HTW Chur
Institut für Informations- und Kommunikationstechnologien

Git: Einführung

Lukas Toggenburger
lukas.toggenburger@htwchur.ch

Ziele von Versionsverwaltungssystemen

- Änderungen an Dateien (insbesondere Quellcode) dokumentieren
- Änderungen rückgängig machen (Rückkehr zu einer älteren Version)
- Änderungen einem Benutzer zuordnen («jemandem die Schuld geben»)

Ziele dieser Session

- Änderungen an Dateien versioniert ablegen und mit einer zugehörigen Nachricht versehen
- Entwicklungszweige anlegen und handhaben können

Bekannte Versionsverwaltungssysteme

- CVS
- Subversion (SVN)
- ClearCase
- Bazaar
- Mercurial
- BitKeeper
- Git

Geschichte

Entstehung 2005.

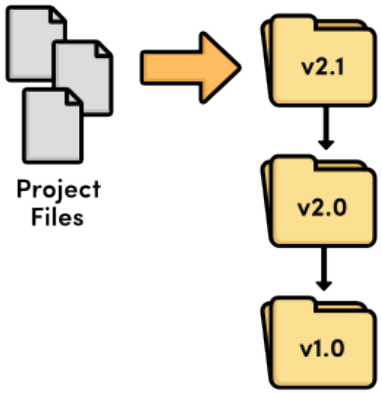
Linux Community verlor die kostenlose Lizenz zur Benutzung von BitKeeper.

Das Git-Grundsystem wurde angeblich in zwei Wochen von Linus Torvalds programmiert.

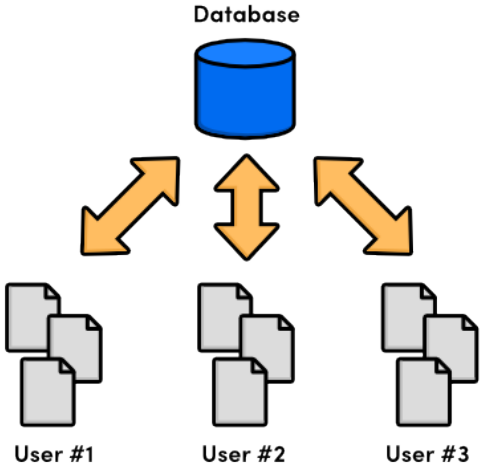
«Git» ist englisch und heisst Blödmann.

Konzepte für Versionskontrolle

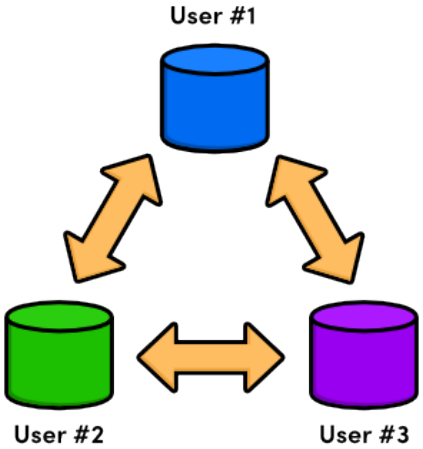
http://rypress.com/tutorials/git/introduction



Versionskontrolle mit mehreren Verzeichnissen



Zentrales System (z.B. SVN)



Verteiltes System (z.B. Git)

Bei Git hat jeder Benutzer eine vollständige Kopie des gesamten Datenstands, inkl. allen alten Versionen.

Installation

Linux

```
sudo apt-get install git bzw. yum install git
```

Windows

http://git-scm.com/download/win

MacOS

https://git-scm.com/download/mac

Arbeiten mit dem Terminal

Das Arbeiten mit Git kann auf zweierlei Arten erfolgen:

- Mittels grafischen Oberflächen (GUIs)
- Mittels Text-Befehlen (über die Kommandozeile)

Grafische Oberflächen können in einer Entwicklungsumgebung (z.B. Eclipse) integriert oder auch als alleinstehende Software realisiert sein. Nachteilig ist, dass GUIs sich tendenziell häufiger ändern und nicht auf allen Betriebssystemen gleichermassen verfügbar sind.

Entsprechend konzentriert sich die vorliegende Einführung ausschliesslich auf die Bedienung mittels Text-Befehlen über die Kommandozeile (auch Terminal, Shell, Konsole oder Eingabeaufforderung genannt).

Ubuntu, Mac OS: Terminal öffnen

Ubuntu

 oder Ctrl-Alt-T (Text einfügen mit Shift-Ctrl-v)

Mac OS

⌘ + Leerschlag ; Terminal suchen

Windows: Git Bash Konsole öffnen

Unter Windows sollte zur Eingabe von Git-Befehlen nicht die Windows-Eingabeaufforderung verwendet werden, sondern die Git Bash Konsole (im Git-Installer integriert).

Man erreicht diese wie folgt:

Ordner im Explorer öffnen → Rechtsklick → «Git Bash Here»

Text aus der Zwischenablage kann mittels «Insert» (allenfalls «Shift-Insert») eingefügt werden.

Innerhalb der Git Bash Konsole funktioniert die Bedienung wie auf einem Unix-System.

Arbeiten mit dem Terminal

Dies sind die wichtigsten Befehle zur Handhabung von Dateien mittels Terminal:

Leeres Verzeichnis anlegen: `mkdir meinverzeichnis` (make directory)
Verzeichnis öffnen: `cd meinverzeichnis` (change directory)
Leere Datei erstellen: `touch meinedatei`
Datei bearbeiten: `nano meinedatei`
Verzeichnis-Inhalte anzeigen: `ls -al` (list)
Aktuelles Verzeichnis anzeigen: `pwd` (print working directory)
Verzeichnis verlassen: `cd ..`
Datei oder Verzeichnis löschen: `rm -rf meinedaten` (Keine Rückfrage!)

Initiale Konfiguration

Bevor wir mit Git zu arbeiten beginnen, sollten wir einige Einstellungen vornehmen. Unter Unix-Systemen sollen diese und alle weiteren Befehle mit dem gewöhnlichen Benutzeraccount (also nicht als root) durchgeführt werden.

Name / Mail-Adresse des Benutzers setzen

(Diese Angaben dienen dazu, nachvollziehen zu können, wer welche Änderung am Datenstand gemacht hat.)

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

Initiale Konfiguration

Git bunt machen

In einigen Anwendungsfällen ist es hilfreich, farbigen Output zu haben. Solcher lässt sich aktivieren mit:

```
git config --global color.ui "auto"
```

Warnung beim Hochladen von Änderungen vermeiden

Die folgende Konfiguration verhindert eine Warnung beim Hochladen (pushen) von Änderungen:

```
git config --global push.default simple
```

(Erklärung: Bei Verwendung von `git push` nicht alle Branches, sondern nur den aktuell gewählten hochladen.)

Initiale Konfiguration

Einen Texteditor definieren

In einigen Fällen wird für gewisse Eingaben ein Text-Editor benötigt. Es sollte festgelegt werden, welches Programm dafür verwendet werden soll. Standardmässig wird dafür auf allen Plattformen `vim` gestartet, der zwar mächtig ist, aber oft als wenig intuitiv wahrgenommen wird.

```
git config --global core.editor "nano"    (Linux und MacOS)
git config --global core.editor notepad   (Windows)
```

Einstellungen prüfen

```
git config --list
```

Falsche getippte Einträge entfernen

```
git config --global --unset-all userr.name
```

GitHub

GitHub (https://github.com/) ist eine Firma, welche öffentlich einsehbare Git-Repositories kostenlos zur Verfügung stellt. (Private Repositories sind kostenpflichtig.)

Zusätzlich bietet sie Dienstleistungen an (z.B. Issue-Tracker, Pull-Requests, Wiki, …) die nicht im Umfang von Git enthalten sind.

Git und GitHub sollten nicht verwechselt werden!

GitHub-Account erstellen

Obwohl Git auch ausschliesslich lokal verwendet werden kann, wird man üblicherweise Änderungen andern zugänglich machen wollen. Erstellen Sie deshalb (sofern nicht schon vorhanden) einen Account bei Github:

https://github.com/join

Benutzername und Email können unabhängig von den Einstellungen im Git-Client gewählt werden.

Die Wahl eines Benutzernamens der ähnlich zum Realname ist (z.B. `vorname.nachname`), vereinfacht die Zusammenarbeit.

Neues GitHub-Repository anlegen

Als nächster Schritt wird ein Repository (eine Datenablage) angelegt:

https://github.com/new

Option: **«Public»** wählen. Diese Repositories sind kostenlos und von jedermann einsehbar.

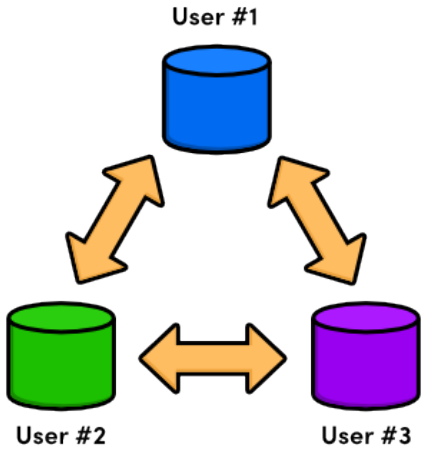
Option **«Initialize this repository with a README»** wählen. Dadurch erstellt Github automatisch eine README (liesmich) Datei. Diese Datei wird übrigens auch angezeigt, wenn das Repository mittels Browser betrachtet wird.

Ein Repository klonen

Als nächsten Schritt werden wir das Github-Repository klonen. Dadurch erhalten wir eine vollständige lokale Kopie, welche wir im weiteren Ablauf auch synchron halten werden. (Ein Repository entspricht einem farbigen «Kübel» in der Abbildung.) Es gibt also in unserem Fall *zwei* Repositories!

Sollte Github pleitegehen und alle Server abschalten, hätten wir sämtliche alten Datenstände inkl. Versionsgeschichte auch lokal vorhanden und könnten mit relativ geringem Aufwand zu einem andern Anbieter wechseln. Bei andern Systemen, z.B. SVN ist das nicht so.

http://rypress.com/tutorials/git/introduction



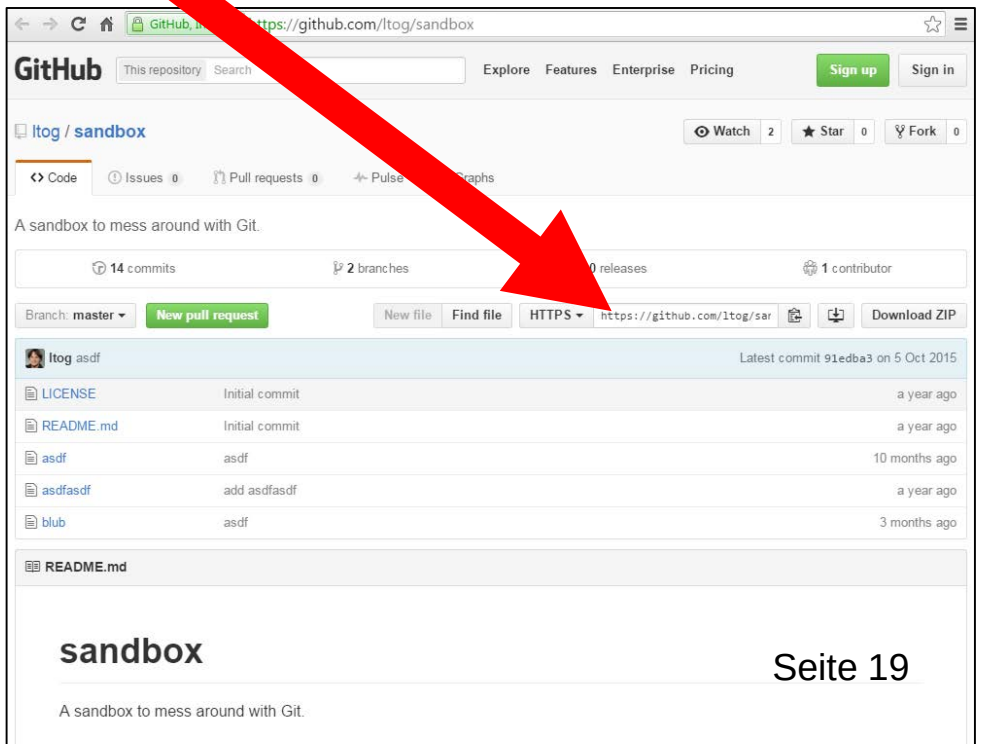
Ein Repository klonen

Das neu erstellte Repository mit folgendem Befehl klonen:

`git clone https://github.com/`*`user/repository`*`.git `*`meinverzeichnis`*

Der Befehl erstellt ein neues Verzeichnis `meinverzeichnis`.

Innerhalb dieses neu entstandenen Verzeichnisses können Dateien durch Git versioniert werden.



Commits

Ein Commit ist eine im Repository definierte Änderung an Dateien, welche von einem Benutzer durchgeführt wurde.

Eine Liste von Commits kann angezeigt werden mit:

```
git log
```

Die Ausgabe bezieht sich auf das lokale Repository, allenfalls gibt es auf Github aber schon neuere Commits. Diese wären dann mit `git log` nicht sichtbar.

Die Commit History anzeigen

Weshalb zeigt `git log` schon einen (lokalen) Commit an?

```
commit 95cadb03c15724fbbd380ad3dbb920fe7465722a
Author: ltog <lukas.toggenburger@htwchur.ch>
Date:   Mon Feb 23 17:45:31 2015 +0100

    Initial commit
```

Dieser Commit wurde von Github in unserem Namen automatisch erstellt, da wir die Option «Initialize this repository with a README» angewählt haben.

Commit Hashes

Jeder Commit entspricht einem bestimmten Stand der Datenablage und erhält einen SHA-1 Hash (eine Art digitaler Fingerabdruck), z.B.
`521f009151a23f98fb6de4eb68fee2207fe08cd3`,
z.T. auch abgekürzt mit den ersten paar Zeichen (von links):
`521f009`

Dieser hasht (verrechnet) die aktuelle Änderung, inkl. Datum, Autor, etc. sowie die komplette Versionsgeschichte.

Der Commit Hash kann später auch verwendet werden um einen bestimmten Commit, bzw. einen bestimmten Software-Stand zu bezeichnen.

Fortlaufende Nummern wie die revision number bei SVN gibt es bei Git nicht.

Neue Datei hochladen

Als nächsten Schritt wollen wir eine neue Datei erstellen, im lokalen Repository ablegen und anschliessend den Stand vom lokalen Repository zum Github-Repository übertragen.

Auf den nächsten paar Folien folgt ein wenig Theorie dazu.

Als Zwischenschritt fürs Ablegen im lokalen Repository tritt die Staging Area, (manchmal auch Index genannt) auf. In der Staging Area werden Änderungen vorgemerkt, welche in Git abgelegt werden sollen.

Neue Datei hochladen: Datei-Zustandsdiagramm

Aus Sicht von Git kann eine lokale Datei in 4 unterschiedlichen Zuständen sein.

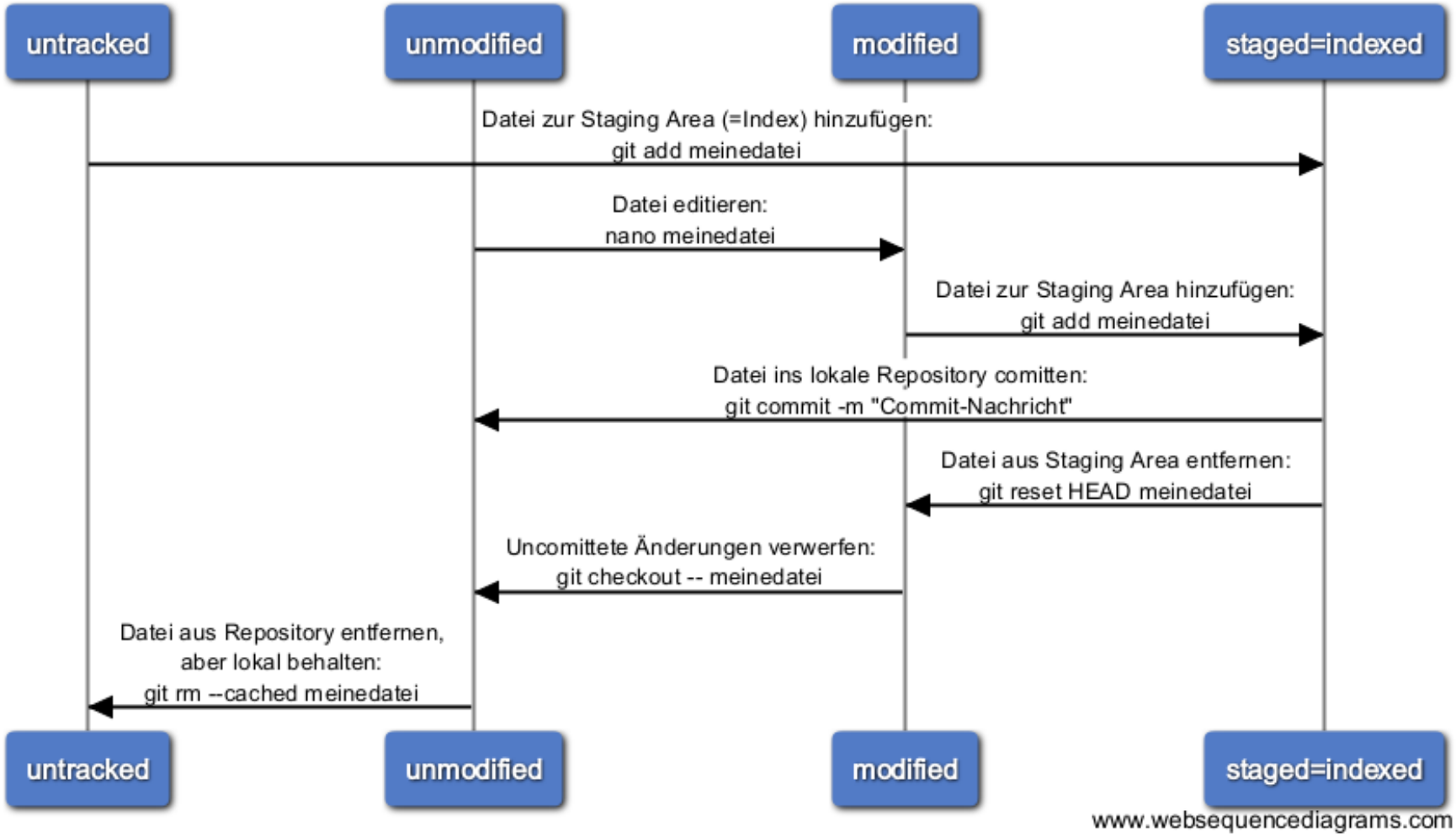
Zuerst einmal wird unterschieden, ob eine Datei überhaupt von Git versioniert/verfolgt wird oder nicht. Ist eine Datei nicht verfolgt, also *untracked*, übernimmt Git keine Verantwortung für diese Datei. Wird eine solche gelöscht, ist sie unwiederbringlich weg.

Dateien, welche exakt so vorliegen, wie sie auch comittet (in Git gespeichert) wurden, haben den Zustand *unmodified*.

Abgeänderte Dateien haben den Zustand *modified*.

Und schliesslich gibt es noch abgeänderte Dateien, welche zum Commit vorgesehen sind, im Zustand *staged* = *indexed*.

Neue Datei hochladen: Datei-Zustandsdiagramm



Neue Datei hochladen: Dateizustände anzeigen

`git status`

zeigt für jede Datei an, in welchem Zustand sie ist:

- *untracked*
- *(unmodified)*
- *modified*
- *staged*

Der Zustand *unmodified* wird als Standard angenommen und deshalb nicht angezeigt.

Zusätzlich gibt `git status` gute Hinweise, mit welchen Kommandos die Zustände einzelner Dateien geändert werden können.

Neue Datei hochladen: Dateizustände anzeigen

Eine beispielhafte Ausgabe von `git status` :

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#         new file:   staged_file
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#         modified:   modified_file
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#         untracked_file
```

Neue Datei hochladen: Datei erstellen/stagen

Als erstes muss die hochzuladende (Text-)Datei angelegt werden. Das kann mit jedem Text-Editor erfolgen. Damit Git die Datei verarbeiten kann, muss sie sich innerhalb des geklonten Verzeichnisses befinden.

`nano meine_datei` / Erstellen durch Notepad, etc.

Neue Datei stagen = zum Index hinzufügen:

```
git add meine_datei
```

Dadurch deklarieren wir, dass die Datei im nächsten Schritt comittet werden soll.

Die Ausgabe von `git status` bestätigt uns, dass die Datei gestaget wurde.

Neue Datei hochladen: Commit durchführen

Erst mit dem folgenden Schritt wird die Datei ins (lokale) Repository comittet (=versioniert abgelegt).

Zu einem Commit gehört immer auch eine kurze, aber prägnante Beschreibung, welche Änderung durchgeführt wurde.

```
git commit -m "Hinzugefügt: meine_datei"
```

Neue Datei hochladen: Commit durchführen

Alternativ zur direkten Eingabe der Nachricht kann auch nur

`git commit`

ausgeführt werden.

Anschliessend öffnet sich ein Text-Editor, in welchem die Commit-Nachricht eingegeben werden kann. (Zeilen mit Raute (#) werden ignoriert.) Geöffnet wird eine temporäre Datei, in welche die Nachricht geschrieben werden soll. Nach Speichern der Datei und Schliessen des Text-Editors wird die Nachricht übernommen und die temporäre Datei gelöscht.

In nano erfolgt das Speichern und Schliessen über `Ctrl-x`. (Der vorgeschlagene Name zum Speichern der Datei soll dabei übernommen werden.)

Neue Datei hochladen: Änderung pushen (=hochladen)

Nach Durchführen des Commits sind alle Dateien im Zustand *unmodified* und `git status` zeigt keine Dateien an.

Dafür zeigt uns `git log` an, dass ein neuer Commit durchgeführt wurde.

Der Commit ist bis jetzt erst im lokalen Repository abgelegt. Mit dem folgenden Befehl werden gemachte Änderungen auch auf das Github-Repository übertragen:

```
git push
```

Man wird nun aufgefordert, sich mit den Github-Accountdaten zu authentisieren.

Datei abändern

Im nächsten Schritt soll die vorher abgelegte Datei abgeändert und die Änderung wieder aufs Github-Repository hochgeladen werden.

Wenn man an mehreren Rechnern oder mit mehreren Benutzern arbeitet, sollte man sich angewöhnen, vor dem Anbringen von Änderungen das lokale Repository auf den Stand des remote (hier Github-) Repositorys zu bringen. So vermeidet man Situationen, wo auseinandergehende Entwicklungen wieder zusammengeführt werden müssen. Der Befehl dafür lautet:

```
git pull
```

Bis hierhin hat der Befehl aber noch keinen Effekt, da nicht von mehreren Repositories aus auf das Github-Repository comittet wurde.

Datei abändern

Gemäss dieser Anleitung können Änderungen vorgenommen und publiziert werden:

Änderungen vom Server herunterladen um Merges und Konflikte zu vermeiden: `git pull`

Datei ändern: `nano meine_datei` (Windows: Notepad, etc.)

Datei stagen: `git add meine_datei`

Änderung committen: `git commit -m "Komplette Überarbeitung"`

Änderung aufs Github-Repository hochladen: `git push`

Probleme beim Hochladen

Grundsätzlich sollten Sie nun in der Lage sein, fortlaufend Änderungen an Dateien mittels Git zu hinterlegen.

Es folgt eine Beschreibung von zwei Problemen, welche sie beim Arbeiten mit mehr als einem lokalen Repository früher oder später antreffen werden.

Beide Probleme treten dann auf, wenn an zwei lokalen Repositories (z.B. von unterschiedlichen Benutzern) quasi-gleichzeitig Änderungen vorgenommen werden. Beispielsweise könnten Sie daran sein, eine neue Funktion zu programmieren während Ihr Kollege eine andere Änderung auf Github pusht.

Stellen Sie die Situation nach, indem sie ihr Git-Repository ein zweites Mal klonen. Achten Sie darauf, dass sie den clone Befehl ausserhalb des bisherigen git-Verzeichnisses ausführen:

`git clone https://github.com/`*`user/repository`*`.git `*`meinverzeichnis2`*

Paralleles Pushen

Erstellen Sie in beiden geklonten Repositories eine neue Datei mit unterschiedlichem Namen und versuchen Sie die Datei zu pushen (verwenden Sie ausnahmsweise `git pull` nicht).

Beim zweiten `git push` werden Sie ungefähr folgende Fehlermeldung erhalten:

```
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/ltog/sandbox.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first merge the remote changes (e.g.,
hint: 'git pull') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Paralleles Pushen

Die Fehlermeldung erscheint, weil Git feststellt, dass serverseitig Änderungen von zwei lokalen Repositories kombiniert werden sollen, welche nie lokal zusammen getestet wurden. (Die Änderungen könnten Programme sein, welche nicht zueinander kompatibel sind.)

Git verwehrt berechtigterweise diese Operation. Das zweite lokale Repository muss zuerst den Commit vom Github-Repository mergen (übernehmen):

```
git pull
```

Das Mergen wird als neuer (lokaler) Commit, ein sogenannter Merge-Commit, aufgefasst. Entsprechend werden Sie aufgefordert, eine neue Commit-Nachricht zu verfassen. Üblicherweise werden Sie die vorgeschlagene Nachricht einfach übernehmen wollen.

Paralleles Pushen

Da die Änderungen in den beiden Repositories zwei unterschiedliche Dateien betrafen, konnten sie leicht kombiniert werden.

Um die beiden Commits (neue Datei, sowie den Merge-Commit) auf Github zu bringen, müssen sie noch gepusht werden mit:

```
git push
```

Wenn für das andere Repository noch `git pull` ausgeführt wird, sind beide lokalen Repositories wieder auf demselben Stand wie das Github-Repository.

Merge-Konflikte

Bis jetzt sind wir mit einem blauen Auge davongekommen, da die Änderungen in den beiden lokalen Repositories unterschiedliche Dateien betrafen und Git die Änderungen automatisch kombinieren konnte.

Wenn wir in den beiden Repositories dieselbe Zeile derselben Datei (unterschiedlich) abändern, wird Git die beiden Änderungen nicht automatisch kombinieren können. In diesem Fall spricht man von einem Merge-Konflikt.

Spielen Sie diese Situation ebenfalls durch, indem Sie versuchen für eine bestimmte Zeile derselben Datei unterschiedliche Inhalte zu pushen. Führen Sie (wieder ausnahmsweise) beim zweiten Repository den Befehl `git pull` (zur lokalen Integration der Änderungen von Github) erst *nach* dem Befehl `git push` aus, damit der Merge-Konflikt auftritt.

Eventuell wollen Sie die Befehle für die beiden Repositories vorgängig notieren.

Merge-Konflikte

Sie sollten nun folgende Fehlermeldung erhalten:

```
$ git pull
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 8 (delta 2), reused 8 (delta 2)
Unpacking objects: 100% (8/8), done.
From https://github.com/ltog/sandbox
   2853d17..1d2afb7  master     -> origin/master
Auto-merging blub
CONFLICT (content): Merge conflict in meine_datei
Automatic merge failed; fix conflicts and then commit the result.
```

Merge-Konflikte

Auch mit `git status` ist der Konflikt sichtbar:

```
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   meine_datei

no changes added to commit (use "git add" and/or "git commit -a")
```

Merge-Konflikte

Wenn wir die Datei mit einem Texteditor anschauen, sehen wir folgendes:

```
<<<<<<< HEAD
blub B
=======
blub A
>>>>>>> 1d2afb7fd96200cf2c2abebeaa86247f443e9e8c
```

Erklärung:

Blau: Lokale Variante
Rot: Variante vom Server
Orange: Marker

Merge-Konflikte

Wir haben jetzt 3 Optionen:

1. Unsere Änderungen erzwingen und die Änderungen vom Server verwerfen
2. Die Änderungen vom Server übernehmen und unsere Änderungen verwerfen
3. Den Konflikt manuell beheben
   (Die Datei manuell bearbeiten und so aussehen lassen, wie sie soll. Alle Marker entfernen.)

```
git checkout --ours meine_datei  # Option 1
git checkout --theirs meine_date # Option 2
vim meine_datei                  # Option 3
git add meine_datei
git commit --m "..." # Erklärung wie gemerget wurde
git push
```

Versionen vergleichen

Ein Vorteil der Verwendung eines Versionskontrollsystems ist, dass unterschiedliche Dateistände miteinander verglichen werden können. Zeilen in grün mit einem Plus-Symbol wurden hinzugefügt. Zeilen in rot mit einem Minus-Symbol wurden entfernt.

Probieren Sie die unterschiedlichen Varianten aus! Konsultieren Sie bei Bedarf noch einmal das Diagramm mit den unterschiedlichen Zuständen, welche eine von Git verfolgte Datei haben kann.

Sind nur wenige Zeilen vorhanden ist der Versionsvergleich vielleicht wenig beeindruckend. Bei grösseren Projekten ist er aber unverzichtbar.

Versionen vergleichen

Modifizierte Version gegenüber eingecheckter (unmodifizierter) Version:

```
git diff                # Änderungen in allen Dateien zeigen
git diff meinedatei # Nur Änderungen in dieser Datei zeigen
```

Gestagete Version gegenüber eingecheckter (unmodifizierter) Version:

```
git diff --staged                # Änderungen in allen Dateien
git diff --staged meinedatei # Änderung in dieser Datei
```

Differenz zwischen zwei eingecheckten Versionen:

```
git diff oldhash newhash
git diff oldhash newhash meinedatei
```

(Für *oldhash* und *newhash* muss jeweils der Hash eines Commits angegeben werden. Die Liste von Commits/Hashes ist mit `git log` einsehbar.)

Änderung an lokaler Datei zurücknehmen

Hat man eine Datei lokal abgeändert und will sie auf den letzten versionierten Stand zurücksetzen, verwendet man:

```
git checkout -- meinedatei
```

Die gemachten Änderungen sind dann unwiederbringlich entfernt, da sie nicht mit `git commit` versioniert worden sind.

Probieren Sie auch dieses Kommando aus.

Nicht besprochene Punkte

- Branching (Mehrere Entwicklungszweige verwalten; Eine der Stärken von Git)
- `.gitignore` (Dateien für Git ausblenden, welche nur lokal existieren und nicht mit `git status` angezeigt werden sollen)
- Stashing (offene Arbeit lokal zwischenspeichern ohne zu committen)
- GitHub Forks (Eine eigene Variante von bestehender Software erstellen)
- GitHub Pull Requests (Für ein fremdes Repository Änderungen zur Integration vorschlagen)
- Bedienung von GUI Tools (TortoiseGit, Eclipse, …)

Nicht besprochene Punkte

- Squashing (vor der Veröffentlichung mehrere lokale Commits zu einem einzigen kombinieren)
- Rebasing (Änderungen eines Branches auf den aktuellsten Stand eines andern Branches aufsetzen)
- Interaktives Hinzufügen von Änderungen zum Index
- Cherry picking: «Herauspicken» eines einzelnen Commits
- Alte Version auschecken und detached-head state
- Orphan branches: Leere Branches, die nicht von einem andern Branch abstammen
- Tagging

Ressourcen

Pro Git Buch: http://git-scm.com/book/ (mit deutscher Übersetzung)

Ry's Git Tutorial: http://rypress.com/tutorials/git/index.html

Mein Spickzettel (englisch):
https://github.com/ltog/know-how/blob/master/git.md

Vielen Dank für Ihre Aufmerksamkeit.