Neues Repository anlegen

Im Arbeitsverzeichnis des Projekts:
`git init`
Hierbei wird das Repository im *Unterverzeichnis* **`.git`** angelegt. Das Repository besteht aus *vielen Dateien* (mindestens 15).

`fossil new` ***`repository-name`***
Hierbei wird die *Datei* mit dem ***`repository-name`*** angelegt (oft mit Namenserweiterung **`.fossil)`**, wobei auch ein absoluter oder relativer Pfad verwendet werden kann. Das Repository ist *eine einzige Datei.* Sinnvollerweise legt man a*lle Repositorys gemeinsam* in ein Verzeichnis.

Repository öffnen und schließen

Da sich bei Git das Repository unterhalb des Arbeitsverzeichnisses befindet, sind sie fest miteinander verknüpft, so dass es kein Öffnen und Schließen gibt.

Ein Repository kann daher nicht mehrfach geöffnet werden.

Im Arbeitsverzeichnis: **`fossil open`** ***`repository-name`***

Hierbei wird im Arbeitsverzeichnis eine kleine Datei **`.fslckout`** (bei allen Systemen außer Windows) bzw. **`_FOSSIL_`** (Windows) angelegt, welche die Verknüpfung zum Repository enthält.

Im Arbeitsverzeichnis: **`fossil close [ -f ]`**
Wenn nicht commitete Änderungen im Arbeitsverzeichnis vorhanden sind, wird die Option **`-f`** benötigt.

Ein Repository kann in verschiedenen Arbeitsverzeichnissen gleichzeitig geöffnet sein – mit derselben Version bzw. Branch oder auch verschiedenen.

Repository klonen

Im Arbeitsverzeichnis: **`git clone`** ***`repopfad`***
Der ***`repopfad`*** kann auch ein URL sein, z. B. per SSH: **`ssh://benutzer@host:repopfad`**

`fossil clone` ***`repopfad repository-name`***
Der **`repopfad`** kann auch ein URL sein, z. B. per SSH: **`ssh://benutzer@host/repopfad`**
Das Repository muss anschließend in einem Arbeitsverzeichnis geöffnet werden (s. o.).

Dateien hinzufügen

Wenn Dateien im Arbeitsverzeichnis erstellt werden, unterstehen diese noch nicht zwingend der Versionierung. Sie müssen hinzugefügt werden.

`git add dateiname`
Dies muss bei späteren Änderungen an der Datei jedes Mal wiederholt werden, damit die Datei „gestaget“ wird, d. h. „zum Index hinzugefügt“.

`git add .`
fügt *alle* Dateien und Unterverzeichnisse hinzu.

`fossil add` ***`dateiname`***
`fossil add .` fügt *alle* Dateien und Unterverzeichnisse hinzu.

Fossil verfolgt Änderungen an einmal hinzugefügten Dateien automatisch.

Änderungen committen

Unter „committen“ versteht man das Festhalten eines Zustands als Version. Hierdurch werden die Änderungen ins lokale Repository geschrieben, aber (noch) nicht in das, welches man vorher geklont hat – außer bei Fossil, falls Autosync eingeschaltet ist (s. u.).

`git commit [ -a ] [ -m` ***`kommentar`*** **`]`**
Gibt man keine Option **`-m`** mit Kommentar an, öffnet sich ein Editor, in den man einen Kommentar eingibt. Sind

`fossil commit [ -m` ***`kommentar`*** **`]`**
Gibt man keine Option **`-m`** mit Kommentar an, öffnet sich ein Editor, in den

Änderungen nicht vorher mittels **`git add`** hinzugefügt („gestaget“) worden, erscheint eine Warnung. Die Option **`-a`** erlaubt das Stagen und Committen in einem Schritt.

`git reset`
holt alle Dateien aus dem Stage zurück, ändert nichts am Arbeitsverzeichnis.

man einen Kommentar eingibt. Möchte man ausnahmsweise nicht die Änderungen an allen Dateien committen, so gibt man die Namen der Dateien an, welche committet werden sollen.

Änderungen ins Remote-Repo übernehmen

Im jetzigen Zustand gibt es keine Branches, sondern nur den Trunk, d. h. den Stamm der Entwicklung, von dem ggf. verzweigt wird, was dann sogenannte Branches erzeugt.

`git push origin master`
Hat man sein Repo von einem anderen geklont, so existiert der Name „**`origin`**“ automatisch, so dass git weiß, in welches (remote) Repo geschrieben werden soll. „**`master`**“ bezeichnet den Trunk. Diese beiden Angaben sind ohnehin default, so dass **`git push`** genügt.

Es wird immer *nur der genannte* Branch gepusht.

`fossil push [ URL ] [ --once ]`
Hat man sein Repo von einem anderen geklont, so ist die Angabe des ***`URL`*** wahlfrei, denn Fossil erinnert sich an den zuletzt verwendeten. Mit der Option **`--once`** wird der ***`URL`*** nicht als Standard gespeichert. Ohne diese Option wird von nun an vor jedem **`commit`** eine Synchronisation zwischen dem eigenen und dem Remote-Repo durchgeführt, sofern eine Verbindung hergestellt werden kann. Es werden immer *sämtliche* Inhalte (außer privaten Branches) übertragen.

Mit **`fossil settings autosync false`** kann man die automatische Synchronisation abschalten.

Status prüfen

Der Status gibt an, welcher Branch aktuell verwendet wird und welche Dateien gegenüber der letzten committeten Version (also gegenüber dem Repo) verändert worden sind.

`git status`
zeigt zusätzlich auch neue Dateien an, d. h. solche, die von Git nicht versioniert werden.

`fossil status`
`fossil extras` zeigt neue Dateien an, d. h. solche, die von Fossil nicht versioniert werden. *Tipp:* ein Mini-Script mit diesen beiden Kommandos erstellen.

Unterschiede zeigen

Oft möchte man sehen, was sich seit dem letzten Commit geändert hat. Das kann man visualisieren.

`git diff` zeigt nicht gestagete Unterschiede an.
`git diff --staged`
zeigt gestagete Unterschiede an.

Es können zwei Versionen angegeben werden, so dass der Unterschied zwischen diesen angezeigt wird.

`gitk` zeigt es ggf. schöner.

`fossil diff` zeigt Unterschiede in Textform an.
`fossil gdiff` zeigt sie mithilfe eines externen Programms an. Unter Windows ist **`windiff.exe`** voreingestellt, in der Einstellung **`gdiff-command`** kann das geändert werden (Empfehlung: **`meld`**)
`fossil gdiff -tk` verwendet zur Anzeige das Tk aus Tcl/Tk, welches installiert sein muss.

Durch Angabe von Dateinamen wird die Anzeige der Unterschiede auf diese Dateien beschränkt.

Durch Angabe von **`-r version`** wird der Unterschied zwischen der genannten Version und dem Arbeitsverzeichnis gezeigt.

Durch Angabe von **`--from version`** und **`--to version`** werden die Unterschiede zwischen den genannten Versionen gezeigt.

Durch Angabe von **`--checkin`** ***`version`*** werden die Änderungen dieses Checkins gegenüber seinem Vorgänger gezeigt.

History

In der History kann man alle bisher gespeicherten Versionen sehen. Alle Versionen kann man auschecken, um den damaligen Zustand wieder herzustellen.

`git log [ -n` ***`anzahl`*** **`]`** zeigt maximal ***`anzahl`*** History-Einträge an.

`fossil timeline [ -n` ***`anzahl`*** **`]`** zeigt maximal ***`anzahl`*** History-Einträge an (default 20).

Es gibt viele weitere Optionen, um bestimmte Einträge auszuwählen. Alle Versionen haben eine Versions-ID (lange Hex-Zahl), die sie eindeutig identifiziert und bei **`checkout`** verwendet werden kann.

Dateien löschen

Üblicherweise beschwert sich die Versionsverwaltung, wenn eine Dateien einfach so fehlt, d. h. gelöscht wurde, ohne sie aus der Versionierung zu nehmen.

`git rm` ***`datei`*** löscht die Datei aus dem Dateisystem. Git wird sich nicht beschweren, dass die Datei fehlt.

Hat man die Datei auf Systemebene gelöscht, gibt man beim nächsten **`commit`** die Option **`-a`** an, so dass die Löschung in der Version vermerkt wird. Git lediglich die Löschung einer bereits entfernten Datei mitzuteilen, ist nicht vorgesehen.

`fossil rm` ***`datei`*** **`[ --hard ]`** teilt Fossil die Löschung mit. Mit der Option **`--hard`** löscht Fossil die Datei auch aus dem Dateisystem. Die Einstellung **`mv-rm-files`** legt fest, ob das auch ohne die Option passiert (per default nicht). Hat man die Datei auf Systemebene gelöscht, kann man vor dem nächsten **`commit`** das Kommando **`fossil addremove`** ausführen, das sämtliche neuen Dateien in Fossil aufnimmt und alle nicht mehr vorhandenen Dateien aus der Versionierung nimmt.

Gelegentlich hat man auf Systemebene eine Datei gelöscht, die man doch zurück haben möchte. Also muss man sie aus dem Repo wieder herstellen. Dies verwirft auch Änderungen an Dateien, die nicht gelöscht wurden, daher Vorsicht!

`git checkout --` ***`datei`***
Dieser Vorgang ist irreversibel.

`fossil revert` ***`datei`***
Dieser Vorgang kann mit **`fossil undo`** zurückgenommen werden.

Dateien umbenennen

Damit die Versionsverwaltungen nicht die alte Datei als gelöscht betrachten und eine neue Datei erstellen (wobei die Historie verloren ginge), sondern die Umbenennung bzw. Verschiebung mitbekommen und eine über diesen Vorgang hinaus gehende History eintragen, muss man entsprechend vorgehen. Beim Verschieben darf der neue Name jeweils auch ein Verzeichnis sein, so dass die Datei unter Beibehaltung ihres Namens in das Verzeichnis verschoben wird.

`git mv` ***`alterName neuerName`***
benennt die Datei um und teilt Git dies mit. Es sollte vor Veränderungen des Dateiinhalts ein **`commit`** durchgeführt werden.

`fossil mv` ***`alterName neuerName`*** **`[ --hard ]`**
teilt Fossil die Umbenennung mit. Mit der Option **`--hard`** wird die Datei nicht nur im Repo, sondern auch im Arbeitsverzeichnis umbenannt bzw. verschoben. Die Einstellung **`mv-rm-files`** legt fest, ob das auch ohne die Option passiert (per default nicht).

Branching

Man kann alles im Trunk (master) entwickeln, aber in Teams ist das keine so gute Idee. Besser ist es, für die Entwicklung jeweils einen Branch (Zweig) anzulegen und in diesem zu entwickeln. Dieser kann regelmäßig (häufig!) in den Trunk zurückgeschrieben werden (Merge). Wenn ein Branch endgültig fertig ist, kann man diesen auch schließen.

`git checkout -b` ***`zweigname`***
erstellt den Branch ***`zweigname`*** und wechselt zu diesem. Evtl. vorhandene Änderungen im Arbeitsverzeichnis bleiben erhalten und können danach committet werden. Mittels **`git checkout master`** und **`git checkout`** ***`zweigname`*** wechselt man zwischen Branch und Trunk hin und her.

`git branch` listet Branches auf mit einem **`*`** am aktuellen Branch.

`fossil commit -b` ***`zweigname`*** committet die aktuellen Änderungen statt in den aktuellen Branch (bzw. Trunk) in den neuen Branch ***`zweigname`*** und verwendet diesen ab jetzt. Die Angabe von weiteren Optionen, insbesondere **`-m`** ist natürlich weiterhin möglich und sinnvoll. Mittels **`fossil checkout trunk`** und **`fossil checkout`** ***`zweigname`*** wechselt man zwischen Branch und Trunk hin und her.

`fossil branch ls` listet Branches auf mit einem **`*`** am aktuellen Branch.

Beim Wechseln zwischen Branches dürfen keine nicht committeten Änderungen vorliegen, denn diese gingen beim Wechsel verloren.

Neben den Namen von Branches können auch Versions-Ids verwendet werden, um die zugehörige Version zu identifizieren. Bei der Angabe von Branchnamen wird automatisch die aktuellste Version dieses Branches genommen.

Merging

Das Übernehmen von allen Änderungen aus einem Branch in einen anderen (insbesondere den Trunk) nennt man mergen. Man wechselt zunächst in den Zielbranch.

`git merge` ***`version`*** **`fossil merge`** ***`version`***

führt die Änderungen der genannten Version im aktuellen Branch durch. Hierbei können Konflikte auftreten, die anschließend manuell gelöst werden müssen. Gerät man hier in Probleme, kann man den Merge-Vorgang widerrufen:

`git merge --abort` (nur aufrufbar bei aufgetretenen Konflikten)

Falls es uncommittete Änderungen gab, kann es passieren, dass der alte Zustand nicht wieder hergestellt werden kann. Daher sollte immer vorher committet werden.

`fossil undo`
Das Kommando **`undo`** wird bei fast allen Operationen angeboten.

Nachdem Konflikte aufgelöst wurden, muss der aufgelöste Zustand erneut committet werden.

Abgleichen mit remote Repository

Änderungen im remote Repository ziehen (meist das, aus dem geklont wurde):

`git fetch` kopiert *aktuellen Branch* von remote ins lokale Repo.

`git pull` macht **`fetch`** und merget aktuellen Branch aus dem Repo ins Arbeitsverzeichnis.

`fossil pull` kopiert *alle* Änderungen von remote ins lokale Repo.
`fossil update` aktualisiert Arbeitsverzeichnis, macht ggf. Autosync vorher.

Änderungen aus aktuellem Repository in das remote Repository schieben (meist das, aus dem geklont wurde):

`git push` überträgt committete Änderungen *am aktuellen Branch* ins remote Repo.

`fossil push` überträgt *alle* committeten Änderungen aus dem lokalen Repo ins remote Repo.

Fossil-Besonderheiten: Um eine komplette Synchronisation durchzuführen, gibt es **`fossil sync`**, welches im Autocommit-Modus vor und nach jedem **`commit`** gemacht wird, um das eigene und das remote Repo synchron zu halten. Auf diese Weise bekommen alle Teammitglieder die Änderungen sehr schnell mit, weil nicht auf ein manuelles **`push`** gewartet werden muss. Alle Teammitglieder sehen alle Branches aller anderen, sofern diese nicht „privat“ sind.

Konfigurieren

Mit **`git config -l`** kann man sich alle Einstellungen anzeigen lassen. **`git config`** zeigt eine Hilfe zur Änderung von Einstellungen.

```
git config --global user.name "Franz Opp"
git config --global user.email "fr@opp.de"
```

stellen Benutzername und E-Mail ein.

`git config --global core.editor emacs` stellt den Editor auf Emacs.

`git config --global merge.tool meld` stellt das Merge-Tool auf Meld.

Mit **`fossil settings`** kann man sich alle Einstellungen anzeigen lassen. Nennt man zusätzlich eine Einstellung, so wird nur diese angezeigt. Nennt man zusätzlich einen Wert, so wird die genannte Einstellung auf diesen Wert gesetzt.

Einfacher geht es mit **`fossil ui`**, wodurch sich der Standard-Webbrowser öffnet und ein User Interface (**`ui`**) anzeigt, in dem man viele Einstellungen vornehmen kann (Reiter „Admin“, Punkt „Settings“).

Es kann auch ein Unterverzeichnis mit dem Namen **`.fossil-settings`** angelegt werden mit Dateien, die den Namen der Einstellungen tragen, beispielsweise **`binary-glob`** für die Dateien, die binäre Inhalte haben dürfen. Diese Konfigurationsdateien sollten eingecheckt und damit versioniert werden.

Tagging

Um Versionen einen Namen zu geben oder sie anderweitig zu kennzeichnen, verwendet man sogenannte Tags.

`git tag 1.0b `*`version`* fügt der genannten Version das Tag „**`1.0b`**“ hinzu. Es kann aus beliebigem Text bestehen, aber Release-Nummern sind häufig.

`git tag` listet alle existierenden Tags auf.

In Git müssen Tags eindeutig sein, d. h. es kann keine zwei Versionen geben, die dasselbe Tag tragen. Daher kann man Tags zur Identifikation verwenden beim Auschecken:

`git checkout 1.0b`

`fossil tag add 1.0b `*`version`* fügt der genannten Version das Tag „**`1.0b`**“ hinzu. Um es der aktuellen Version hinzuzufügen, kann man **`current`** angeben. Die Option **`--propagate`** führt dazu, dass auch alle folgenden Versionen das Tag übernehmen, bis es mittels **`fossil tag cancel 1.0b`** gestoppt wird.

`fossil tag find 1.0b` findet alle Versionen, die ein bestimmtes Tag tragen.

`fossil tag list` listet alle existierenden Tags auf. Hierbei werden auch Branches aufgelistet.

Hoppla – bitte zurück!

Hat man eine Datei verfrickelt, möchte man gerne den letzten Stand aus dem Repo wiederherstellen.

`git checkout -- dateiname`
Das funktioniert nur, wenn die Datei noch nicht „gestaget“ wurde. Falls sie bereits gestaget wurde, gibt es aber keine

`fossil revert `*`dateiname`*
setzt diese Datei zurück.
`fossil revert`

Warnung – es passiert einfach nichts.
`git reset` ***`dateiname`***
holt die Datei aus dem Stage zurück, so dass anschließend das obige Kommando erfolgreich ausgeführt werden kann.

setzt *alle* Dateien zurück.

Auch dieses Kommando kann mit **`fossil undo`** rückgängig gemacht werden.

Geht es darum, alle Änderungen an allen Dateien gegenüber dem remote Repo zu verwerfen, kann man sich das remote Repo erneut ziehen und alles überschreiben.

`git fetch origin` zieht das remote Repo über das eigene lokale.
`git reset --hard origin/master` ersetzt alle Dateien mit denen aus dem lokalen Repo.

In Fossil fällt das schwer, wenn man Autosync verwendet, weil dann alle Commits direkt auf dem remote Repo ausgeführt wurden. Ohne Autosync kann man einfach alles wegwerfen und ggf. das Repo von remote erneut klonen und dann in einem frischen Verzeichnis öffnen.

Dateien ignorieren

Bei der Entwicklung fallen oft viele Dateien an, die zwar im Arbeitsverzeichnis oder darunter liegen, aber nicht versioniert werden sollen. Das können Dateien mit rein lokaler Bedeutung sein, oder es kann sich um generierte Dateien handeln, die das Repo nur aufblähen würden.

Bei Git erzeugt man eine Datei namens **`.gitignore`**, in der entsprechende Muster eingetragen werden, beispielsweise **`*.so`**, um alle Shared Objects zu ignorieren. Bei mehreren Mustern ein Muster pro Zeile.

Bei Fossil trägt man die zu ignorierenden Dateimuster ein mittels **`fossil settings ignore-glob '*.so,*.o'`**. Dies kann man auch in der Web-Oberfläche (s. u.) eintragen.

Wahlweise kann man im Verzeichnis **`.fossil-settings`** auch eine Datei namens **`ignore-glob`** erzeugen mit einem Muster pro Zeile.

Grafische Oberfläche

Gelegentlich wird bei Git **`gitk`** mitgeliefert, mit dem man die History und die Änderungen visualisieren kannn. Ansonsten muss man externe Programme bemühen, von denen es reichlich gibt.

`fossil ui` startet die grafische Oberfläche im Webbrowser, über die man die Einstellungen, die History, die Änderungen, das Wiki, den Bugtracker und das Forum erreicht. In der History (Timeline) kann man einfach zwei Versionen nacheinander anklicken – schon werden die Änderungen dargestellt.

Wiki, Bugtracker, Technotes, Forum

Diese hilfreichen Tools gibt es bei Git nicht. Git interpretiert die Unix-Philosophie als *„Mach nur eine Sache und diese gut.“* Fossil interpretiert die Unix-Philosopie als *„Mach es so, dass es funktioniert.“*, unter Berücksichtigung des KISS-Prinzips *(keep it simple and stupid)*, denn Fossil ist klein, Repos bestehend aus nur einer einzelnen Datei usw.

Fossil-Wiki

Das Wiki sieht man sofort, wenn man **`fossil ui`** oder **`fossil server`** aufruft. Man kann auch beliebig editieren – die Änderungen werden beim Synchronisieren mit übertragen. Allerdings gibt es hier eine „last one wins“-Strategie, d. h. die Wiki-Seiten werden nicht versioniert. Dafür gibt es aber Abhilfe, indem man über den URL
`http://locahost:8080/doc/`*`version/pfad_zur_datei`*

aufruft und dabei ***`pfad_zur_datei`*** eine Wiki-Datei (**`*.wiki`**), eine Markdown-Datei (**`*.md)`** ist, die entsprechend interpretiert werden. Dateien mit den folgenden Erweiterungen werden 1:1 ausgeliefert: **`.css .gif .html .jpg .jpeg .png .txt`**

Diese Dateien befinden sich dann ganz normal im Arbeitsverzeichnis und werden mit versioniert. Allerdings können nur Leute mit Checkin-Recht diese Dateien bearbeiten, das Recht zum Bearbeiten von Wiki-Seiten genügt nicht.

Technotes

Technotes sind besondere Wiki-Seiten, die mittels des URL **`/technoteedit`** erstellt werden. Im Unterschied zu gewöhnlichen Wiki-Seiten erscheinen sie in der History (Timeline) als ein eigener Typ und sind mit ihrem Erstellungszeitpunkt verknüpft, während sich Wiki-Seiten öfters mal ändern.

Fossil-Bugtracker

Unter „Tickets“ findet man den Bugtracker von Fossil. Diese werden wie Dateien versioniert und zwischen den Repos beim Synchronisieren ausgetauscht.

Um aus einem Ticket auf Versionen hinzuweisen, genügt es, die ID in eckigen Klammen in den Text einzufügen. Das geht sowohl im Feld „Version“ als auch in der Beschreibung. Selbstverständlich kann man auch auf alle anderen URLs verweisen. Sofern diese relativ sind, können sie auf alles im Repo verweisen: Versionen, Artefakte, Forumsbeiträge, Tickets, Wiki-Seiten usw.

Fossil-Forum

Zusätzlich zum Wiki gibt es das Forum, in dem sich die Anwender austauschen können. Durch das Antworten auf Forumsbeiträge können Diskussionen entstehen.

Auch im Text von Forumsbeiträgen kann man auf Versionen mithilfe der ID in eckigen Klammern verweisen, allerdings nicht in den Überschriften.

Forumsbeiträge kann man zwar „löschen“, sie bleiben in der Historie aber erhalten.

Fossil als Webserver

Fossil kann als Webserver dienen oder über einen solchen angesprochen werden. Auf diese Weise können ganze Repos veröffentlicht werden – einschließlich Dokumentation auf Wiki-Seiten, Bugtracker und Forum. Durch entsprechende Benutzer- und Sicherheitseinstellungen können die Zugriffe reglementiert werden, sie unter „Admin“, „Users“.

Auch sind fortgeschrittene Dinge wir automatische Benachrichtigungen per E-Mail möglich. Das und vieles andere geht aber über ein kleines Cheat-Sheet hinaus.

Wie geht's weiter?

Die eingebaute Hilfe kann einem – neben dem Web – oft gut weiterhelfen.

`git help` ***`kommando`*** **`fossil help`** ***`kommando`***

Die Original-Webseiten sind selbstverständlich eine gute Quelle:

- http://git-scm.com
- http://fossil-scm.org

Darüber hinaus gibt es auch sehr empfehlenswerte Webseiten:

- https://trunkbaseddevelopment.com
- https://www.toptal.com/software/trunk-based-development-git-flow
- http://kean.github.io/post/trunk-based-development

Was sollte man überdenken?

Nur manche der veränderten Dateien einzuchecken ist oft keine gute Idee, denn genau diese Kombination – manche Dateien verändert, manche unverändert – hat man ja wohl kaum komplett so testen können. Bei diesem Vorgehen schleichen sich leicht Inkonsistenzen ein.

Ebenso gefährlich ist das nachträgliche Ändern der Historie, was Git durchaus erlaubt.

Problematisch ist auch das sogenannte Rebasing, was ebenfalls nur in Git möglich ist.