Herbert Braun

Unvergessen

Erste Schritte mit dem Versionskontrollsystem Git und mit GitHub

Das robuste und flexible Versionskontrollwerkzeug Git archiviert Inhalte und unterstützt Teamwork. Spätestens seit GitHub zur führenden Open-Source-Plattform wurde, ist Git auch bei Anfängern in Sachen Versionskontrolle en vogue – dabei gibt es durchaus ein paar Einstiegshürden.

Versionsverwaltung zählt zu jenen Werkzeugen, die man nicht mehr missen will, wenn man sich einmal daran gewöhnt hat. Ein paar Stunden Einarbeitungszeit sind dafür in jedem Fall gut investiert – und das gilt nicht nur für Entwickler: Von Versionierung profitieren auch Textautoren, Grafiker, Excel-Artisten und alle anderen, deren Schöpfungen sich als Dateien manifestieren.

Ein Versionsverwaltungssystem (Version Control System, VCS) protokolliert die eigene Arbeit: Wie ist diese merkwürdige Anweisung in meinen Code geraten, die jetzt Probleme macht? Ah, vor sechs Monaten habe ich versucht, damit einen Bug zu fixen. Solche Erkenntnisse nützen umso mehr, je umfangreicher und langwieriger ein Projekt wird und je mehr Leute daran arbeiten. Als Archiv stellt das System ein Backup bereit. Und die Wiederherstellbarkeit längst überholter Versionen kann viele Arbeitsstunden retten.

Überraschenderweise kann selbst etwas so Sprödes wie eine Versionsverwaltung in Mode kommen. Maßgeblich liegt das an GitHub, dem Dreh- und Angelpunkt der Open-Source-Szene. Diese Plattform machte Git auch weit außerhalb der klassischen Software-Entwicklung bekannt.

Zentralisiert versus verteilt

Vor Git verwendeten Entwickler für die Versionskontrolle meist Concurrent Versions System (CVS) oder später Apache Subversion. Subversion verdrängte die ältere Generation der Versionskontrollsysteme und ist nach wie vor weit verbreitet. Als zentralisiertes System dreht sich bei Subversion alles um einen Server, der das Repository – also die Datenbank mit dem Code und dem Versionsarchiv – beherbergt. Von dort checkt jeder Mitarbeiter die benötigten Dateien aus und nach erfolgreicher Bearbeitung wieder ein.

Git dagegen ist ein verteiltes System. Das bedeutet: Jeder Entwickler arbeitet mit einer lokalen Kopie des kompletten Repository. Typischerweise „klont" er sich dieses von einem Server, pflegt Änderungen ein und aktualisiert das gehostete Archiv, falls er die Rechte dazu hat.

Dieser Ansatz hat ein paar Vorteile. So gibt es bei zentralisierten Versionskontrollsystemen mit dem Server einen Single Point of Failure: Fällt dieser aus, stockt die Arbeit, zusätzliche Backup-Lösungen müssen gegen mögliche Datenverluste vorbeugen. In verteilten Systemen dagegen hat jeder Beteiligte ein vollständiges Backup und kann damit auch ohne Netzwerkverbindung arbeiten. Erst beim Hochladen auf den Server gehen Daten durch die Leitung. Und nicht einmal das ist nötig: Entwickler können auch ohne zentralen Server direkt untereinander Code austauschen.

Gitologie

Den Grundstein für das Git-System legte 2005 Linus Torvalds. Er benötigte für die Entwicklung des Linux-Kernels eine Versionsverwaltung, die mit einer riesigen Menge Code und sehr vielen Mitarbeitern umgehen konnte. Zuvor hatte die Mehrzahl der Linux-Entwickler anhand von BitKeeper die Vorzüge einer verteilten Versionsverwaltung kennengelernt. Als eines der ersten Systeme dieser Art stand BitKeeper jedoch unter einer kommerziellen Lizenz.

Nachdem die Allianz mit dem Hersteller zerbrochen war, musste schnell quelloffener Ersatz her. Im April 2005 programmierte Torvald deshalb sein eigenes Versionskontrollsystem, für das seither der Google-Angestellte Junio Hamano zuständig ist. Torvalds' Humor findet sich in der Namenswahl wieder: „git" steht

im britischen Englisch für „Blödmann“ und stellt sich in seiner Manpage als „stupid content tracker“ vor.

Gitarrenbau

Allen Ernstes würde man Git wohl kaum dumm nennen, und besonders einfach gestrickt ist es auch nicht. Trotz gewisser Einstiegshürden und reichhaltiger Möglichkeiten bleibt der Aufwand, um mit Git produktiv arbeiten zu können, überschaubar.

Linux-Anwender installieren sich Git einfach über den Paketmanager. Unter Mac OS und Windows lädt man sich die Software von www.git-scm.com herunter. Wie alle gängigen Versionsverwaltungen ist Git in erster Linie ein Konsolenprogramm, doch liegt den Downloads auch ein Client mit grafischer Oberfläche bei.

Lange stand Git in dem Ruf, sich in Windows-Umgebungen nicht besonders wohlzufühlen. Um die Lücke zu Unix-basierten Systemen zu schließen, enthält das Windows-Paket zusätzlich einen SSH-Client für die verschlüsselte Dateiübertragung und eine Bash-Konsole. Fast alle Git-Anweisungen lassen sich auch in der Windows-Eingabeaufforderung eingeben, aber Git Bash kann auch Git-Kommandos vervollständigen. Achtung: Das Stammverzeichnis Ihrer Festplatte erreichen Sie in Bash nicht über c:, sondern über cd /c.

Beispielprojekt soll eine neue Website sein – ein frisches Verzeichnis mit je einer HTML-, CSS- und JavaScript-Datei. Navigieren Sie mit der Konsole dorthin und geben Sie ein:

```
git init
```

Git legt daraufhin ein leeres Repository im Unterverzeichnis .git an (das Sie wegen des vorangestellten Punktes mit einem ls-Kommando in der Bash nicht sehen). Es enthält ein paar Dateien und Ordner, die Sie vorerst nicht zu interessieren brauchen. Für jedes Verzeichnis kann es nur ein Repository geben. Um das Repository zu löschen, brauchen Sie nur den .git-Ordner zu löschen; beim Verschieben oder Kopieren des Arbeitsverzeichnisses nehmen sie ihn einfach mit.

Über den Zustand des Repository können Sie sich jederzeit mit git status informieren. Die Ausgabe sollte lauten:

```
On branch master
Initial commit
Untracked files:
```

… gefolgt von einer Liste der im Verzeichnis abgelegten Dateien. master ist der Standard-Zweig („Branch“); ein Commit legt eine neue Revision des Projekts ins Repository.

Idealerweise sollten alle Dateien im Projektverzeichnis ins Repository Einzug halten, damit Git Sie nicht ständig wegen „untracked files“ anmotzt. Sie können Dateien ausschließen, indem Sie eine Textdatei namens .gitignore anlegen und darin Dateinamen einzeln oder mit Platzhaltern ausschließen (zum Beispiel *~ für Dateien, deren Name auf eine Tilde endet).

Comm mit!

Bevor Sie Dateien ins Repository committen, markieren Sie diese – „stagen“ heißt das bei Git. Die add-Anweisung bevölkert die Staging Area:

```
git add .
```

Die Staging Area ist eine Art Verladerampe. Von dort können Sie die Änderungen an Ihren Dateien in verschiedene Commits sortieren. Das ist praktisch, wenn Sie im Eifer des Gefechts nicht nur ein neues Feature in Ihren Code integriert haben, sondern auch an anderer Stelle noch Rechtschreibkorrekturen vorgenommen oder Ihren Code aufgeräumt haben. Diese drei Änderungen können Sie der Reihe nach einzeln in die Staging Area packen und von dort aus committen, damit die Versionsgeschichte nachvollziehbar bleibt.

Git staget nur geänderte Dateien, sodass git add . im Normalfall das Gewünschte erledigt – auch für Dateien in Unterverzeichnissen. Zur Sicherheit schauen Sie am besten mit git status noch mal nach. Hier verrät Git auch, wie man Dateien aus der Staging Area wieder entfernt: durch die Anweisung

```
git rm --cached <dateiname>
```

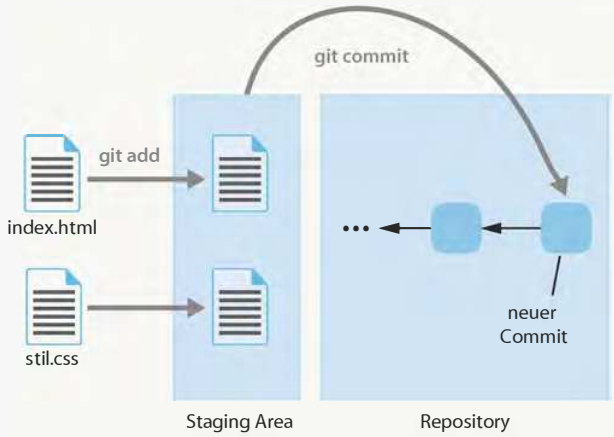
Nun stoßen Sie den ersten Commit an:

```
git commit
```

„Please tell me who you are“ bittet Git Sie jetzt vorher, falls Sie noch keinen Namen und keine Mail-Adresse hinterlegt haben.

Einen neuen Commit anlegen

Zuerst schieben Sie mit git add Dateien in die Staging Area, wodurch sie für den nächsten Commit vorgemerkt sind. Der Befehl git commit sichert den neuen Commit dann im Repository.



Die dafür nötigen Anweisungen zeigt es freundlicherweise gleich an; sie beginnen alle mit git config. Git kennt Konfiguration auf drei Ebenen: System (etc/gitconfig im Programmverzeichnis), Benutzer (.gitconfig im Benutzerverzeichnis) und Repository (in .git/config). Die ersten beiden Ebenen sprechen Sie mit git config --system beziehungsweise git config --global an. Nach Änderungen können Sie mit git config --list testen, ob alles passt.

Hinter Gittern

Einem erfolgreichen git commit sollte nun nichts mehr im Weg stehen. Git übergibt an den Standard-Konsoleneditor, mit dem Sie jetzt eine Commit-Message eingeben sollen – in der Regel ist das vi(m). Dieser Editor-Veteran treibt manchem den Angstschweiß ins Gesicht, aber ein paar elementare Kenntnisse reichen: i wechselt in den Editiermodus, Esc zu den Kommandos. Dort können Sie mit :w speichern und mit :q beenden (oder beides mit :wq kombinieren). Bei unüberwindbarer vi-Phobie können Sie die Commit-Message auch direkt anhängen: git commit -m "Nachricht".

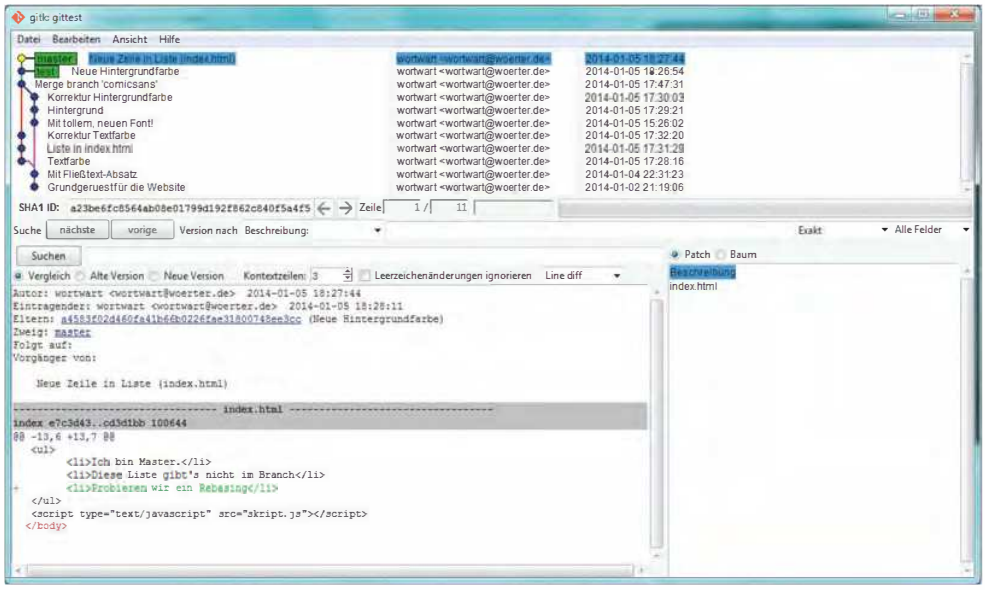
Eine gelungene Commit-Message informiert darüber, welchen Zweck eine Änderung erfüllt. Als guter Stil gilt, in der ersten Zeile als Überschrift eine knappe Zusammenfassung zu liefern. Die zweite Zeile lassen Sie leer und erläutern nachfolgend detailliert die Änderungen. Für eine solche vorbildliche und umfangreiche Commit-Message taugt das Anhängsel -m "Nachricht" dann allerdings nicht.

Das Ergebnis des Commits können Sie im Log begutachten: Die Anweisung git log listet alle Commits auf. Deren Namen sind allesamt 40-stellige Hex-Knäuel – mit dem Algorithmus SHA1 errechnete Hashes des Datenbestands. Die log-Anweisung informiert Sie außerdem über Autor, Datum und Beschreibungstext des Commits.

Die Filterungs- und Formatierungsmöglichkeiten des Git-Logs lassen nichts zu wünschen übrig: Die automatische Vervollständigung zu git log -- listet fast hundert Optionen auf. Eine gute Alternative zum Standard-Log steht Ihnen mit der grafischen Ausgabe einer im Git-Paket enthaltenen Tcl/Tk-Anwendung zur Verfügung, die Sie aus dem betreffenden Arbeitsverzeichnis über die Konsole mit gitk starten.

Differenziert

Editieren Sie ein wenig herum in den Dokumenten und legen Sie mit git add . und git commit ein paar Commits an. Sie können diese beiden Kommandos auch mit git commit -a zusammenfassen. Zum Löschen und Umbenennen stehen Ihnen die an Unix angelehnten Kommandos git rm und git mv zur Verfügung.



Das GUI-Tool gitk aus dem Git-Paket stellt die Geschichte eines Repository übersichtlich dar.

Wenn sich eine zu große Zahl von Änderungen vor einem Commit angehäuft hat, können Sie diese auf mehrere Patches aufteilen. Dazu wählen Sie beim Staging mit `git add -i` den interaktiven Modus und im folgenden Dialog die Patch-Option, die Sie bei jedem bearbeiteten Textblock fragt, ob er ins Staging soll. Was nach dem Commit übrig bleibt, können Sie einfach mit `git add` ins Repository einspielen.

Die Änderungen etwa an der Datei skript.js stellt Git mit `git diff skript.js` dar. Die von @-Zeichen eingeschlossene Zeile zeigt die Zeilennummern an; `@@ -10,6 +10,7 @@` bedeutet zum Beispiel, dass das folgende Fragment bei beiden Versionen in Zeile 10 beginnt und in der aktuellen Fassung 7 Zeilen umfasst. Gelöschte Zeilen erscheinen rot, eingefügte grün, geänderte kombinieren beides.

Nach unbefriedigenden Änderungen oder einer Code-Vernichtung in geistiger Umnachtung möchten Sie den früheren Stand Ihrer Arbeit wiederherstellen? Dafür gibt es `git checkout`:

```
git checkout master skript.js
```

Solange Sie keine Verzweigungen (sogenannte „Branches“) haben, können Sie `master` auch weglassen; es bezeichnet den Standardzweig. Natürlich sind auch hier Wildcards zulässig, um mehrere oder alle Dateien auf einmal wiederherzustellen – aber passen Sie auf, dass Sie keine wertvollen Inhalte im Arbeitsverzeichnis überschreiben, die noch nicht eingecheckt wurden.

Zeitmaschine

Der einfachste Weg zu älteren Versionen führt über den 40-stelligen Hash des Commits – den Sie glücklicherweise auf die ersten vier Stellen abkürzen können, zum Beispiel:

```
git diff e87f index.html
```

Dieser Befehl zeigt die Unterschiede zwischen der aktuellen Version der Datei index.html und der im Commit e87f. Damit die ältere Version die neue ersetzt, schreiben Sie:

```
git checkout e87f index.html
```

Die neue alte Datei ist bereits in der Staging Area und somit für den nächsten Commit vorgesehen. Mit `checkout` können Sie auch vollständig auf den Status eines beliebigen Commits zurückwechseln, indem Sie den Dateinamen weglassen:

```
git checkout e87f
```

Sie können sich jetzt umsehen, sollten aber ohne weitere Vorkehrungen keine Änderungen an Ihren Dateien vornehmen. Mit

```
git checkout master
```

kommen Sie zum aktuellen Stand Ihrer Entwicklung zurück. Wenn Sie Ihre Arbeit verwerfen und endgültig zu einem früheren Stand zurückkehren wollen, verwenden Sie den Befehl

```
git reset --hard e87f
```

Damit schmeißen Sie alle Änderungen weg, die Sie nach diesem Commit vorgenommen haben.

Sollten Sie es sich doch wieder anders überlegen und zu einer neueren Version zurückkehren wollen, könnte `git log` eine kleine Panikattacke auslösen: Hier findet sich keine Spur von den späteren Commits. Doch Git vergisst nichts. Mit `git log -g` können Sie die komplette Versionsgeschichte einsehen und wie gehabt zu einer beliebigen Revision wechseln.

Verzweigt

Die Geschichte eines Git-Repository verläuft in der Regel nicht linear, sondern steckt voller Verzweigungen und Zusammenführungen. Ein alltägliches Praxisbeispiel: Sie arbeiten an einem Website-Relaunch, müssen aber sofort einen Tippfehler in der Version beheben, die derzeit noch online steht. Dazu legen Sie von dieser älteren Version eine Verzweigung an, einen „Branch“. So können Sie an beiden Baustellen parallel arbeiten und die Ergebnisse später zusammenführen („mergen“).

Springen Sie mit `git checkout e87f` zu der Version, die derzeit online ist – in diesem Beispiel wäre das der Commit mit dem Hash e87f. Dann legen Sie einen neuen Branch an:

```
git branch korrektur
```

Anschließend wechseln Sie mit `git checkout korrektur` in diesen frisch erzeugten Branch. Auch dieses Anweisungspaar können Sie zu einem Kommando zusammenfassen, nämlich:

```
git checkout -b korrektur
```

Beheben Sie in der Datei den Fehler, schauen Sie sich das Ergebnis an und laden Sie die Dateien dann auf den FTP-Server. Anschließend wechseln Sie zurück in den master-Branch, um am Relaunch weiterzuarbeiten:

```
git checkout master
```

Das stellt den alten Status quo wieder her; auch `git log` zeigt den neuen Branch nicht an. Selbstverständlich können Sie in beiden Branches Änderungen committen, indem Sie mit `git checkout` zwischen ihnen wechseln.

In Git gilt es als guter Stil, neue Features und Code-Reparaturen erst einmal in einen Branch auszulagern, bis sie reif zur Veröffentlichung sind – auch mehrmals täglich. So bleibt im master-Branch immer eine stabile Code-Basis. Den Relaunch einer Webseite würden Sie ebenfalls in einen eigenen Branch auslagern und dort die Entwicklung vorantreiben, während die stabile Version Ihrer Webseite unberührt im master-Branch liegt.

Branching funktioniert in Git besonders schnell und robust. Das liegt an der Art, wie das Repository aufgebaut ist: Es ist ein Mini-Dateisystem, das keine Diffs, sondern komplette Dateien speichert und identische Dateiversionen mit Zeigern referenziert. Ein neuer Branch ist also zuerst nichts weiter als eine neue Referenz und kopiert wird nur, was sich geändert hat.

Wiedervereinigt

Um die unterschiedlichen Varianten wieder zusammenzuführen, wechseln Sie mit `git checkout master` in den Branch, der nach dem Merge übrig bleiben soll, und führen aus:

```
git merge korrektur
```

Im einfachsten Fall haben Sie den `master` nicht verändert, seit sich der Branch von ihm getrennt hat. In diesem Fall muss Git nur das Arbeitsverzeichnis

aktualisieren, was Sie an der Nachricht „Fast-forward“ in der Merge-Bestätigung erkennen.

Auch wenn sich beide Zweige geändert haben, legt Git oft genug Cleverness an den Tag, um alles sauber zusammenzufügen – oft sogar, wenn es sich um Modifikationen in derselben Datei handelt. In diesen Fällen leitet Git Sie zum Commit-Dialog weiter, mit dem Sie den Merge abschließen. Doch auch der beste Algorithmus stößt gelegentlich an seine Grenzen, wie bei dieser Ausgabe:

```
Auto-merging stil.css
CONFLICT (content): Merge conflict in stil.css
Automatic merge failed; fix conflicts
and then commit the results
```

In Fällen wie diesem wurde in beiden Branches die gleiche Datei an den gleichen Stellen bearbeitet. Git hat direkt in der Datei die problematischen Stellen markiert, zum Beispiel:

```
body {
<<<<<<< HEAD
    font-family: Verdana, Arial;
=======
    font-family: "Comic Sans MS";
>>>>>>> korrektur
}
```

Die Gleichheitszeichen trennen die Varianten in `korrektur` und in `HEAD`, dem aktuell geöffneten `master`-Branch. Reparieren Sie die gekennzeichneten Stellen und aktualisieren Sie das Repository mit `git commit -a`. Git hat sogar schon die Commit-Message vorbereitet („Merge branch ‚korrektur‘“).

Geschichtsklitterung

Die Anweisung `git branch --merged` listet nun `korrektur` auf. Zum Abschluss der Operation können Sie diesen für eine bessere Übersichtlichkeit löschen:

```
git branch -d korrektur
```

Das bedeutet nicht, dass Git die Geschichte umschreibt: Der Branch lässt sich immer noch rekonstruieren, Sie können ihn nur nicht mit `git checkout korrektur` öffnen und weiterbearbeiten. Wie zu jeder Revision im `master` können Sie auch zu denen in gelöschten Branches zurückkehren, wenn Sie über `git reflog` den Commit-Hash herausgefunden haben. Wie gesagt: Git vergisst nichts – und schon gar nicht so etwas Wichtiges wie die Eltern-Commits eines Merge.

Tele-Git

Git lässt sich ohne zentralen Server benutzen – das heißt, streng genommen gibt es in Git keinen Server, sondern nur Remote-Repositories, denen es egal ist, ob sie auf einem Arbeitsrechner oder einem Server liegen und über welchen Kanal die Verbindung läuft. In der Praxis laufen Projekte mit mehreren Beteiligten dennoch über einen ständig erreichbaren Server mit unterschiedlichen Lese- und Schreibrechten. Firmen werden sich ohne großen Aufwand meist selbst einen aufsetzen, Open-Source-Projekte können dafür kostenlos auf Plattformen wie GitHub, Bitbucket oder Gitorious zurückgreifen.

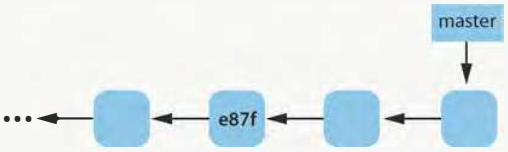
Um ein Remote-Repository mit Git in den eigenen Arbeitsbereich zu übernehmen, klont man es. Das schließt auch die komplette Projektgeschichte ein. Dennoch entsteht keine identische Kopie: Das lokale Repository stuft die Branches im Original als „Remote Branches“ ein, die zunächst verborgen sind. Holt man sich spätere Updates von der Originalquelle, muss man diese erst in den lokalen Arbeits-Branch mergen.

Der Vorteil dieser Git-spezifischen Arbeitsweise: Der Benutzer kann seine Arbeit besser von der seiner Kollegen trennen. Git eignet sich daher sehr gut für Open-Source-Projekte, wo sich die Mitwirkenden meist kaum kennen. Diese Flexibilität hat allerdings den Preis, dass ein ungeschickter Anwender seine Änderungen in einen ungenutzten Branch schreiben kann, wo sie unbeachtet verschwinden.

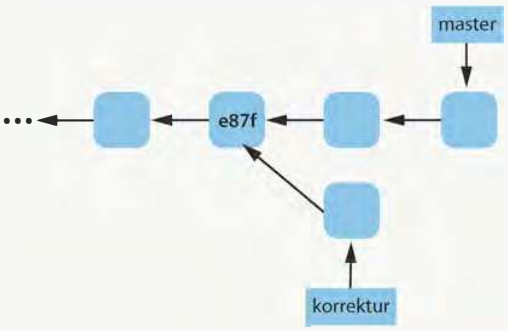
Um die Arbeit mit Remote-Repositories zu üben, melden Sie sich am besten bei GitHub an und forken ein beliebiges Projekt

Branches und Merges

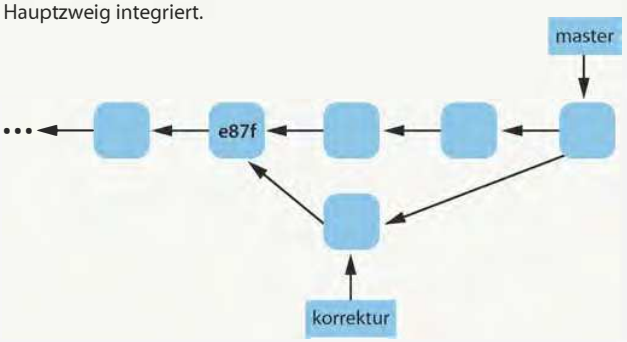
In einer linearen Entwicklung legen Sie nacheinander Commits auf dem Branch master an. Die Zeiger verweisen jeweils auf den vorigen Commit.



Von einem älteren Commit mit dem Hash e87f zweigen Sie einen neuen Branch ab. Dort nehmen Sie Korrekturen vor, von denen der Branch master unberührt bleibt.



Ein Merge verschmilzt schließlich den Branch korrektur mit dem Branch master. Jetzt sind die Korrekturen auch im Hauptzweig integriert.



– das erfordert nichts weiter als das Anklicken eines Buttons („Fork“). Jetzt haben Sie eine private Version des Open-Source-Projekts in Ihrem GitHub-Profil.

GitHub stellt jedem Repository eine Klon-URL für HTTPS- und SSH-Verbindungen zur Verfügung. Git kennt noch eine dritte Variante, das Protokoll git://; dieses hat weniger Overhead als HTTPS, kennt aber keine Authentifizierung, weswegen GitHub damit nur den Download zulässt. SSH ist ein wenig umständlich einzurichten, daher soll für den ersten Versuch HTTPS genügen.

```
git clone https://github.com/...
```

`clone` legt das lokale Projektverzeichnis an und kopiert die ausgecheckten Arbeitsdateien sowie den Repository-Unterordner hinein. Mit `git remote` zeigen Sie die Kurznamen der verbundenen Remote-Repositories an – das ist in diesem Fall nur `origin`. Falls Ihr Projekt ein Fork ist, sollten Sie noch die URL des Original-Repository hinterlegen, um von dort Updates einspielen zu können:

```
git remote add updates https://...
```

Nun sollte `git remote` die Repositories `origin` und `updates` auflisten. Um frische Daten von Letzterem herunterzuladen, schreiben Sie:

```
git fetch updates
```

Die Updates des Originalprojekts werden Sie jetzt aber vergeblich im Arbeitsverzeichnis suchen: Git hat sie in einem Branch `updates/master` abgelegt. Sie müssen die Zweige also zusammenführen („mergen“):

```
git merge updates/master
```

Um die beiden letzten Schritte zusammenzufassen, können Sie auch `git pull updates` verwenden.

Hochschieben

Nun gilt es, das eigene Remote-Repository auf GitHub – `origin` – zu aktualisieren. Das tun Sie mit `push`:

```
git push origin master
```

Ohne zusätzliche Konfiguration erwartet `push` als zweites Argument den hochzuladenden Branch. Git wird Sie bei jedem Push nach Benutzername und Passwort fragen. Mit der Schreibweise https://user@domain.com/… können Sie sich den Benutzernamen sparen – und wenn es Sie nicht stört, dass das GitHub-Passwort im Klartext auf der Platte liegt, geben Sie es mit https://user:passwort@… gleich mit an. Die URL ändern Sie nachträglich mit:

```
git remote set-url origin https://...
```

Auf Dauer empfiehlt sich, SSH für die Server-Kommunikation zu benutzen. Dabei erzeugen Sie einen Schlüssel zur Authentifizierung – Details dazu verrät [1]. Der GitHub-Client, den man alternativ zur Kommandozeile nutzen kann, überträgt Daten standardmäßig über HTTPS.

Falls noch andere Benutzer Schreibrechte für `origin/master` haben, könnte es sein, dass GitHub `push` mit einer Fehlermeldung beantwortet: Falls es nämlich seit dem letzten `fetch` oder `pull` neue Commits gab, verweigert Git den Upload – selbst wenn ganz andere Dateien betroffen sind. In diesem Fall bleibt nur, die aktuelle Version herunterzuladen, lokal zu mergen und einen neuen Upload-Versuch zu starten.

GitHub

Das beschriebene Ein-Klick-Forken von Repositories passt gut zum Git-Konzept der Dezentralisierung und der schnell angelegten Branches. Wer möchte, dass das Originalprojekt Änderungen aufgreift, aber keine Schreibrechte dafür hat, schickt ihm einen Pull Request, also ein „Zieh dir doch bitte meine Änderungen in dein Repository“.

Repositories ordnen sich bei GitHub dem Benutzernamen unter. Daher lassen sich nach Belieben neue Projekte starten, da der Namensraum nicht verstopft ist. Alle mit Gratis-Account veröffentlichten Projekte sind frei zugänglich; mit den privaten Repositories verdient GitHub sein Geld.

Mit Gist stellt GitHub Entwicklern ein kleines Tool zur Verfügung, das Code-Schnipsel verwaltet und diese ähnlich wie komplette Projekte verteilt. Auch die direkte Einbindung des gehosteten Codes (zum Beispiel per `<script src="…">`) ist möglich.

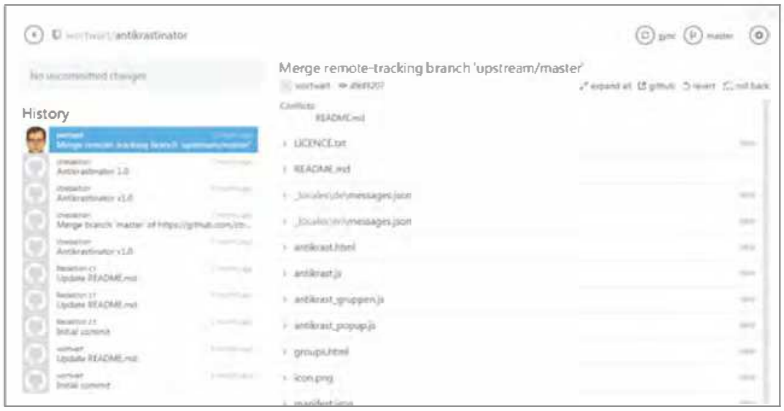
Weil GitHub Repository-Inhalte direkt an den Browser senden kann, lassen sich damit auch statische Webauftritte aufziehen. Um GitHub Pages zu nutzen, legt man ein Repository mit dem

Werkzeugkasten

Einige Werkzeuge versuchen, den Umgang mit Git zu erleichtern – angefangen bei den im Git-Kit enthaltenen Tools **gitk** zur Visualisierung der Logs und **Git GUI**, einem grafischen Git-Frontend. Ob Letzteres jedoch einfacher zu bedienen ist als das Konsolenprogramm, darf man bezweifeln. Sehr viel übersichtlicher präsentiert sich der schmucke **GitHub-Client** für Windows, Mac und Android.

Für die meisten Editoren und IDEs gibt es Git-Plug-ins; Nutzer von Sublime Text etwa können zwischen **sublime-text-git** und **SublimeGit** wählen. Per Menü oder Tastenkürzel pflegt man damit Text ins Repository ein, und für Commit-Messages muss man sich nicht mehr mit vi herumschlagen.

In Sachen Workflow lässt Git den Anwendern freie Hand, was nicht immer ein Vorteil ist. Die Git-Erweiterungen **Gitflow** und **Legit** geben ein Gerüst vor und packen Standardabläufe zu rasch nutzbaren Befehlen zusammen.



GitHub-Anwender können mit diesem schön gestalteten grafischen Client ihre Versionsverwaltung erledigen.

Namen nutzername.github.io an. Anschließend pusht man HTML-, CSS-, JavaScript- und sonstige Dateien. Die Webseite ist standardmäßig unter http://nutzername.github.io abrufbar; über einen CNAME-Eintrag kann man aber auch eine eigene Domain aufschalten [2].

Serverseitige Skripte sind bei den GitHub Pages nicht möglich, daher bietet sich ein HTML-Generator wie Jekyll [3] oder das darauf aufsetzende Octopress an, das speziell für GitHub Pages entworfen wurde [4]. Dem Verzicht auf serverseitiges Skripting steht auf der Haben-Seite eine sichere und werbefreie Hosting-Umgebung mit praktischem Upload-Mechanismus gegenüber.

Di-Gitalisierung

Blog-Hosting, eine Jobbörse, Projekte favorisieren und kommentieren, Nachrichten schreiben, Nutzern folgen, Aktivitäts-Streams … GitHub ist eindeutig mehr als ein Ort, um Code auszutauschen. Ebenso lässt es sich als soziales Netzwerk ansehen, als eine Art Facebook für (Web-)Entwickler – mit ähnlicher Monopolstellung.

Auch für die Karriere-Planung erweist es sich als wichtig: Ein gut gepflegter GitHub-Account zählt mancherorts mehr als Lebenslauf und Zeugnisse. GitHub dient mittlerweile als Vorbild für Teamwork-Plattformen. So beschreibt der Autor der neuen Webanwendung Penflip diese als „GitHub for writers".

Bei jedem System, das man wählt, entscheidet man sich auch für ein Umfeld aus Tools, Diensten und anderen Benutzern – und im Fall von Git könnte dieses Ökosystem kaum lebendiger sein. Umso besser, dass sich hier auch eine technisch überzeugende Software durchgesetzt hat. (dbe)

Literatur

[1] GitHub Help: Generating SSH Keys: https://help.github.com/articles/generating-ssh-keys
[2] GitHub Help: Setting up a custom domain with Pages: https://help.github.com/articles/setting-up-a-custom- domain-with-pages
[3] Oliver Lau, Statisch, praktisch, gut, Jekyll generiert Blogs und andere Webseiten, c't 25/13, S. 184
[4] Ragni Serina Zlotos, Kommandozeilenblogger, Octopress erzeugt und veröffentlicht Blogs mit statischem HTML, c't 9/13, S. 158

www.ct.de/1405176 c't