedia.org/wiki/Perforce (26.11.2017)

https://en.wikipedia.org/wiki/Perforce_Helix (26.11.2017)

https://de.wikipedia.org/wiki/Bazaar (26.11.2017)

https://en.wikipedia.org/wiki/GNU_Bazaar (26.11.2017)

https://de.wikipedia.org/wiki/Mercurial (26.11.2017)

https://en.wikipedia.org/wiki/Mercurial (26.11.2017)

https://de.wikipedia.org/wiki/Darcs (26.11.2017)

https://en.wikipedia.org/wiki/Darcs (26.11.2017)

http://darcs.net (26.11.2017)

Bilder

https://github.com/progit/progit2/tree/master/images (26.11.2017)