Aufgaben

Dieses Dokument enthält die Aufgaben zur Git-Schulung, die von den Teilnehmern hintereinander auszuführen sind.

Glossar

Es existiert ein separates Glossar rund um das Thema git, welches, wenn es einmal fertig ist, als “Schulungsglossar” abgelegt ist.

Initiales Repo anlegen

Ein leeres Repo wird an beliebiger Stelle mit
git init
angelegt. Damit werden die Datenstrukturen für das Repo angelegt, genauer der Ordner .git im Initialzustand. Das Verzeichnis, in dem “git init” ausgeführt wurde, ist ab sofort ein Working Directory. Es empfiehlt sich also, für das initiale Repo zunächst ein Verzeichnis anzulegen.

- Leere Repos? -> eher nicht, in Praxis eher fertige Strukturen bzw. Templates

ZUBEHÖR: Repository git-tutorial.zip

Basis

git init -> einmalig beim Aufsetzen des Repos, schreibt .git Metadaten
git add -> “Staging” für jedes Objekt **und** jede Version, welches gesichert werden soll (“Es werden Changes verwaltet, keine Versionen” @RP: theoretischer Ausflug, wenn gewünscht)
git commit -> festschreiben aller im Staging erfassten Änderungen
git commit -m “text” -> commit message mitgeben (nicht immer erwünscht)

-> Wie komme ich da wieder ran? im nächsten Abschnitt!

Alte Versionen

Was ist früher passiert?
git log -1 -> letzter commit
git log -10 -> letzte 10 commits
git log

Zugriff auf “frühere” Versionsstände: (später: einfach “andere” Versionsstände)
git checkout -- -> (Änderungen verwerfen)

git checkout HEAD -> Stand vom HEAD
git checkout HEAD^1
git checkout HEAD^2
git checkout <SHA1> -> beliebiger Stand, analog SVN Revision

-> Vergleichen auf diese Weise offenbar mühselig. Schneller mit:
git diff: WA vs. index, index vs. HEAD, HEAD vs. HEAD^1
git diff: datei a vs. datei b

git tag <NAME> -> statt SHA1 (später mehr)

gitk -> ENDLICH! SEHEN! (--all wird bei den branches gezeigt)

Author-Informationen konfigurieren

git config: E-Mail konfigurieren, zusammen mit gpg einen PGP-Schlüssel konfigurieren (vielleicht später?)
git config --global user.name “Klaus Mustermann”
git config --global user.email “klaus.mustermann@elektronische-post.de”
git config --global user.signingkey <GPG-Key> -> möglicherweise lokal überschreiben

-> wer ein funktionierendes GnuPG hat, kann gleich ausprobieren:
git tag -s -> signierte tags
-> Hinweis: Es gibt noch weitere Tag-Typen, bei Bedarf git tag --help

Branches anlegen

NEWSFLASH! WIR HABEN SCHON EINEN BRANCH ANGELEGT!

git branch -> zeigt alle branches, aktiver mit *
git branch xyz -> branch anlegen
git checkout, commits, hin- und herwechseln.

Branch switches: Keine Kollision

- Datei a/abc inhalt 1 anlegen, einchecken
- Datei a/abc inhalt andern, nicht einchecken
- branch wechseln git branch b, b/abc inhalt einchecken

Branch switches: Kollision

- Datei a/xyz inhalt 1 anlegen, einchecken
- Datei b/xyz inhalt 2 anlegen, einchecken
- git branch a, a/xyz inhalt verändern, branch wechseln

-> Kollision erstmal ignorieren (lokale Änderung verwerfen)

Branches vergleichen

git diff -> “universelles Konzept” (alles läßt sich diffen)

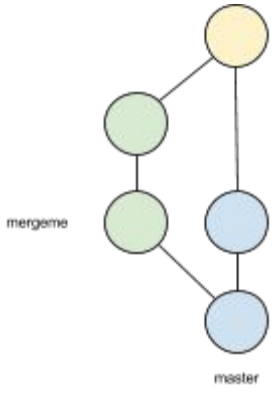
Das Clonen

- git clone /path/to/git-tutorial
->

Einfaches Mergen

Branches: master
Repo: Branch mergeme

- git checkout master
- git diff mergeme
- gitk --all
- git merge mergeme



Remote

Einfaches Fetchen

Vorbereitung: 2er Teams.
- git clone /path/to/git-tutorial.git fetch-repo
- cd fetch-repo
- git remote add -m master -t master other /Other/player/fetch-repo

2er teams: Repos gegenseitig eintragen. Player A fetcht von Player B und zurück. Basis ist ein Clone des Tutorial-Repos.

- git fetch other: warum sieht man die änderungen nach dem fetch nicht -> gitk, dann merge!
- “ist das nicht lästig?” -> was macht pull? git fetch + git merge

Einfaches Pushen

Vorbereitung: 2er Teams.
- git clone /path/to/git-tutorial.git push-repo
- cd push-repo
- git remote add -m master -t master other /Other/player/push-repo

- git push other

Hinweise:
    - Die Repos sollten trotzdem strukturell intakt sein.
    - Git kann alles wiederherstellen, was committed wurde.
    - NIE in repos mit WC pushen, es sei denn, das ziel ist ein chaos.

remotes anlegen: git remote
git remote

Bare Repos

Vorbereitung: alle zusammen.
- git clone /path/to/git-tutorial.git
- cd git-tutorial
- do something
- git push

clonen vom bare-repo. gleiche Konstellation wie zuvor, nur wird nicht direkt gepusht, sondern über ein --bare repo. erste einfache Konflikte.
-> Üben: Verändern vom modifizierten Dateien (working copy, noch nicht eingecheckt), über fetch, merge und pull.

Tags und Remote Repos

- git tags, git push
- tags werden beim push per default nicht publiziert! (Ausnahme: --tags option bei remote)

Revert

Vorbereitung: in git-tutorial, alle zusammen.
- Änderung am master vornehmen
- git commit
- git revert HEAD^1
- git log / gitk -> baut einen “inversen patch” (neutralisiert den Effekt der gewählten commits)

Mergen

Grundsätzliches

- Flag --no-ff beim Integrieren von Features (siehe “revert”)
- Strategien mit Einfluss auf Merges: Early und Late (Projektentscheidung)
    -> Early: viele kleine Merges (integrates development)
        + Änderungen leicht zu überschauen und zu integrieren
        + Änderungen sind schnell an alle Entwickler ausgerollt (aber nicht reif)
        - keine vollständige Umsetzung von features
        - keine saubere Trennung der features
    -> Late: ein großer Merge (isolated development)
        + Features sauber getrennt
        + Features vollständig umgesetzt
        - Änderungen nicht so leicht überschaubar
        - Änderungen werden spät an alle Entwickler ausgerollt (aber reif)

Konflikte

- git checkout master
- git merge conflict

- Konflikte auflösen: weitere Anforderungen, dann zwei Branches zusammenführen
-> Hilfe mit git gui
-> Hilfe mit vi
-> Hilfe mit IDEA (oder eine andere IDE)
-> Vorschläge?

Konflikte bei Binärdateien

- git checkout master
- git merge binary -> Konflikt beheben

Konflikt mit Icons provizieren: Player A bekommt Icon Set 1, Player B bekommt Icon Set 2. Mergen, Konflikte auflösen, Limits: keine Binärdateien, keine Verwaltung der Resourcen (unbenutzte Icons werden nicht getrackt)

Rebase und History Rewrite

git rebase - wann wird es gebraucht? (inklusive -i)
git reset - erklären (Was passiert, wenn ein HEAD verschoben wird?)
git rerere - erklären
git reflog

Content Merge

(viel später)
- CSS refactoring, im feature branch
- Anpassung im master
- im integrationsbranch zusammenführen
- auf dem master ausrollen
-> hat's funktioniert? wenn ja: toll! wenn nein: auch gut, entwickler sollen auch noch selber denken. ;)

Anderes

git archive -> Archive mit eingebautem SHA1 Kommentaren zum Wiederfinden der Version
git submodule -> geht (analog svn externals), aber braucht man das wirklich?

Speicherplatz: Wieviel Platz braucht das gesamte Projekt, viel davon .git? Weitere Beispiele:
- Linux Kernel (RP: .git ca. 500mb, ausgecheckter Code 500mb)
- X86

Anwendung

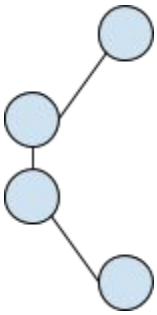
Best Practice

Branching Modell ("Workflow")

- nicht wirklich ein Workflow, daher passt Modell besser
- "Standard-Modell": "A Successful Git Branching Model"
    - "Erfolgreich": von nvie.com erfolgreich eingesetzt (Don't Trust The Internet)
    - Grundprinzipien: Feature Branches, "Trivial Merges" aka Fast Forward (ausser...)
- Dank Git und seinen leichtgewichtigen Branches können Branches häufig eingesetzt werden.

- Ziel: Branches effektiv einsetzen (maximaler Nutzen)
- Branchkategorien:
    - master: Stabile, ausrollbare Releases
    - develop: Nicht stabil für nightly builds
    - Feature Branches (*)
        - Neue Funktionen/Anpassungen
        - aus develop
        - nach develop
    - Release Branches (release-<versionsnummer>)
        - Änderungen, die fürs neue Release notwendig sind. (Versionsnummer...)
        - aus develop
        - nach master
        - und develop
        - und master-Version taggen
    - Bugfix (bugfix-<Identifizierer>)
        - “Großer” Bug muss schnell beseitigt werden
        - aus master
        - nach master
        - und develop
        - und master-Version taggen
- Git Flow ist eine Git-Erweiterung, die die Kommandos etwas zusammenfasst
    - git flow init - initialisiert das Repository für Git Flow (Standardeinstellungen empfohlen)
    - git flow help oder git flow <kommando> help

http://nvie.com/posts/a-successful-git-branching-model/



Real World Beispiele

Konfigurationsmanagement

- kein Modell zur Softwareentwicklung, sondern zur Konfigurationsverwaltung
master: “plain”-Konfiguration

dev: Konfiguration Entwicklungsmaschine
test: Konfiguration Testmaschine
kunde: Konfiguration Kundenmaschine
- änderung auf master
- anpassung auf speziellen Systemen
- mergen bei änderungen
-> warum funktioniert das? Weil git Änderungen trackt, nicht Versionen