FSST

Eine kleine Einführung in

Git & Github bzw. Gitlab für Li(E)clipse

Für die vierte Schulstufe an einer HTL

Version 0.1

April 2018
HTL Anichstraße

Inhaltsverzeichnis

1 Was ist Git?

1.1 Grundlegende Idee

Git ist ein freies Versionsverwaltungssystem. Es wurde von Linus Torvalds ins Leben gerufen um den Linux Kernel-Entwicklern ein kostenloses Tool für die Versionsverwaltung zur Verfügung zu stellen.
Ein Versionsverwaltungssystem dient dazu die eigenen Änderungen zu überwachen, sie rückgängig zu machen, sie anderen über sogenannte "Repositories"(Repos) zur Verfügung zu stellen oder Aktualisierungen von anderen einzuholen. Git besitzt kein zentrales Repository sondern tritt als verteiltes System auf - aus der Sicht eines Bearbeiters ist die eigene Arbeitskopie (Working Copy) ein eigenes Repo für sich. Bei Bedarf können Änderungen von öffentlich zugänglichen Repos in die eigene Working Copy eingespielt werden oder die eigenen Änderungen z.B. über einen Patch den anderen zugänglich gemacht werden, dies bringt folgende Vorteile:

- Die eigene Arbeit kann einfach wieder in die zentrale Basis integriert werden.
- Es kann zeitgleich weiterentwickelt werden, z.B. jeder an verschiedenen Features.
- Die Versionierung verhindert, dass bereits getätigte Arbeiten verloren gehen bzw. überschrieben werden.
- Bei Bedarf kann zu früheren Versionen zurückgekehrt werden oder simultan an verschiedenen Versionen gearbeitet werden (auch "Branches"genannt).

1.2 Funktionsüberblick

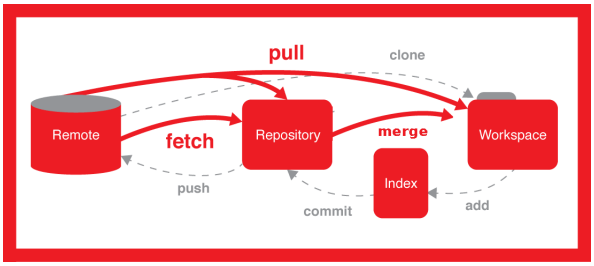

Abbildung 1: Git Funktionsüberblick: Quelle: foxutech.com

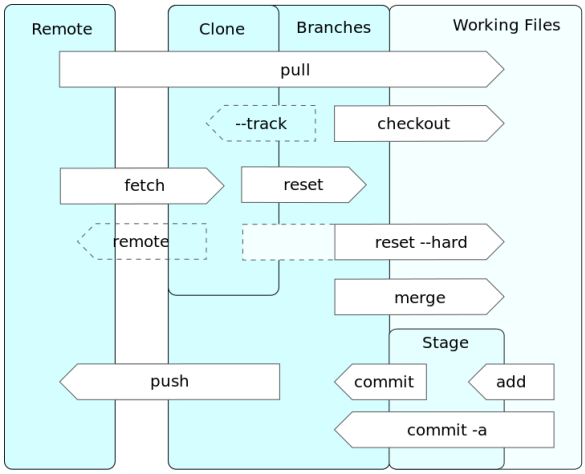

Abbildung 2: Git Funktionsüberblick: Quelle: Wikipedia

1.3 Ein Erklärungsversuch

1.3.1 Repositories

Das Ziel von Git ist es den zeitlichen Verlauf von Files in einem Projekt zu verwalten. Diese Information wird in einem **Repository** gespeichert. Ein Repository beinhaltet unter anderem auch:

- Ein Set von **commit objects**
- Ein Set von Referenzen zu den commit objects welche **heads** genannt werden

Die **commit objects** beinhalten wiederum:

- Ein Set von **files** welche den Zustand des Projekts zu einem bestimmten Zeitpunkt beschreiben
- Einen **SHA1 name** - ein hash über den commit der das Object eindeutig beschreibt - gleiche commits ergeben gleiche hash-sums!
- Eine Referenz auf ein **parent commit object**. Die Referenzen zu den **parent commit objects**, denn vorherigen **commit objects**, repräsentieren damit eine time-line die sich auch als Graph darstellen lässt. Die eigentliche Versionskontrolle kann man sich nun als Manipulation den Commit-Time-Graph vorstellen - geht was schief springt man zurück auf eine frühere Version.

Heads: Ein **head** ist eine einfache Refernez auf ein **commit object**. Der **head**, der auf die aktuelle Verision zeigt wird **HEAD** genannt - also GROSS geschrieben.

Für Erklärungen anhand eines Beispiels siehe: Kapitel4 auf Seite:3

1.4 Nützliche Links und Tutorials im Web

www.rogerdudler.github.io/git-guide/index.de.html
www.thomas-krenn.com/de/wiki/Git_Grundbegriffe
www.help.github.com/articles/git-and-github-learning-resources/
www.sbf5.com/ cduan/technical/git/git-1.shtml
https://about.gitlab.com/

2 Was ist Github bzw. Gitlab?

Github ist ein hosting service welcher mit Git kompatibel ist um remote repositories zentral in einer cloud zu speichern. Es fördert aktive die Verbreitung von open source in dem öffentliche einsehbare Repos gratis gehostet werden. Nur für private Repos muss man zahlen.
Seit Github 2018 von Microsoft übernommen wurde fand ein gewisser Exodus von Open-Source-Projekten weg von Github und hin zu Gitlab statt.
In den folgenden Jahren werden wir mit **Gitlab** arbeiten!

3 Git installieren

- Installation:
  Git für OS X herunterladen: https://git-scm.com/download/mac
  Git für Windows herunterladen: https://gitforwindows.org/
  Git für Linux geht ganz einfach: `sudo apt install git`
- Einen Github oder Gitlab Account anlegen:
  einfach auf https://github.com registrieren

4 Git mit der Commandozeile

4.1 Erstes Anlegen eines Projekts und Repositories

1. In Li(E)clipse ein neues Projekt anlegen
2. In der CMD zum Verzeichnis des neuen Projektes wechseln
3. Ein git-repository anlegen mit: `git init`
   Es wird ein verstecktes Unterverzeichnis (.git) angelegt, welches das lokale Repository beinhaltet.
4. Mit `git status` überprüfen welche Files/Folder „beobachtet“ werden
5. Mit `git add .` alle Files und Folders im Verzeichnis zum repository hinzufügen
6. Nochmal mit `git status` überprüfen ob jetzt alle Files „beobachtet“ werden
7. Ein README.md file erzeugen und dann mit
   `git add README.md`
   hinzufügen (ist „good practice“) und wird auf Github immer angezeigt.
8. Einen ersten Commit durchführen um die Änderungen in den HEAD des lokalen repositories zu bringen:
   `git commit -m "Comment: first commit"`
9. Auf der Github-Webseite in neues repository anlegen, aber **NICHT** die Option: **Initialize this repository with a README** anwählen!
10. Nun das lokale repository auf unser entferntes (github) repository hochladen:
    `git remote add origin https://github.com/markus-sig/Github-Intro.git`
    `git push -u origin master` (erfordert username und password)
11. Auf der Github Webseite überprüfen ob das Projekt komplett abgebildet ist und die README.md angezeigt wird

4.2 Arbeiten am Projekt - The Github flow

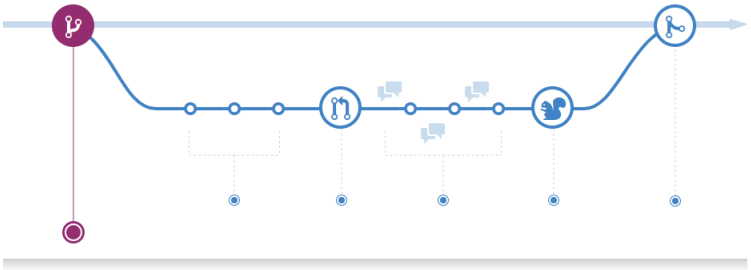

Abbildung 3: Github Flow - branching Quelle: https://guides.github.com

Branching ist ein key-concept von Git. Man geht davon aus, das im **master** branch immer eine lauffähige (deployable) Version liegt (never break the master!!!). Alle neuen Ideen und Änderungen werden in **branches** hinzugefügt und getestet. Neue branches sollen immer vom bestehenden master branch geklont werden. Wenn man dann mit der Funktionalität im neuen branch zufrieden ist, wird dieser wieder mit dem master branch **ge-merged** (zusammengeführt).

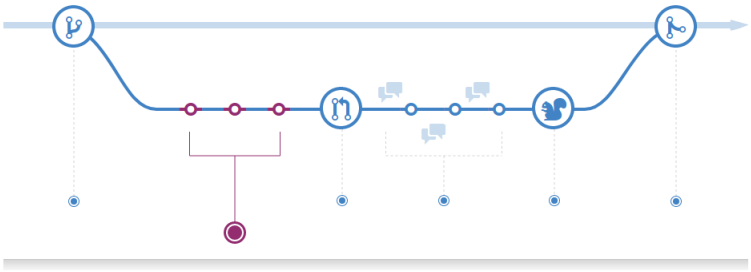

Abbildung 4: Github Flow - commits Quelle: https://guides.github.com

Commits sollen gemacht werden soblald es Änderungen in der branch gibt z.B. neues File erstellt oder ein bestehendes verändert. Diese Commits sollen mit kurzen, klar verständlichen **commit messages** versehen werden. Sie bilden die Entwicklungs-Time-Line und ermöglichen einerseits das Zurückspringen auf frühere Versionen und andererseits machen sie den Entwicklungsprozess für Aussenstehende oder Mitarbeiter transparenter und verständlicher.

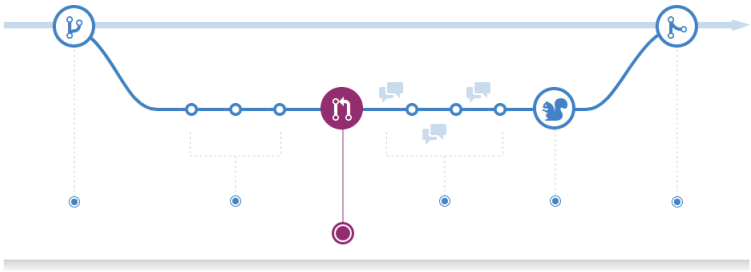

Abbildung 5: Github Flow - pull requests Quelle: https://guides.github.com

Pull requests starten eine Diskussion mit den Mitarbeitern, entweder weil man eine Frage hat oder seine commits von anderen prüfen lassen möchte.

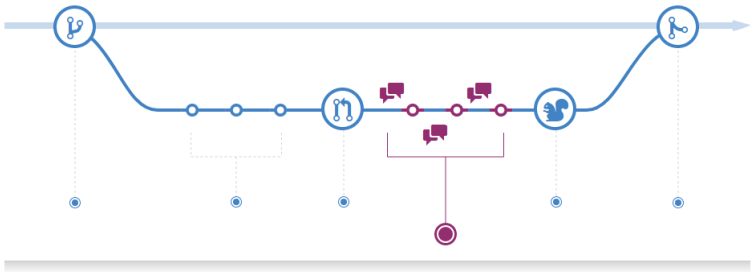

Abbildung 6: Github Flow - discussions Quelle: https://guides.github.com

Discussions: Nun wird über den Code diskutiert, eventuelle Änderungen vorgeschlagen oder, wenn alles okay ist, freigegeben für ein test deployment oder für den merge mit dem master branch.

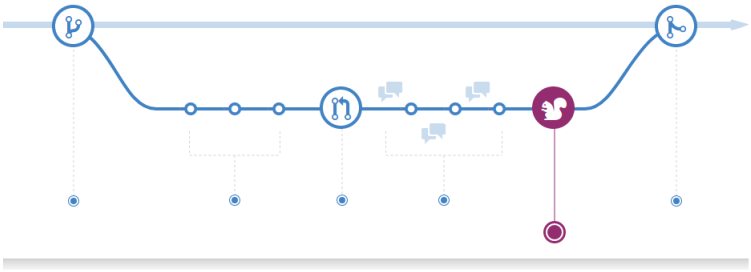

Abbildung 7: Github Flow - branch deployment Quelle: https://guides.github.com

Branch deployment: Ein optionaler Schritt. Man kann von der Branch aus den Code in einem Produktionsumfeld testen um eventuelle letzte Fehler zu finden.

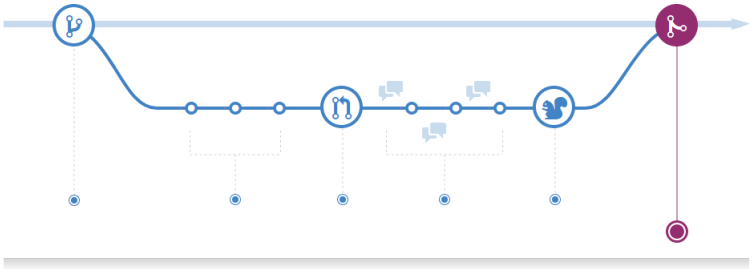

Abbildung 8: Github Flow - merging Quelle: https://guides.github.com

Merging: Die Änderungen oder neuen Features werden (nach reviews und testing) in den master branch übernommen.

4.3 Beispiel Github flow

Voraussetzung: ein Projekt und Repository sind bereits angelegt - siehe Kapitel 4.1 auf Seite 3.

1. Ein Test ob man wirklich den aktuellen master im lokalen repository hat:
   `git status` sollte folgende Statusmeldung liefern:
   Auf Branch master
   Ihr Branch ist auf dem selben Stand wie 'origin/master'. nichts zu committen, Arbeitsverzeichnis unverändert
   Das Command `git branch` kann ebenfalls verwendet werden um den aktuellen branch anzuzeigen.

2. Nun einen neuen branch erzeugen:
   `git checkout -b NameDesNeuenBranches`
   Um den neuen branch zu löschen:
   `git branch -d NameDesNeuenBranches`
   Um zum master zurück zuwechseln:
   `git checkout master`
   Um den branch ins remote repository hochzuladen:
   `git push origin NameDesNeuenBranches`

3. Nun einige Änderungen im neuen Branch vornehmen, zb. neues .py File anlegen

4. Die Änderungen mit `git status` anzeigen lassen und dann mit `git add .` für einen commit vorbereiten.

5. Nun die Änderungen commiten mit `commit -m "changed the helloworld.py file"`
   Achtung: Damit sind sie nur im lokalen repository!

6. Nun die Änderungen und damit den ganzen Branch ins remote repository bringen:
   `git push origin NameDesNeuenBranches` und auf der Github website überprüfen.

7. Angenommen wir sind mit unseren Änderungen zufrieden und wollen den neuen Branch wieder mit dem master mergen. Dann gibt es mehrer Möglichkeiten, entweder auf der Webseite von Github und dann das neue, gemerged'te repository wieder ins lokale repository clonen, oder zuerst local mergen und dann hochladen:
   - remote merge auf der website:
   Die Webseite ist recht verständlich, einfach den Anweisungen folgen und immer Comments schreiben!!! Im Anschluss das lokale repository updaten damit es auf dem gleiche Stand ist wie das remote repository:
   `git pull origin master`
   und eventuell noch den lokalen branch löschen mit `git branch -d NameDesBranches`