

Git-Workshop

Chemnitzer Linuxtage 2010

Valentin Hänel, Julius Plenz

13. März 2010



Veröffentlicht unter der CreativeCommons-Lizenz (By, Nc, Sa)

<http://github.com/Feh/git-workshop/raw/pdf/folien/git.pdf>

Wer kennt wen?

Wer kennt oder hat schon mal eines der folgenden Systeme benutzt?

- ▶ CVS/RCS
- ▶ SVN
- ▶ Mercurial, Darcs, Perforce, Bazaar
- ▶ Git

Wer kennt Git?

Wer hat schonmal ...

- ▶ `git` eingegeben
- ▶ Ein Git-Repository selbst erstellt?
- ▶ ... oder geklont?
- ▶ Einen Commit gemacht?
- ▶ Per Git mit anderen Leuten zusammengearbeitet?

Ablaufplan

- ▶ Zu Anfang führen wir in die einfachsten Git-Kommandos ein und üben einige wichtige Arbeitsschritte.
- ▶ Im zweiten Teil des Workshops wollen wir dann gemeinsam an einem Projekt arbeiten, um beispielhaft den Entwicklungsprozess einer Software zu demonstrieren.

Wer bin ich? – Name und E-Mail einstellen

- ▶ Für alle Projekte (wird in ~/.gitconfig gespeichert)
 - ▶ `git config --global user.name "Max Mustermann"`
 - ▶ `git config --global user.email max@mustermann.de`
- ▶ ... oder alternativ nur für das aktuelle Projekt:
 - ▶ `git config user.email maintainer@cool-project.org`
- ▶ Außerdem, für die, die wollen: Farbe!
 - ▶ `git config --global color.ui auto`

Ein Projekt importieren oder erstellen

- ▶ Ein neues Projekt erstellt man wie folgt:
 - ▶ `mkdir projekt`
 - ▶ `cd projekt`
 - ▶ `git init`
- ▶ Um ein bestehendes Projekt zu importieren, »klont« man es mit seiner gesamten Versionsgeschichte:
 - ▶ `git clone git://git.plenz.com/git-tips`

Begriffsbildung

- ▶ **Index/Staging Area:** Bereich zwischen dem Arbeitsverzeichnis und dem Repository, in die Änderungen für den nächsten **Commit** gesammelt werden
- ▶ **Commit:** Eine Änderung an einer oder mehrerer Dateien, versehen mit Metadaten wie Autor, Datum und Beschreibung
- ▶ **Referenz:** Jeder **Commit** wird durch eine eindeutige SHA1-Summe identifiziert. Eine Referenz »zeigt« auf einen bestimmten Commit
- ▶ **Branch:** Ein »Zweig«, eine Abzweigung im Entwicklungszyklus, z. B. um ein neues Feature einzuführen.

Ein typischer Arbeitsablauf

- ▶ Eine *datei* verändern, und die Änderungen in das Repository »einchecken«:

1. `vim datei`
2. `git status`
3. `git add datei`
4. `git commit -m 'datei angepasst'`
5. `git show`

Referenzen und ignorierte Dateien

Relative Referenzen:

- ▶ HEAD: Der letzte Commit (wird per `git show` angezeigt)
- ▶ HEAD[^]: Der vorletzte Commit
- ▶ HEAD~*N*: Der *N*.-letzte Commit

Dateien ignorieren:

- ▶ Globbing-Muster in `.gitignore` schreiben. Z. B.:
 - ▶ *.aux
 - ▶ *.bak
 - ▶ *.swp
 - ▶ .gitignore (die Datei selbst)

Informationen über das Repository erhalten

- ▶ Den jüngsten Commit im vollen Umfang anschauen:
 - ▶ `git show`
- ▶ Die gesamte Versionsgeschichte, die zum aktuellen Zustand führt, anzeigen:
 - ▶ `git log`
- ▶ Was hat sich verändert?
 - ▶ `git diff`
- ▶ Das Repository visualisieren:
 - ▶ `gitk`
- ▶ ... oder textbasiert:
 - ▶ `tig`

Änderungen rückgängig machen

Einen neuen Commit erstellen, der eine alte Änderung rückgängig macht:

- ▶ `git revert commit`

Den Index zurücksetzen:

- ▶ `git reset HEAD`

Den Zustand von vor zwei Commits wiederherstellen:

- ▶ `git checkout HEAD~2`

Die Version von *datei* anschauen, wie sie vor zwei Commits war:

- ▶ `git show HEAD~2:datei`

Die letzten zwei Commits **unwiederbringlich** löschen:

- ▶ `git reset --hard HEAD~2`

Branches: Abzweigungen

Wir arbeiten schon die ganze Zeit im master-Branch!

Was genau sind Branches? – Nichts anderes als Referenzen auf den jeweils obersten Commit einer Versionsgeschichte.

Branches ...

- ▶ erstellen: `git branch name`
- ▶ auschecken: `git checkout name`
- ▶ erstellen und direkt auschecken: `git checkout -b name`
- ▶ auflisten: `git branch -v`
- ▶ löschen: `git branch -d name`

Idealisierter Workflow: Ein Branch pro neuem Feature oder Bugfix.

Beispiel: Zwei Branches

Zwei Branches erstellen, und auf jedem einen Commit machen.
Dann das Resultat in gitk anschauen.

- ▶ `git branch apfel`
- ▶ `git checkout apfel`
- ▶ Commit machen
- ▶ `git checkout master`
- ▶ `git checkout -b birne`
- ▶ Commit machen
- ▶ `gitk --all`

Merging: Branches Zusammenfügen

Simple Merge:

- ▶ `git merge neues-feature`

Fast-Forward Merge:

- ▶ Wird *topic* in *master* gemerget und *topic* basiert auf *master*, dann wird kein Merge-Commit erstellt, sondern nur der Zeiger »weitergerückt« bzw. »vorgespult«.

Hilfe, Konflikte!

Bei einem merge kann es zu Konflikten kommen. Wie geht man damit um?

- ▶ `vim konfliktdateien`
- ▶ `git add konfliktdateien`
- ▶ `git commit -m "Merge-Konflikt behoben"`

Das Unterfangen abbrechen:

- ▶ `git reset HEAD`

Hinaus in die weite Welt!

- ▶ Wir wollen unsere Arbeit mit der anderer Entwickler austauschen!
- ▶ Durch die verteilte Architektur von `git` braucht es keinen *zentralen* Server zu geben.
- ▶ Das Entwicklerteam muss sich auf einen *Workflow* einigen:
 - ▶ Shared Repository
 - ▶ Maintainer/Blessed Repository
 - ▶ Patch-Queue per E-Mail
 - ▶ ... oder auch alles durcheinandergemixt.

Remote Repositories / Remote Branches

Remote Repositories verwalten:

- ▶ `git remote -v`
- ▶ `git remote add name url`
- ▶ `git remote rm name`
- ▶ `git remote update`
 - ▶ Fragt bei allen Remote Repositories an, ob es neue Commits gibt. (Eigene Commits werden durch dieses Kommando **nicht** veröffentlicht!)

Details der Repositories ändern (z. B. Vertipper):

- ▶ `vim .git/config`

Remote Branches auflisten:

- ▶ `git branch -r`

Fremden Code holen, eigenen versenden

Aus einem anderen Repository neuen Code »ziehen«:

- ▶ `git pull remote branch`
 - ▶ `git pull blessed master`

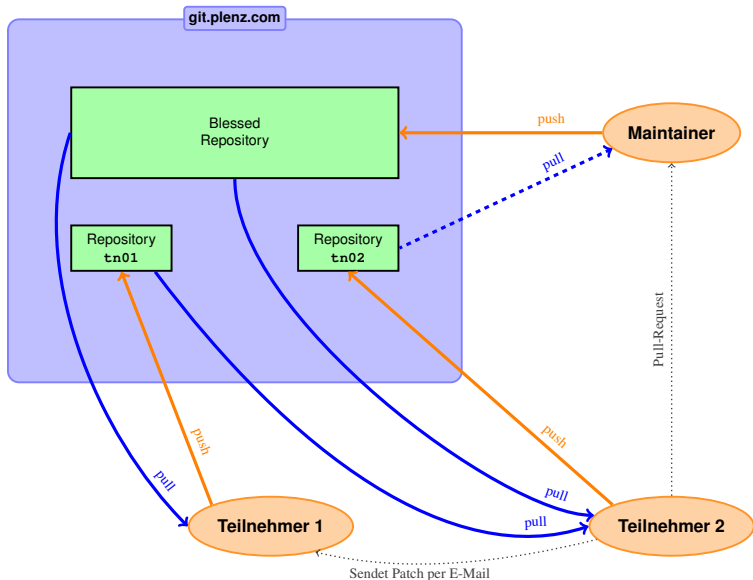
Was hinter den Kulissen passiert:

1. `git fetch remote branch`
2. `git merge remote/branch`

Eigene Commits »pushen« oder per E-Mail senden:

- ▶ `git push remote branch`
- ▶ `git format-patch seit-wann`

Die Infrastruktur



Wir tauschen uns aus

- ▶ Remote *blessed* klonen (read-only):
 - ▶ `git clone -o blessed git://git.plenz.com/git-tips`
 - ▶ `cd git-tips`
- ▶ Schreibbares Repository eintragen: *meins*
 - ▶ `git remote add meus
ssh://tn01@git.plenz.com:2222/repos/tn01`
- ▶ Repository eines anderen Teilnehmers hinzufügen:
 - ▶ `git remote add tn02 git://git.plenz.com/tn02`
- ▶ Schauen, »wie weit« die anderen sind:
 - ▶ `git remote update`
 - ▶ `gitk --all`

Kür: Was noch fehlt

- ▶ Rebase
- ▶ `git stash`
- ▶ Remote Branches löschen
- ▶ Git-Aliase
- ▶ Tags
- ▶ Reflog

Danke!

Vielen Dank für eure Teilnahme!

Fragen und Feedback gerne per Mail:

Valentin Haenel
valentin.haenel@gmx.de

Julius Plenz
clt10@plenz.com

Rebasing

- ▶ **Rebase:** Einen Branch auf eine »neue Basis« stellen:
 - ▶ `git rebase master topic`
- ▶ Interaktiv Commits neu ordnen, bearbeiten, zusammenfassen oder aufteilen:
 - ▶ `git rebase -i HEAD~5`
- ▶ **Wichtig:** Man darf *niemals* Commits aus einem bereits veröffentlichten Branch – auf dem also womöglich Andere ihre Arbeit basieren – durch `git rebase` verändern!
 - ▶ **Daher: Nur Unveröffentlichtes gegen Veröffentlichtes rebase:**
 - ▶ `git rebase origin/master`
 - ▶ `git rebase v1.1.23`