(Kopfzeile)

Die Git Revolution

Vom Klonen und Pushen in verteilten Welten

Eine Revolution ist der Gegenbegriff zur Evolution, bei der sich eine Weiterentwicklung vollzieht, während bei einer Revolution immer eine plötzliche Neuerung oder Umkehr von Altbekanntem gemeint ist. Revolutionen finden heutzutage im umgangssprachlichen Sinne meist unter Anwendung von Gewalt statt, wie beispielsweise bei der Ablösung einer politischen Macht. Was für eine Revolution das verteilte Versionskontrollsystem Git ausgelöst hat lesen Sie auf den folgenden Seiten.

von Michael Johann

Wenn Sie in Projekten mit mehr als einem Entwickler gearbeitet haben, ist die Wahrscheinlichkeit relativ groß, dass Sie bereits mit einem Versionskontrollsystem gearbeitet haben. Ich selbst habe im Laufe meiner doch schon etwas langen Karriere so ziemlich alle gängigen Systeme benutzt.

Hierzu zählen:

- Microsoft VSS
- CVS
- Subversion
- Rational ClearCase
- Git
- Mercurial

Weitere Systeme, die das fortgeschrittene Alter eines Software-Veteranen verraten können, sind:

- RCS
- SCCS

Alle Systeme haben ihre Vor- und Nachteile, wobei allerdings die Nachteile bei einigen insbesondere älteren und proprietären Systemen überwiegen.

Die einfache Nachführung von Änderungen an Dateien beherrschen alle Systeme. Lediglich wenn es zu komplizierten Branch- und Merge-Vorgängen kommt, trennt sich spätestens die Spreu vom Weizen.

Grundsätzlich sollten Sie ein Versionskontrollsystem nach Ihrem Bedarf auswählen. Wenn Sie auf nur einem lokalen Rechner und für sich allein arbeiten, reichen Systeme mit weniger Funktionsumfang völlig aus. Features wie verteilte Repositories sind dann eher für Entwickler gedacht, die in größeren Teams mit unterschiedlichen Standorten arbeiten. Zudem unterstützen einige Systeme das Halten von lokalen Repositories, die bei Bedarf in einem zentralen Repository zusammengeführt werden können.

Die besten Versionskontrollsysteme sind auch für alle wichtigen Betriebssysteme verfügbar und oft sogar als Open Source-Projekte angelegt. Die Integration in die bekannten Entwicklungsumgebungen und die Bereitstellung von Kommandozeilenwerkzeugen ist sicherlich auch ein wichtiger Aspekt bei der Auswahl eines geeigneten Systems.

In den letzten Jahren sind einige neue Sterne am Himmel der Versionskontrollsysteme aufgetaucht. Hierzu zählen Subversion, Git und Mercurial. Vielleicht haben Sie bereits Subversion im Einsatz gehabt und kennen sich schon gut mit der Funktionsweise aus. Dann wissen Sie, dass

Subversion, das sich anschickt, der Nachfolger von CVS zu sein, ein zentrales Repository benötigt, in dem alle Versionen verwaltet werden.

In letzter Zeit aber ist eine Art Social Web für Entwickler entstanden, ganz nach Web 2.0 Manier. Da gibt es Plattformen, die unzählige Projekte hosten und im Sinne der Open Source Bewegung Entwickler organisieren und Repositories verwalten. Subversion ist da mit seinem zentralen Ansatz etwas problematisch, denn wer Änderungen einchecken will, muss diese Rechte auch haben und entsprechend verantwortungsvoll handeln.

In einer verteilten Welt sind verteilte Repositories ein echter Vorteil und das ist die eigentliche Revolution hinter Git. Ich habe die Auswirkungen dieser Revolution vor einigen Jahren im Zuge der Nutzung von Ruby on Rails erfahren. Damals war mein Favorit noch Mercurial, das ebenfalls wie Git funktioniert und sehr vielversprechend war. Mit dem Wechsel von Subversion auf Git, hatte Ruby on Rails allerdings eine gewisse Richtung vorgegeben. Heute gibt es mit Github (www.github.com) eine mächtige Plattform für Entwickler, die einfach nur überzeugt.

Pro und Kontra verteilte Versionskontrolle: Vorteile

Sie fragen sich vielleicht, was die Vorteile eines verteilten Repositories sind und warum hier von der Git-Revolution berichtet wird. Die folgenden Vorteile überzeugen schnell von der Sinnhaftigkeit verteilter Repositories:

Höhere Geschwindigkeit

Wenn Sie mit einem zentralisierten Versionskontrollsystem arbeiten, werden alle Änderungen über ein Netzwerk übertragen. Lokale Zugriffe auf ein Repository sind logischerweise wesentlich schneller. Dieser Geschwindigkeitsvorteil ist nicht zu unterschätzen. Wenn Sie beispielsweise ein Subversion-Repository mit vielleicht 1000 Dateien initial auschecken, kann dies bei einer DSL-Verbindung schon einige Minuten dauern. Änderungen an 100 Dateien, die zwischen Client und Server ausgetauscht werden müssen, können je nach Umfang ebenfalls lange Zeit in Anspruch nehmen.

Ein verteiltes Versionskontrollsystem muss nur dann über das Netz kommunizieren, wenn Changesets ausgetauscht werden müssen. Alle anderen Arbeiten und auch das Taggen von Versionsständen können bis dahin lokal erfolgen und sind binnen weniger Sekunden erledigt.

Unabhängigkeit vom Netz

Mit einem verteilten Versionskontrollsystem sind sie bei Arbeiten unterwegs oder an Orten ohne Netzwerkzugriff unabhängig und können trotzdem auf sämtliche Revisionen und die komplette Historie zugreifen. Andere Systeme, die einen zentralisierten Ansatz bieten, sind nicht immer besonders skalierbar, denn der Zugriff von einigen Dutzend Benutzern gleichzeitig kann ein solches System schon mal an seine Leistungsgrenzen führen. Abhilfe schaffen dann Replikationstools, die aber im Grunde nur Geld kosten und das bieten, was Systeme wie Git bereits kostenlos mitbringen.

Merge mit Historie

Subversion ist nicht in der Lage, über die Historie eines Changesets einen Merge vorzunehmen. Benutzer von Subversion müssen jede einzelne Revision eines Branches manuell zusammenführen. Wer dabei den Überblick verliert, endet schnell in einem Chaos aus Konflikten oder fehlenden Teilen.

Wenn Dateien und Verzeichnisse umbenannt wurden, kann Subversion ebenfalls nicht automatisch mergen. Dies ist allerdings auch schon die größte Schwachstelle von Subversion. Der Grund, warum Git hier punktet, ist, dass verteilte Versionskontrollsysteme auf den Prinzipien von Branches und Merges aufbauen.

Pro und Kontra verteilte Versionskontrolle: Nachteile

Es gibt nicht nur Vorteile. Damit Sie ein korrektes Bild von einem verteilten Versionskontrollsystem wie Git erhalten, sollen auch die Nachteile kurz erläutert werden. Diese Nachteile können Sie dann vermeiden.

Keine „aktuellste“ Version

Aufgrund der Peer-To-Peer Architektur existiert keine zentral abrufbare „aktuelleste Version“. Diese lässt sich aber herstellen, wenn alle Lieferanten in ein „zentrales“ Repository pushen. Damit ist ein Repository gemeint, das als zentrales Repository fungiert.

Kein Backup

Auch wenn es ein zentrales Repository gibt, ist es dennoch nicht gleichzeitig ein Backup. Denn wenn Änderungen erst zu spät veröffentlicht werden, ist nicht klar, was damit geschehen soll. Ein zentrales Repository, wie im Abschnitt zuvor beschrieben, hilft hier. RegulärеBackups dieses Systems sollten trotz allem natürlich regelmäßig erfolgen.

Komplexe Versionsnummern

Weil ein verteiltes Versionskontrollsystem keine einheitlichen Versionsnummern über verschiedene Repositories halten kann, gibt es stattdessen eine GUID (Globally unique identifier), die beispielsweise bei Git so aussieht.

```
ee632c122bd8f13eb1e885ff230453bddfa772a1
```

Wenn zwei Kollegen sich also abstimmen und Versionsnummern austauschen, wird es schon eher schwierig, sich an diese lange Nummer zu erinnern. Daher ist es möglich, Tags zu vergeben, die Menschen wieder einfacher verstehen.

Git ist für längere Versionsnummern bekannt, die praktisch niemand mit einem Blick unterscheiden kann. Daher ist eine gute Unterstützung durch Werkzeuge sinnvoll.

Git it on...

Git ist ebenso wie Mercurial und Bazaar ein verteiltes Versionskontrollsystem und ebenfalls als Open Source-Projekt verfügbar. Es wurde 2005 von Linus Torvalds, dem Kopf der Linux-Gemeinde, als Ersatz für ein zuvor genutztes Versionierungssystem entwickelt. Der Linux-Kernel sollte mit diesem System verwaltet und versioniert werden.

Die folgenden Punkte waren die wichtigsten Anforderungen an das zu schaffende System:

- Es muss schnell sein, um die relativ große Menge an Code und die vielen Dateien handzuhaben.
- Es muss eine Möglichkeit geben, Patches zu verwalten, damit nicht beliebige Nutzer das Zielsystem verändern können.
- Das System muss dezentral funktionieren, damit Bandbreiten geschont und eine gewisse Ausfallsicherheit bei Störungen eines zentralen Systems umgangen werden können.

In Punkto Management von Patches bietet Git eine Menge Funktionen mehr als die Konkurrenz. So kann beispielsweise ein Email-System zur Verwaltung von Patches genutzt werden.

Git ist inzwischen in allen gängigen Linux-Distributionen verfügbar und unterstützt aktuell auch Mac OS X und Windows. Im Folgenden erfahren Sie, wie Sie mit Git ein Repository erstellen und die wichtigsten Werkzeuge und Workflows nutzen, um Projekt-Sourcen mit Git zu verwalten.

Doch zunächst einige Worte zur Installation auf den wichtigsten Plattformen.

Wie bereits erwähnt, gibt es Git für nahezu alle wichtigen Betriebssysteme. Um die Installation von Git zu erläutern, werden die Installationen unter Mac OS X, Ubuntu Linux und Windows 7/Vista gezeigt.

Wie so oft gibt es bei Open Source-Projekten die Möglichkeit, Git mit einem Installationsprogramm oder manuell zu installieren. Für Mac OS X wird die einfache Variante mittels eines grafischen Installationsassistenten gezeigt. Unter Linux finden Sie ein Paket für den jeweiligen Paketmanager, und für Windows ergibt sich eine spezielle Variante durch die Nutzung der Cygwin-Shell.

Das Git-System für Apples Betriebssystem finden Sie unter http://code.google.com/p/git-osx-installer/ und kann durch einen einfachen Installationsprozess ins System integriert werden. Die erfolgreiche Installation können Sie in einem Terminal überprüfen, indem Sie den folgenden Befehl eingeben.

```
$ git --version
git version 1.7.3.1
```

Zum Zeitpunkt der Entstehung dieser Seiten war die Version 1.7.3.1 aktuell. Mit der erfolgreichen Installation steht Ihnen Git nun in vollem Funktionsumfang zur Verfügung.

Unter Linux ist die Installation ebenfalls kein großer Aufwand. Ubuntu beispielsweise bietet eine einfache Installation mittels *apt-get* an.

Geben Sie auf der Kommandozeile einfach folgende Anweisung ein:

```
$ sudo apt-get install git-core git-gui
```

Die entsprechenden Pakete werden installiert, und danach steht Ihnen Git unter Ubuntu zur Verfügung.

Die Installation von Git unter Windows stellte anfangs eine besondere Herausforderung dar. Es gab bei der Umstellung des Ruby on Rails-Repositories auf Git einige Diskussionen über die Unterstützung von Git für Windows. Allerdings beruhigten sich die Gemüter recht bald, da mit Cygwin unter Windows eine Unix-Shell zur Verfügung steht, die Git bestens unterstützt.

Eine weitere Alternative, die eine Git-Bash und ein GUI bereitstellt, ist MSYSGIT. Sie finden die jeweils aktuelle Version unter:

```
http://code.google.com/p/msysgit
```

Die Installation ist mit einem Assistenten verbunden und in wenigen Minuten erledigt. Danach verfügen Sie über eine vollständige Git-Installation, mit der Sie wie unter einem Unix-System arbeiten können.

Git-Aktionen können mit dem Hauptprogramm *git* durchgeführt werden.

Wenn Sie zu einzelnen Aktionen Hilfe benötigen, liefert Git mit den Manpages die nötigen Informationen oder sie erhalten über die Kommandozeile ein paar Anhaltspunkte, wenn sie lediglich git aufrufen.

Konfiguration

Bevor Sie mit Git arbeiten, sollten Sie einige globale Einstellungen vornehmen, die Ihnen die Arbeit mit Git vereinfachen. Dazu gehören die Konfiguration der Nutzerinformationen und die farbliche Hervorhebung von Statusinformationen oder Informationen zu Diffs etc.

Git kennt die folgendermaßen verwendete Aktion *config*.

```
$ git config --global user.name "Michael Johann"
$ git config --global user.email "mjohann@rails-experts.com"
```

Mit den beiden Konfigurationsschritten wird die globale Datei *.gitconfig* im Home-Verzeichnis des aktuellen Benutzers angelegt.

Weitere Angaben zur automatischen Farbgebung können Sie mit den folgenden Aufrufen machen.

```
$ git config --global color.status auto
$ git config --global color.branch auto
```

Diese Angaben werden ebenfalls in der Datei *.gitconfig* abgelegt, so dass diese nun folgendermaßen aussieht.

```
[user]
   name = Michael Johann
   email = mjohann@rails-experts.com
[color]
   status = auto
   branch = auto
```

Es gibt eine Vielzahl weiterer Konfigurationsmöglichkeiten, die allerdings unseren inhaltlichen Rahmen sprengen würden. Sollten Sie konkreten Informationsbedarf zu diesen Konfigurationsparametern haben, empfehle ich Ihnen die Manpages zu Git.

Repositories erstellen

Nach ersten Informationen und Arbeitsschritten zur Konfiguration lernen Sie nun, wie ein Repository mit Git angelegt wird. Ein Repository wird mit der Aktion *init* erstellt. Dabei ist das aktuelle Verzeichnis gleichzeitig das Verzeichnis für die Git-Objekt-Datenbank mit dem Namen *.git*.

Da wir aber später dieses lokale Repository klonen wollen, müssen wir es zu einem Repository ohne Arbeitsverzeichnis machen. Dabei handelt es sich um ein sogenannten Bare-Repository.

Wir werden jetzt ein Projektverzeichnis und ein paar Dateien anlegen, um die typischen Workflows kennen zu lernen. Hierzu erstellen Sie einfach ein beliebiges Verzeichnis:

```
$ mkdir testapp.git
```

Dann wechseln Sie in das soeben erstellte Verzeichnis. Um nun ein tyisches Git-Repository zu erstellen, rufen Sie lediglich folgende Aktion auf:

```
$ git init --bare
Initialized empty Git repository in /Users/mjohann/projects/testapp.git/
```

Das war schon alles. Kein Aufsetzen eines Servers oder irgendwelche Zeremonien für die Administration. Ausgehend von diesem neuen Repository können nun die Arbeiten an unserem Projekt beginnen.

Angriff der Klone

Jetzt haben wir also ein leeres Repository, welches als Quasi-Zentrales Repository fungieren soll. Um nun die Arbeit zweier Entwickler zu simulieren, müssen wir diesem anderen Entwickler unser Repository bereitstellen. Mit Subversion würden wir lediglich den Zugriff auf das Repository gewähren und der Entwickler könnte sich eine Arbeitskopie ziehen. Aber das war es dann auch schon. Die komplette Historie liegt dann nicht lokal vor.

Git erlaubt es, durch das sogenannte Klonen, die Replizierung eines kompletten Repositories inklusive der kompletten Historie, so dass bei Bedarf und bei Mangel einer Netzwerkverbindung sämtliche Informationen aus der Vergangenheit auch lokal verfügbar sind.

Schauen wir uns diesen Vorgang etwas genauer an, indem wir das lokale Repository (testapp.git) nun klonen:

```
$ cd ..
$ git clone testapp.git testapp-one
Cloning into testapp-one...
done.
```

```
warning: You appear to have cloned an empty repository.
```

Das war es schon. Wenn wir nun in das Verzeichnis testapp-one wechseln, sehen wir eine Arbeitskopie und im .git-Verzeichnis das Repository (Finger weg vom .git-Verzeichnis ;)). Die Arbeitskopie enthält allerdings noch keine Dateien oder Verzeichnisse, da wir ein leeres Repository geklont haben.

Jederzeit kann Git uns über den Status des Arbeitsverzeichnisses informieren. Hierzu dient die Aktion *status*:

```
$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

Wie Sie sehen, sehen Sie „Nichts“, lediglich weist uns Git darauf hin, dass wird mit *git add* neue Dateien in die Versionsverwaltung aufnehmen können. Da bis zum jetzigen Zeitpunkt aber noch keine Dateien existieren, legen wir nun einfach einmal eine Datei *README* an und erfragen nochmals den aktuellen Status.

```
$ touch README
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  README
nothing added to commit but untracked files present (use "git add" to track)
```

Jetzt erklärt Git, dass eine neue Datei namens README existiert, diese aber nicht versionsgeführt ist.

Fügen wir also die neue Datei dem Repository hinzu:

```
$ git add .
```

Damit werden alle Dateien und Verzeichnisse des aktuellen Verzeichnisses in die Versionskontrolle überführt. Wollten wir nun einige Dateien hinzufügen, hätten wir diese Dateinamen entsprechend verwenden müssen. Kontrollieren wir nun wieder den Status, erhalten wird den Hinweis, dass eine neue Datei hinzugefügt wurde:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   README
#
```

Diese Änderung ist allerdings noch nicht endgültig bestätigt worden. Es fehlt, wie bei anderen Versionssystemen auch, ein „commit“. Das holen wir direkt nach:

```
$ git commit –m „Initialer Commit“
[master (root-commit) d98693d] Initialer Commit
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README
```

Wir haben jetzt ein Projektverzeichnis inklusive einer Arbeitskopie und eines Git-Repositories, in welches wir eine Datei versionieren.

Änderungen verfolgen

In unserer REAME Datei ist noch kein Text vorhanden, was eine gute Grundlage für eine Änderung ist. Denn schließlich ist das der Grund für den Einsatz einer Versionskontrolle. Geben Sie also einfach ein paar Zeilen Text in die Datei und lassen Sie uns schauen, wie uns Git bei der Versionierung hilft.
Meine Änderungen sind:

```
Dies ist ein Beispielprojekt für Git.
Viele Grüße an die Leser des JavaMagazins.
```

Wenn wir nun Git nach dem Status fragen, erhalten wir folgende Hinweise:

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   README
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Git zeigt uns an, dass eine Änderung an der Datei README erfolgt ist und dass noch kein Commit erfolgt ist.
Welche Änderungen dies sind, kann und Git auch verraten, wenn wir einfach mit folgender Aktion nach den Differenzen fragen:

```
$ git diff
diff --git a/README b/README
index e69de29..264ecea 100644
--- a/README
+++ b/README
@@ -0,0 +1,2 @@
+Dies ist ein Beispielprojekt für Git.
+Viele Grüße an die Leser des JavaMagazins.
\ No newline at end of file
```

Das ist doch schon sehr ausführlich, nur gut, dass wir aktuell nicht kiloweise Änderungen im Projekt haben, sonst wäre die Ausgabe entsprechend lang. Mit diesen Informationen können wir nun die Änderungen Committen.

```
$ git commit -am "Some text changes"
[master f070d15] Some text changes
```

```
 1 files changed, 2 insertions(+), 0 deletions(-)
```

Der Parameter ‚a' sollte eigentlich immer verwendet werden. Er sorgt dafür, dass sämtliche Änderungen (auch gelöschte Dateien, etc.) an bekannten Dateien vor dem Commit ins Changeset übernommen werden.

Push it to the limit

Mit den nun gemachten Änderungen können wir uns zufrieden zurücklehnen und das Ganze ins Originale Repository pushen. Bei einem Push werden sämtliche Changesets und die komplette Historie des lokalen Klons gesammelt, komprimiert und in den sogenannten *origin master* branch kopiert. Dies geschieht mit folgender Eingabe:

```
$ git push origin master
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), 524 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
To /Users/mjohann/projects/testapp.git
 * [new branch]      master -> master
```

Wenn wir nun einen weiteren Entwickler ins Team bekommen, kann dieser sich wieder einen Klon besorgen und darauf die im nächsten Abschnitt gezeigten Änderungen vornehmen.

Geben Sie also folgenden Befehl ein:

```
$ cd ..
$ git clone testapp.git testapp-two
Cloning into testapp-two...
done.
```

Wenn Sie nun in das Verzeichnis testapp-two wechseln und dort *git log* eingeben, sehen Sie die komplette Änderungshistorie:

```
Author: Michael Johann <mjohann@rails-experts.com>
Date:   Mon Nov 1 11:21:18 2010 +0100

    Some text changes

commit d98693db8d3494a29d3f135d90a5b8a7460c6845
Author: Michael Johann <mjohann@rails-experts.com>
Date:   Mon Nov 1 11:19:14 2010 +0100

    Initialer Commit
```

Ja, ich weiß, das ist alles an einem Feiertag passiert, aber ich war grad im Urlaub und konnte den Artikel nur dort abschließend schreiben.

Guten Tag

Die Handhabung von Tags in Git ist sehr einfach. Sie nutzen das Kommando *git tag* zur Vergabe von Tags. Mit dem aktuellen Revisionsstand können wir auf der Kommandozeile Folgendes eingeben, um das Tag *V0.1* zu vergeben.

```
$ git tag -a V0.1 -m "First really good version"
```

Die Liste aller Tags können Sie mit git tag ohne Angabe von Parametern abfragen.

Das war doch einfach und schmerzfrei, oder?!

Wir haben jetzt vor den nun folgenden Änderungen am Projekt ein Tag vergeben, das sich auf einen bestimmten Stand im Repository bezieht. Entwickler A, der das originale Repository bereitgestellt hat und auf Klon *testapp-one* arbeitet, kennt dieses Tag noch nicht und wird auch erstmal nichts davon erfahren. Entwickler B macht jetzt einige Änderungen und baut schonmal fleißig an Version V0.2 weiter.

Hierzu können wir einfach ein paar Verzeichnisse und Dateien anlegen:

```
$ mkdir –p src/com/tectalic/util
$ vim src/com/tectalic/util/Echo.java
```

Der Inhalt der Klasse Echo ist der Renner in der Entwicklung und sieht so aus:

```
package com.tectalic.util;

public class Echo {
   public static String echo(String input) {
      return input;
   }
}
```

Zugegebenermaßen nicht wirklich aufregend, aber hier geht es um Git und nicht um Java.

Eine weitere Abfrage des Status mit Git zeigt folgende Ausgabe:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  src/
nothing added to commit but untracked files present (use "git add" to track)
```

Aha, wir bekommen auch noch mitgeteilt, dass wir ein paar Dinge mit *git add* hinzufügen müssen, wenn wir eine Versionierung wünschen. Nichts leichter als das:

```
$ git add .
$ git commit -am "New Echo Impl"
[master f672261] New Echo Impl
 1 files changed, 7 insertions(+), 0 deletions(-)
 create mode 100644 src/com/tectalic/util/Echo.java
```

Wenn Sie möchten, können Sie diesen Stand nun mit einem Tag versehen und pushen. Entwickler A kann dann von Ihnen aufgefordert werden, die gemachten Änderungen mit *git pull* zu übernehmen.

Der übliche Zyklus bewegt sich also zwichen clone, push und pull. Es gibt noch unzählige weitere Möglichkeiten, die aus Platzgründen hier nicht erklärt werden können. Die vorherigen Erläuterungen sollten auch lediglich einen kurzen Appetizer auf Git darstellen. Wenn Sie zuvor mit Subversion gearbeitet haben, wird Ihnen sicher das eine oder andere aufgefallen sein, was in Git anders ist. Der folgende Abschnitt beschreibt nun den Bereich der sich komplett von Systemen wie Subversion unterscheidet.

Die eigentliche Revolution

Soweit sollten die Erklärungen zu den Grundlagen von Git reichen. Bisher war das alles eher auch schon aus Subversion bekannt und nichts hat die Revolution sichtbar gemacht. Gut, anstelle eines zentralen Repositories wird das ganze verteilt. Wahnsinn, aber wo ist die eigentliche Umkehr vom Bekannten? Das Stichwort heist hier: Branches.

Wenn Sie bereits Bücher oder Artikel zu Subversion gelesen haben, wird Ihnen aufgefallen sein, dass auch dort die Anlage von Repositories und das Aus- und Einchecken von Dateien und Verzeichnissen als erstes erklärt wird. Erst später, wenn der Autor glaubt, dass der Leser alles bisherige verstanden hat, geht es ans Eingemachte: Das Branching. Braucht man ja nicht so oft, glauben Sie vielleicht. Ist auch viel zu kompliziert, denkt da der ein oder andere. Irgendwie hat ja jeder Recht, aber mit Git rückt das Branching in den Bereich der Grundlagen und wird fast zum Kinderspiel, weil Git auf diesen Prinzipien aufbaut. Also, schauen wir uns einmal genauer an, wie die typischen Workflows mit Git aussehen können. Workflows eben, in denen oft und gerne gebrancht und gemerget wird ;)

Grundsätzlich kennen Entwickler die folgenden Bereiche:

Es gibt einen Releasebereich, in dem getaggte Release eines Projektes liegen.

Dann gibt es einen Branch für die Entwicklung (development) und einen für die Hotfixes, falls wider erwarten doch noch Fehler in der Software entdeckt werden.

Wenn man im Team auch noch einen Versionmanager-Gott hat, kann man auch noch Branches für Features einführen, aber was ist, wenn der Profi mal im Urlaub ist? Scherz beiseite, sehen wir uns das ganze einmal als Grafik an.

Folie-12.png

Eine Sammlung typischer Branches und Worfklows entlang der Zeitachse

Da ist zu erkennen, dass insgesamt vier Branches im Repository über die Zeit geführt werden.

Branch	Zweck
Master	Der Hauptbranch für die Release
Bugfixes	Hotfixes werden hier bearbeitet
Development	Hauptentwicklungsstrang
Feature-1...Feature-n	Features werden hier zuerst angelegt
Release-Branch	Optional (Nicht im Diagramm) für minimale Fixes

Abbildung 1 (Folie-12.png) zeigt das fertige Schaubild mit allen Branches, die wir sinnvollerweise gebrauchen können. Um zu verstehen, wie alles zusammenhängt und wie wir das Ganze mit Git im Alltag nutzen können, sollen die nächsten Seiten den Aufbau erklären.

Branch: Master

Auch Master-Branch genannt, werden hier die Hauptrelease gepflegt, die an unsere Kunden ausgeliefert werden. In den Beispielen zuvor haben wir den Master-Branch schon kennengelernt. In Git ist der Standard-Branch immer Master und wir müssen diesen nicht explizit anlegen. In Abbildung 1 ist der Master-Branch entsprechend über der Zeitleiste zu sehen und enthält die Tags V0.1, V0.2 etc. Hier werden auch die Änderungen aus den anderen Branches eingepflegt.

Wenn Sie auf einem bestimmten Branch arbeiten wollen, können Sie dies mit folgender Aktion erledigen:

```
$ git checkout <branch_name>
```

Für den Master-Branch wäre das dann konkret so:

```
$ git checkout master
```

Branch: Bugfixes

Für den Fall, dass ein Kunde einen Fehler in unserer Software gefunden hat, sollte ein Branch für Hotfixe bereitsstehen. Dieser Branch kann, wie in Abbildung 1 zu sehen ist, direkt vom Master-Branch abgeleitet werden. Auf der Kommandozeile sieht das dann so aus:

```
$ git checkout –b bugfixes master
```

Wenn Sie die Liste aller Branches sehen möchten, können Sie folgende Aktion ausführen:

```
$ git branch
* bugfixes
  master
```

Der Stern vor dem Branchnamen gibt an, auf welchem Branch Sie sich aktuell befinden. Ausgehend von der Grafik in Abbildung 1 können wir nun einen Hotfix vornehmen, commiten und dann auf den Master-Branch mergen. Um diesen Vorgang nachzuvollziehen, ändern Sie bitte einfach ein paar Daten an der Klasse *Echo*.

```
package com.tectalic.util;

public class Echo {
    public static String echo(String input) {
      System.out.println("Diese Software ist fehlerfrei");
      return input;
    }
}
```

Dann werden die Änderungen commitet:

```
$ git commit –am „Bug fixed“
[bugfixes c744ed1] Bug fixed
 1 files changed, 2 insertions(+), 1 deletions(-)
```

Laut Pfeil in Abbildung 1, wird der Bugfix nach Master gemerget und mit V0.2 getaggt und dann in das zentrale Repository gepush:

```
$ git checkout master
Switched to branch 'master'
$ git merge bugfixes master
Fast-forwarding to: bugfixes
Already up-to-date with master
Merge made by octopus.
 src/com/tectalic/util/Echo.java |    3 ++-
 1 files changed, 2 insertions(+), 1 deletions(-)
```

Sie sehen also, die Änderungen sind im Master-Branch angekommen.

Branch: Development

Die allgemeinen Entwicklungsarbeiten können immer und gerne auf einem Development-Branch beispielsweise mit dem Namen development durchgeführt werden. Auch hier ist in Abbildung 1 zu sehen, dass der Ursprung vom Branch Master kommt.

Entsprechend können Sie Entwickler A oder B mit dessen Erstellung beauftragen. Da bisher alle Änderungen auf dem Klon testapp-two erfolgt sind und diese Änderungen noch nicht ins zentrale Repository gepusht wurden, kann Entwickler A auf seinem Klon einfach den Development-Branch erstellen:

```
$ git checkout –b development master
```

```
Switched to a new branch 'development'
```

Es empfiehlt sich natürlich immer, von Zeit zu Zeit die aktuellen Inhalte vom zentralen Repository zu holen (pull):

```
$ git pull origin master
From /Users/mjohann/projects/testapp
 * branch            master     -> FETCH_HEAD
Updating f070d15..f672261
Fast-forward
 src/com/tectalic/util/Echo.java |    7 +++++++
 1 files changed, 7 insertions(+), 0 deletions(-)
 create mode 100644 src/com/tectalic/util/Echo.java
```

Beachten Sie auch, dass der Bugfix noch nicht vorhanden ist, weil Entwickler B diese Änderungen noch nicht gepusht hat. Der Entwickler A kann nun ein paar Änderungen vornehmen oder aber auch gleich für einzelne Features entsprechend die bereits erwähnten Feature-Branches anlegen:

```
$ git checkout –b feature-login development
Switched to a new branch 'feature-login'
```

Fügen wir abschließend eine neue Klasse namens Login hinzu, um die ganzen Branches anschließend im zentralen Repository zusammen zu führen:

```
$ mkdir src/com/tectalic/auth
$ vim src/com/tectalic/auth/Login.java
package com.tectalic.auth;


public class Login {}
```

Dann sollten Sie wie zuvor gelernt, die neue Datei mit *git add* hinzufügen und mit *git commit* bestätigen.

Anschließend soll das „neue Feature“ in den Development-Stream überführt werden:

$ git checkout development

$ git merge feature-login development

Abschlussarbeiten

Jetzt gibt es ein neues Feature und einen Bugfix. Im zentralen Repository ist davon aber noch nichts angekommen. Also pushen Entwickler A und B ihre Änderungen und wir beginnen mit den Abschlussarbeiten für Release V0.3:

```
$ git push origin development
Counting objects: 12, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (8/8), 655 bytes, done.
Total 8 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
To /Users/mjohann/projects/testapp.git
 * [new branch]      development -> development
```

Damit ist der development-Branch endlich auch „zentral“ verfügbar.

Nun folgt Entwickler B:

```
$ git push origin master
Counting objects: 14, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (8/8), 693 bytes, done.
Total 8 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
To /Users/mjohann/projects/testapp.git
   f672261..6ae670d  master -> master
```

Entwickler B soll die neue Version bereitstellen (Inklusive Bugfix und Login-Feature). Dazu muss er erstmal sein lokales Repository updaten (pull):

```
$ git pull
remote: Counting objects: 12, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 8 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
From /Users/mjohann/projects/testapp
 * [new branch]      development -> origin/development
Already up-to-date.
```

Aha, hier ist also ein neuer Branch namens development angekommen.

Dann wollen wir mal mergen:

$ git checkout master (Der Übersicht halber)

$ git merge development master

Already up-to-date with master

Merge made by octopus.

src/com/tectalic/auth/Login.java | 3 +++

1 files changed, 3 insertions(+), 0 deletions(-)

create mode 100644 src/com/tectalic/auth/Login.java

Hier ist also das neue Feature angekommen. Der Bugfix liegt eh schon auf master, so dass Entwickler B nun die Version taggen kann:

```
$ git tag –a V0.3 –m „Bugfixes and Loginfeature“
```

Mit *git log* können Sie sich nochmal die gesamte Historie ausgeben lassen.

Fazit

Mit Git ist besonders der Umgang mit Branches und Merges ein Kinderspiel. Der Artikel konnte lediglich an der Oberfläche kratzen und hat keinesfalls einen vollständigen Überblick zum verteilten Versionskontrollsystem Git gegeben. Wer beispielsweise unter Windows arbeiten muss, kann mit Tortoise-Git entsprechend komfortabel vorgehen wie mit Subversion. Auch kennt Git eine Möglichkeit Subversion Repositories nach Git zu migrieren. Die weiteren Schritte sollten nun die Installation von Git und das erstellen eines Accounts bei www.github.com sein. Manchmal haben Nutzer von Subversion Verständnisprobleme, wenn Sie nach Git wechseln wollen. In diesem Fall hilft es, sich komplett gedanklich von Subversion zu befreien und sich ernsthaft mit den Grundlagen von Git zu befassen. Eine Suche bei Ihrem Suchmaschinenbetreiber listet zudem eine große Auswahl an Tutorials und Informationen auf, um Ihnen den Umstieg weiter zu vereinfachen.

Links & Literatur

[1] www.github.com - Führender Anbieter für Git-Repositories
[2] www.git.cz - Heimat von Git

Autor
Michael Johann ist Chefredakteur des Magazins RailsWay (www.railsway.de) und Autor des Buches "Ruby on Rails für JEE-Experten", (Hanser Verlag). Er ist Berater und Trainer für JRuby on Rails und regulärer Sprecher auf Konferenzen rund um den Globus. Vor dem Switch zu Ruby on Rails wurde er als JEE-Experte und als Chefredakteur von Java-Spektrum bekannt.