Git-Schulung

–Firma–

Valentin Hänel, Julius Plenz

–Datum–

Übersicht

Session 4

- Weitere nützliche Kommandos
- Forgeschrittene Nutzung
- Submodules
- Interaktion mit SVN

Übersicht

Dateien ignorieren

- `.gitignore` regelt, welche Dateien Git ignorieren soll
- Enthält Zeilen mit Globbing-Ausdrücken, z. B.
  - `*.[oa]` für C-Objekt-Dateien
  - `*.tmp` für temporäre Dateien

- Regeln werden kaskadierend angewendet
  1. Definitionen im aktuellen Verzeichnis
  2. Aus höheren Verzeichnissen im aktuellen Projekt

Dateien, die in allen Repositories ignoriert werden sollen

```
git config --global core.excludesfile ~/.gitignore
```

`.gitignore` einchecken

- `.gitignore` in einem Repository erstellen und einchecken
  - Enthält für das Projekt typische »Abfalldateien«
- Alle Mitentwickler haben nun die gleichen Ignore-Einstellungen
  - In `.git/info/excludes` kann die Liste erweitert werden
  - Ein `!` negiert den Ausdruck: `!*.pdf` hebt eine Ignore-Anweisung auf PDF-Dateien auf

Aufräumen – `git clean`

Alle nicht von Git verwalteten Dateien und Verzeichnisse entfernen

```
git clean -fd
```

Alle ignorierten Dateien entfernen

```
git clean -fdX
```

Ignorierte und nicht von Git verwaltete Dateien löschen

```
git clean -fdx
```

- Funktioniert gut bei einer gepflegten `.gitignore`
- Kann in den Build-Prozess eingebaut werden

Achtung! Dateien werden unwiederbringlich gelöscht!
Wenn unsicher, `-n` verwenden (zeigt an, was gelöscht werden würde).

`git blame` – Wer hat diese Änderung ausgeführt?

- Sucht für jede Zeile einer Datei den Autor der letzten Veränderung
- $\rightarrow$ Bei Unklarheiten direkt den Autor fragen
- Gibt auch Zeit und Commit aus

Autoren aller Zeilen in *datei*

```
git blame datei
```

Autoren der Zeilen 10–20

```
git blame -L 10,20 datei
```

Nach Inhalten suchen

- Analog zum UNIX-Befehl `grep` gibt es `git grep`
- Sucht nur in Dateien, die mit Git verwaltet werden
- Optionen an `grep` angepasst
- Ist multithreaded und daher schneller als `grep`!

In allen Dateien nach `foobar` suchen

```
git grep foobar
```

Nur in *`feature`* suchen

```
git grep foobar feature
```

Übersicht

Git: Eigene Kommandos

- Git überprüft bei unbekannten Sub-Kommandos, ob sie existieren
  - `git foobar` → Git sucht nach `git-foobar`
- Einfach, Git zu erweitern
  - Alle Parameter werden unverändert weitergegeben
  - Exit-Code signalisiert den Erfolgs-Status

Git-Kommandos: Beispiel

~/bin/git-release

```
#!/bin/bash

[ $# -ne 2 ] && {
    echo "Usage: git release <name> <tag>" >&2
    exit 1;
}

ar="./${1}-${2#v}.tar.gz"
git archive --format=tar "$2" | gzip > "$ar"
scp "$ar" user@host:/var/www/...
```

- `git release foobar v0.4.2-rc2`
  - Stand von `v0.4.2-rc2` wird als `foobar-0.4.2-rc2.tar.gz` gespeichert
  - Das Archiv wird anschließend hochgeladen

Daten wiederherstellen

- Was tun, wenn nach einem `reset` oder `branch -D` Commits verloren scheinen?
- Git protokolliert alle Vorgänge, die Daten speichern, im *Reflog*

Anzeigen des Reflog

```
git reflog
```

Normales `log` mit Informationen aus dem Reflog

```
git log -g
```

Daten aus dem Reflog übernehmen

- Identifizieren Sie, welche(n) Commit(s) Sie brauchen
- Nutzen Sie Git Kommandos `merge`, `cherry-pick` oder `branch` zum Wiederherstellen
  - Auch wenn die Commits nicht mehr sichtbar sind, sind sie noch vorhanden

Fehlerhaften `reset` rückgängig machen

```
git reset --hard HEAD^
git merge HEAD@{1}
```

Wie lange hält der Reflog?

- Reflog-Einträge werden nach einer bestimmten Zeit gelöscht
  - 90 Tage für alle erreichbaren Commits
  - 30 Tage für alle unerreichbaren

Bedenken Sie! Dateien oder Veränderungen die (noch) nicht mit Git gespeichert sind, können mit diesem Mechanismus nicht wiederhergestellt werden

Commits »sezieren«: `git bisect`

- In der aktuellen Version tritt ein Fehler auf
- In früheren Versionen gab es diesen Fehler nicht

- Welcher Commit hat diese Änderung eingeführt?

- `git bisect` sucht den schuldigen Commit
  - Benötigt ca. $\log_2(n)$ Schritte
  - Automatisierung möglich

Benutzung von `git bisect`

Fehlersuche starten

```
git bisect start
git bisect bad bad-commit
git bisect good good-commit
```

- »Divide and Conquer«: Git präsentiert einen Commit, der in etwa in der Mitte zwischen `good` und `bad` liegt
- Überprüfen, ob der Bug existiert
  - Wenn ja, `git bisect bad`
  - Wenn nein, `git bisect good`

Fehlersuche beenden

```
git bisect reset
```

Bisect automatisieren

- Skript schreiben, das den Test automatisiert
  - Gibt `0` zurück, wenn Programm keine Fehler hat
  - Sonst: Rückgabewert 1–127
- `git bisect` führt für jeden Schritt das Skript aus
  - Rückgabewert `0` $\rightarrow$ markiert Commit als `good`
  - Sonst ist der Commit `bad`

Bisect automatisieren

```
git bisect start bad-commit good-commit
git bisect run test-script.sh
```

Aufgaben automatisieren

- Problem: keine Umlaute in der ersten Zeile der Commit-Nachricht erwünscht
- Lösung: entsprechenden *Git-Hook* einrichten

```
#!/bin/sh

if head -n 1 "$1" | grep -q '[äöüßÄÖÜ]' ; then
    echo >&2 'Umlaute in der ersten' \          'Zeile der C
    exit 2
fi
```

Git-Hooks

- Hooks (inkl. Beispielen) befinden sich in `.git/hooks`
- Können in beliebiger Sprache entwickelt werden
- Werden nicht in das Repository eingechekt

- Durch Umbennenung werden Beispiel-Hooks aktiviert
  - z. B. `commit-msg.sample` → `commit-msg`
- Hooks müssen ausführbar sein (`chmod +x`)

Mögliche Hooks

- `applypatch-msg`
- `commit-msg`
- `post-commit`
- `post-receive`
- `post-update`
- `pre-applypatch`
- `pre-commit`
- `pre-rebase`
- `prepare-commit-msg`
- `update`

Übersicht

Externe Abhängigkeiten einbinden

- Zusätzliche Repositories bindet Git durch *Submodules* ein
- So verwalten Sie Abhängigkeiten (Plugins, Bibliotheken etc..)
- Das enthaltende Repository wird *Superproject* genannt

Submodule hinzufügen

```
git submodule add url name
git submodule init
git commit -m "Submodule name hinzugfügt"
```

Wie wird die Information gespeichert?

- Informationen über Submodules werden in der Datei `.gitmodules` hinterlegt
- Diese wird als versionierte Datei im Repository gespeichert

Beispieleintrag in `.gitmodules`

```
[submodule "name"]
    path = pfad
    url = url
```

- Ausserdem speichert Git die SHA1 vom `HEAD` vom Submodule
- $\rightarrow$ und zwar in einem *Gitlink*
  - Dieser heißt so wie der Pfad des Submodule
  - z. B. Pfad: `submodule-one` – Dateiname: `submodule-one`
- Der Gitlink wird im Repository gespeichert
- Dadurch kann der genaue Zustand wiederhergestellt werden

Submodules initialisieren

- Wird ein Projekt geklont, sind die Submodules leer

Registrieren der Submodules in `.git/config`

```
git submodule init
```

Submodules klonen, und `HEAD` auf den richtigen Commit setzen

```
git submodule update
```

In einem Schritt

```
git submodule update --init
```

- Nachteil: die lokale URL kann nicht angepasst werden

Submodules anzeigen

Status aller Submodules

```
git submodules
```

- Zeigt SHA1 und den Name an
  - (–) bedeutet: kein Eintrag in `.git/config`
  - (+) bedeutet: SHA1 im Submodule ist anders als im Gitlink

Rekursiv alle Submodules anzeigen

```
git submodules --recursive
```

Submodules verändern

- Wir unterscheiden zwischen eigenen und fremden Veränderungen

- Für eigene Veränderungen können Sie die Submodule (fast) wie ein normales Git-Repository behandeln
- Problem: das Submodule hat einen Detached-`HEAD`
- Den geeigneten Zustand via Gitlink in dem Superproject speichern

- Bei fremden Veränderungen wird der Gitlink für das Submodule aktualisiert
- Es reicht ein: `git submodule update`
- Mögliches Problem: der Commit im Submodule ist nicht öffentlich verfügbar
  - Lösung: Kontakt zum verantwortlichen Entwickler

Submodules entfernen

- Leider existiert noch kein `git submodule remove` Kommando

Submodule entfernen

`rm -rf` *`submodule`*
–> entfernt das Verzeichnis
`git rm` *`submodule`*
–> entfernt den Gitlink
`vim .gitmodules`
–> Entsprechenden Eintrag manuell löschen
`git add .gitmodules`
`git commit`
–> Änderungen Comitten
`vim .git/config`
–> Submodule Eintrag in der `.git/config` löschen

Übersicht

SVN und Git

- Git kann direkt mit SVN-Repositories umgehen
  - Import der SVN-Einträge
  - Export von Git-Commits
- Konzepte von Git und SVN sind allerdings unterschiedlich
  - SVN bevorzugt eine komplett lineare Entwicklungsgeschichte
  - Git ermuntert zur Entwicklung in vielen Branches
- Importierte SVN-Geschichte ist linear
- Zu exportierende Git-Commits müssen linearisiert werden

`git svn` – Der SVN-Layer

Ein SVN-Repository importieren

```
git svn clone svn://hostname/svn/trunk
```

Auf neue SVN-Änderungen überprüfen

```
git svn fetch
```

Einen Branch *feature* nach SVN hochladen

```
git checkout feature
git svn rebase  # "fetch" passiert automatisch
git svn dcommit
```

Vorteile von `git svn`

- Flexibles Branching von Git
  - Erlaubt aufgabenorientierte Entwicklung
- Ein Branch kann lange Zeit entwickelt werden
  - Änderungen müssen *nicht* direkt ins SVN-Repo einfließen
  - Branch basiert jeweils auf dem neusten SVN-Status
  - Patch der Änderungen in einem Branch können vorerst per Patch verteilt werden
- Internet-Verbindung nur notwendig bei `git svn dcommit`!
  - `git svn log` etc. funktionieren auch offline (und sind schneller)