Git Workflow

Einführung

Als einen der ersten Schritte in unserem Entwicklungsplan stand die Einführung eines Versionskontrollsystems. Versionskontrolle ist heutzutage Gegenstand beinahe jedes Entwicklungsprozesses. Die beiden großen Vorteile eines solchen Systems sind hier „Rollback Fähigkeit“ und verteilte Entwicklung. Wir haben uns für GIT als Versionskontrollsystem entschieden. GIT ist „Industry Standard“ und wird sowohl in kleineren Unternehmen als auch in sehr großen die Welt umspannenden OpenSource Projekten eingesetzt.

Um GIT effektiv zu nutzen empfiehlt sich die Nutzung eines dedizierten Arbeitsprozesses, eines sogenannten „Workflows“. In der GIT-Welt haben sich hier verschiedene Arbeitsweisen zu Standards entwickelt und es empfiehlt sich hier auf die Erfahrung anderer zurückzugreifen, anstatt das Rad neu erfinden zu wollen. Was für die Entwickler des Linux Kernel‘s oder Google gut ist, kann für „Mousenect“ so schlecht nicht sein.

Unser Team besteht aus vier Entwicklern. Zwei von uns hatten schon Berührungspunkte zu Programmen der Versionskontrolle und einer explizit mit GIT. Der Schwerpunkt unseres Projekts liegt klar in der Entwicklung der Software und nicht in der Ausarbeitung der Funktionen von GIT. Wir benötigen also einen einfach zu befolgenden, effizienten, klaren „Workflow“ mit möglichst wenigen Fallstricken.

Beschreibung „Centralized Workflow“

Wir beginnen mit einem zentralen Repository als Einstiegspunkt für alle Änderungen am Projekt. „Master“ ist der default Development Branch und ein „Stable“ Branch enthält die gegenwärtig funktionale Version unseres Produkts.

Alle Entwickler „pullen“ von diesem zentralen Repository und können dann lokal und unabhängig voneinander an den Projektdateien arbeiten. „Staging“ und auch das „committen“ von Änderungen findet also nur lokal statt und erst wenn ein gewünschter Zustand erreicht ist, werden sie zum Server „gepusht“. Der gesamte Entwicklungsprozess soll dabei im „Master“ Branch stattfinden. Im Master-Branch wird also getestet, ausprobiert und beraten. Der Master-Branch ist die Sandbox der Software. Zusätzlich zum Master-Branch werden wir noch einen „Stable“ nutzen, um jederzeit über eine saubere Codebasis zu verfügen. Im Verlauf der Entwicklung wird „Stable“ immer eine lauffähige Version der Software enthalten. Mehr Branches werden für diesen Workflow nicht benötigt.

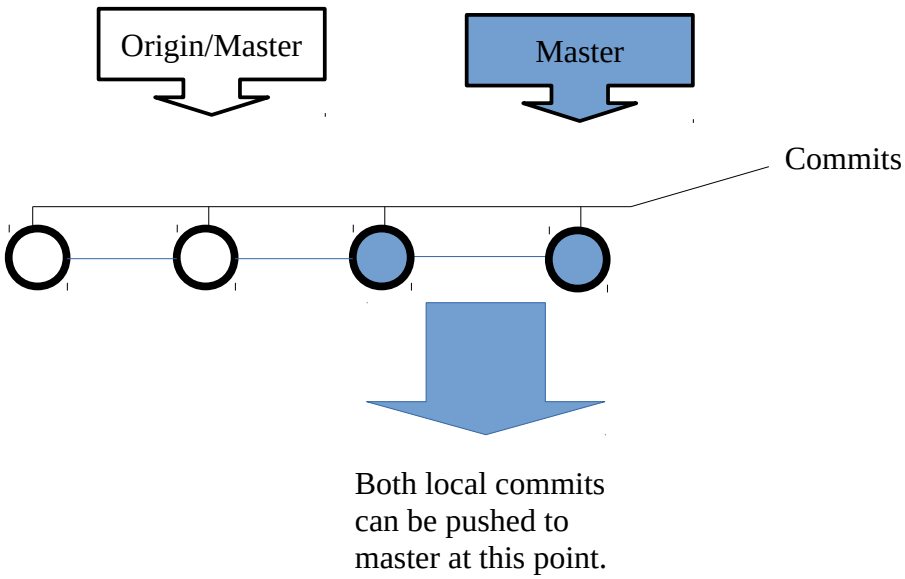
Was ist denn nun aber eigentlich ein Branch? Nun, mit dem ersten Commit speichert GIT ein Abbild des Arbeitsverzeichnisses. Mit jedem weiteren Commit werden dann nur noch die Metadaten jeder weiteren Änderung gespeichert. Die Funktion des „Branching“ erlaubt nun beliebig viele Abbilder des aktuellen Arbeitsverzeichnisses zu speichern. So können verschiedene Entwicklungszustände bearbeitet, gespeichert und wieder aufgerufen werden.

Praktischer Verlauf

Die Codebasis wird mit einem initialen Commit zum Server-Repository hinzugefügt und dann von jedem Entwickler geklont. Alle Entwickler verfügen außerdem über Schreibrechte auf dem Server-Repository um ihre lokalen Änderungen hinzufügen zu können. Sogenannte „pull-requests“ werden nicht benötigt. Gearbeitet wird hauptsächlich im Master-Branch. In den Stable-Branch wird nur nach einer Beratung im Team gemerged oder gepusht. Ob Master oder Stable, gepusht werden nur funktionsfähige Software Versionen. Es wäre ziemlich sinnfrei den anderen Entwicklern defekten Code zum Arbeiten zu geben. Eigener Code wird also nur getestet zum Server gepusht.

Beispiel:

Entwickler Horsti klont das Master Repository und beginnt seine Magie auf den Entwicklungsprozess wirken zu lassen. Horsti sollte dabei möglichst kleinteilige Commits erzeugen, anstatt einen ganzen Tag Arbeit in einen einzigen Commit zu packen. Das macht die Commit-Historie nachvollziehbarer für andere und das zurückrollen/wiederherstellen funktionsfähiger Softwareteile einfacher.
Nachdem Horsti seine Arbeit beendet hat, ist er nun bereit seine Daten auf den Server zu übertragen.



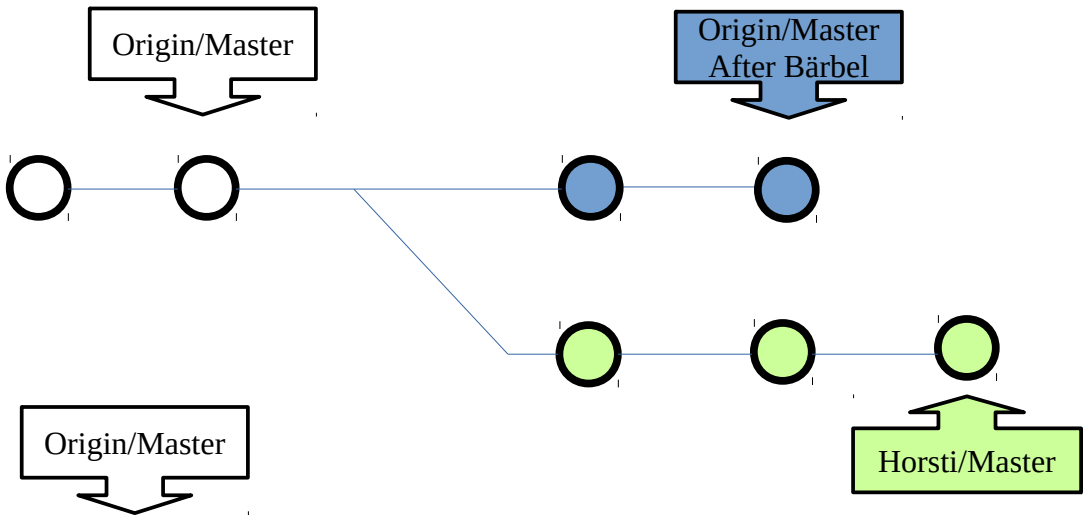
Das obige Beispiel zeigt den Idealzustand für den Entwickler. Der Master-Branch auf dem Server hat sich noch nicht verändert und Horsti ist daher nun in der Lage seine lokalen Änderungen direkt auf den Server zu pushen. „Origin/Master“ steht hier und im folgenden Text für das zentrale Server-Repository, währen „Master“ für sich genommen immer auf den individuellen lokalen Master zeigt.

Eine wichtige Regel lässt sich hieraus schon ableiten. **Pulle immer von Origin/Master bevor Du mit der Arbeit beginnst.** In distributiven Entwicklungsumgebungen ist der kluge und vorausschauende Umgang mit Konflikten zwischen verschiedenen Entwicklungsversionen von sehr wichtiger Bedeutung. Unterscheidet sich also Origin/Master von Master und der Entwickler versucht Änderungen zu pushen, so rennt er in einen sog. „Merge-Conflict“.
Durch stetiges „pull-before-edit“ lassen sich viele Konflikte von vornherein vermeiden und wir verwenden weniger Zeit mit dem lösen dieser Konflikte.

Merge – Konflikte

Der vorherige Idealzustand ist leider nicht der Normalzustand. Sehen wir uns folgendes Beispiel an. Horsti und Bärbel arbeiten am selben Projekt. Beide haben gerade von Origin/Master gepulled und sind somit auf dem selben Stand. Beide beginnen nun unabhängig von einander, verschiedene Teile des Projektes weiter zu entwickeln.

Als erstes ist Bärbel fertig und möchte ihre Änderungen nun zu Origin/Master pushen. Kein Problem. Da Horsti noch nicht gepusht hat, befindet sich Origin/Master immer noch im Ausgangszustand und Bärbel kann wie gewohnt pushen. Horsti ist nun auch endlich fertig und möchte nun seinerseits pushen. Leider antwortet ihm der Server mit einer Fehlermeldung, da Origin/Master sich nun nicht mehr im Ausgangszustand, sondern auf dem Stand von Bärbels Master befindet.



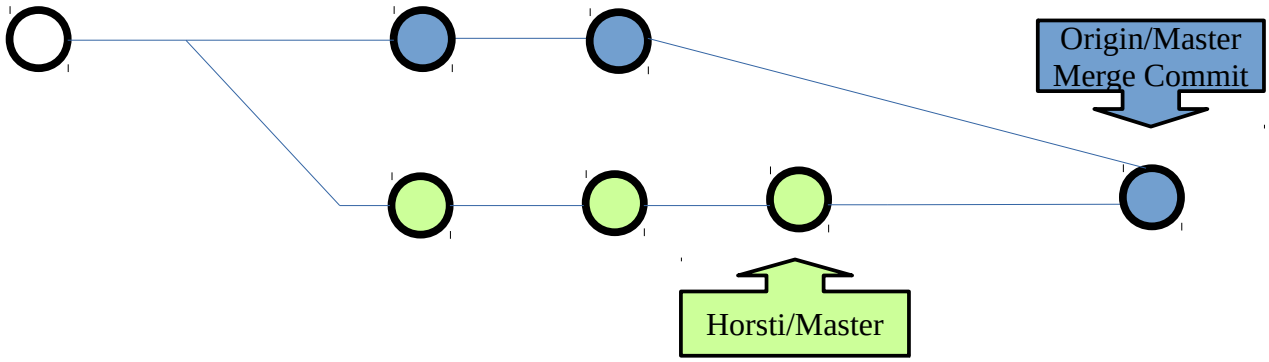
Der Push von Horsti verursacht eine solche Fehlermeldung:

```
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'D:/repos/origin/'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository
pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
details.
```

Was also tun? Origin/Master und Master müssen „gemerget“ also zusammengeführt werden. Hier bieten sich zwei verschiedene Methoden an. Das sogenannte „Rebasing“ und der „Merge“. Der „Merge“ ist eine „Non-Destructive“ Methode. Das heißt keine der bereits existierenden Branches wird in irgendeiner Art und Weise verändert und das Ergebnis wird in einem einzigen „Merge-Commit“ zur Git-Historie hinzugefügt. Diese Methode ist von beiden die sicherere und wird in diesem Workflow verwandt.

Rebasing setzt die lokalen commits auf die Spitze von Master und führt so zu einer klaren, linearen Commit-History. Das ist in vielerlei Hinsicht wünschenswert, kann in öffentlichen Repositories aber auch zu fatalen Problemen führen. Da beim Rebasing die Commit-Historie umgeschrieben und dies ja nur im lokalen Repository geschieht, könnte GIT von da an die Zusammenarbeit mit Origin/Master verweigern mit dem Hinweis darauf, dass die Commit-History zwichen Master und Origin/Master divergiert. Dies führt zur letzten goldenen Regel **„Rebase niemals einen öffentlichen Branch!“**.

„Merging“ unterscheidet sich nicht massiv vom „Pullen“. Zunächst wird schlicht ein „Pull“ ausgeführt. Falls Origin/Master Änderungen enthält, welche wir lokal nicht besitzen, erhalten wir einen Hinweis mit Informationen über Unterschiede zwischen beiden Repositories. GIT hat die Änderungen in den unterschiedlichen Dateien markiert und wir können mit dem editieren beginnen. Nach dem alle Konflikte bereinigt sind und alle Dateien der „Staging-Area“ hinzugefügt wurden, kann nun mit einem Commit der Merge abgeschlossen und Master gepusht werden. Das ganze sollte dann so aussehen:



FAZIT:

Mit diesem Workflow sind wir in der Lage unser Projekt zu Entwickeln ohne dabei zu viel Zeit mit dem Erlernen von GIT oder dem ständigen Austausch von USB Sticks zu verschwenden. Um das Repository abzusichern ist es durchaus sinnvoll ein Gruppenmitglied als GIT-Administrator zu bestimmen. Derjenige ist dann für die Schulung der Gruppenmitglieder verantwortlich und arbeitet sich intensiv in die Nutzung von GIT ein. Es obligt ihm zu entscheiden, wann welche Aktion innerhalb von GIT sinnvoll ist und gibt Hilfestellung bei Problemen. Zwingend nötig ist eine solche Position allerdings nicht. Ich würde es als „Empfehlenswert“ bezeichnen.

Beachte immer die „Goldenen Regeln“:

- Pushe nur lauffähige Software
- Pulle immer von Origin/Master bevor Du mit der Arbeit beginnst
- Rebase niemals einen öffentlichen Branch!

Werden diese Regeln befolgt, solltet ihr den meisten Fallstricken aus dem Wege gehen können.

Danke für die Aufmerksamkeit
Torben Siebke – GIT Administrator / Mousenect