Versionskontrolle - Git

von Florian Goldbach

Ich habe folgend eine Beschreibung der in den Kursunterlagen vorhandenen Git-Befehle erstellt. Diese sind mit Screenshots und allgemeinen Beschreibungen versehen.

Das dazugehörige Github Repository finden Sie hier:
https://github.com/Florgol/GitFeatureExploration

git init

Man könnte sagen, dass die Existenz des Repositories voraussetzt, dass der „git init“ Befehl benutzt wurde, nichtsdestotrotz ist er hier dokumentiert.

```
...e\GitFeatureExploration>git init
```

Dieser Befehlt erzeugt ein leeres Git Repository im aktuellen Verzeichnis. Es erscheint dort ein .git Ordner. Dieser sollte allerdings nicht manuell verändert werden, da dies das Repository beschädigen könnte.

git status

„git status“ gibt informationen über 3 verschiedene Szenarien aus.

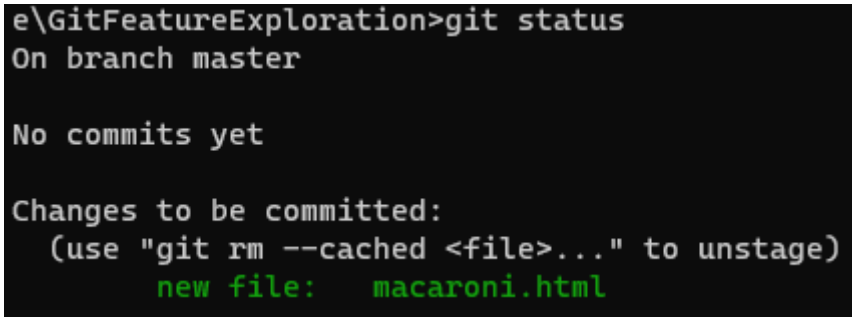
1. Untracked files: Dateien, die sich im Projekt-Ordner befinden, aber noch nicht dem Git Repository hinzugefügt wurden. Als Beispiel habe ich hier eine Datei macaroni.html in meinen leeren Projekt-Ordner kopiert und dann „git status“ aufgerufen. Daraufhin erhalte ich folgenden Output:

```
e\GitFeatureExploration>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        macaroni.html

nothing added to commit but untracked files present (use "git add" to track)
```

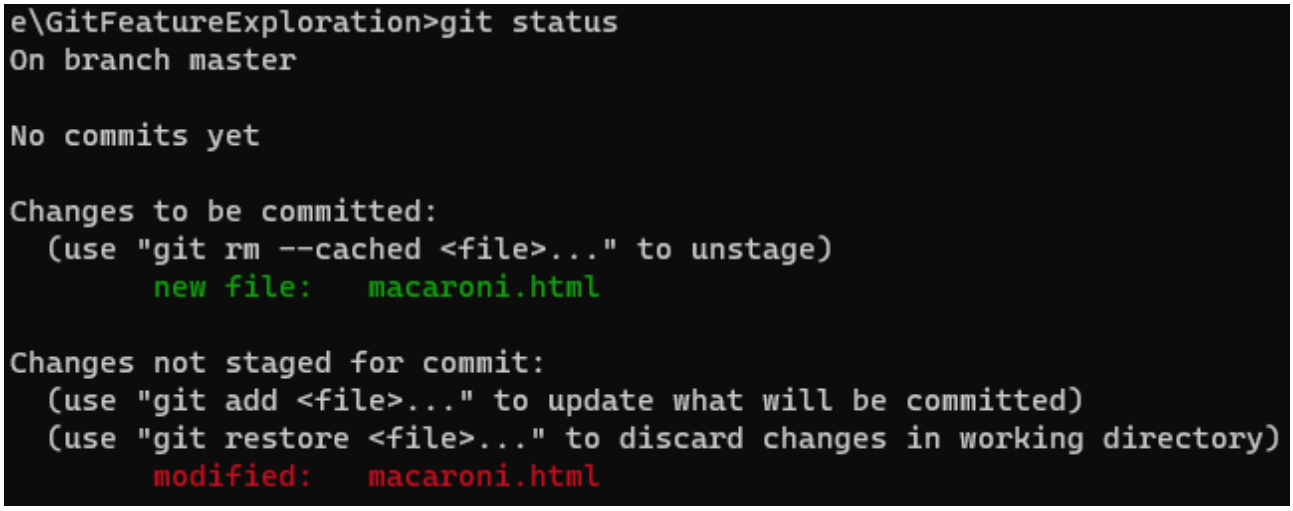
Wie man erkennen kann, werde ich darauf hingewiesen, „Untracked files“ mit „git add <file>“ dem Repository hinzuzufügen.

2. Changes to be comitted: Dateien, die mit dem Befehl „git add <file>“ in die „staging area“ verschoben wurden. Diese sind nun bereit mit dem Befehl „git commit -m “Hier könnte man eine Nachricht für den Commit einfügen“ “ in der „repository history“ gespeichert zu werden.

In unserem Beispiel habe ich „macaroni.html“ zuerst mit „git add macaroni.html“ zur „staging area“ hinzugefügt. Wenn ich nun „git status“ aufrufe erhalte ich die folgenden informationen:



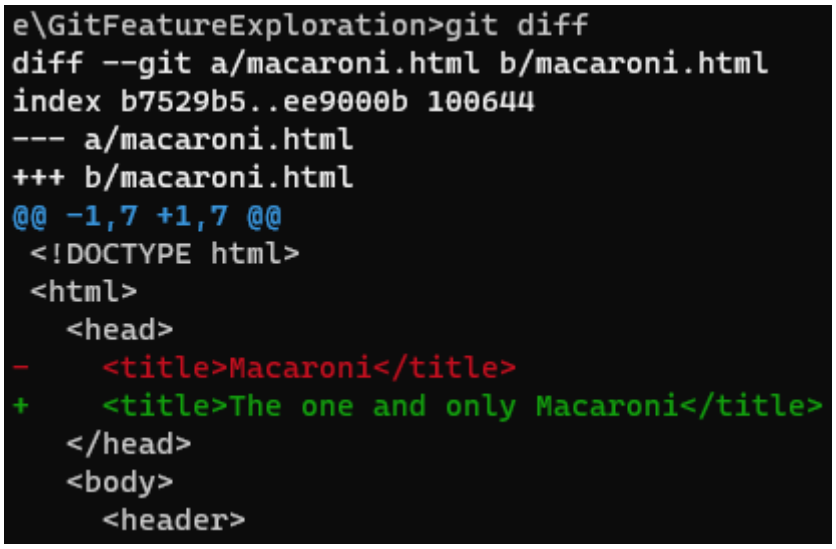
3. Changes not staged for commit: Als nächstes habe ich nun etwas code in meinem macaroni.html file hinzugefügt. Git nimmt diese Veränderung wahr, denn wenn ich nun „git status“ aufrufe erhalte ich die folgende Nachricht:



git diff

Mit „git diff“ kann man sich die Unterschiede zwischen dem „working directory“ (Dateien, die noch nicht zu „staging area“ hinzugefügt wurden), der „staging area“ (Dateien in der „staging area“), „commits“ (Dateien die „committet“ wurden) und „branches“ (Die letzten „commits“ auf 2 „branches“ vergleichen) anzeigen lassen.

In unserem Beispiel habe ich eine Veränderung in meinem „working directory“ vorgenommen. Nun kann ich mir mit „git diff“ die Unterschiede zwischen den Dateien, die bereits in der „staging area“ sind anzeigen lassen. Es handelt sich hier nur um eine Datei: macaroni.html.



Wie beschrieben gibt es weitere Möglichkeiten zu vergleichen. Ich liste hier kurz die „git diff“ Befehle dafür:

Unterschiede zwischen „staged area“ und letztem „commit“: **git diff --staged**
Anmerkung: Nützlich, denn es zeigt an welche Veränderungen man gerade an dem Repository mit einem „commit“ vornehmen möchte.

Unterschiede zwischen 2 spezifischen commits: **git diff <commit1> <commit2>**
Anmerkung: Man kann hier die hashwerte für die commits einfügen (e.g. 5e3e78b)

Unterschiede zwischen 2 branches: **git diff <branch1>..<branch2>**
Beispiel: git diff main..feature1

git remote add origin

Möchte man sein lokales Repository mit einem „remote“ Repository (z.B. auf Github, GitLab, ..) verbinden.

Ich habe in meinem GitHub account ein Repository erstellt und es mit meinem lokalen Repository verbunden:

```
e\GitFeatureExploration>git remote add origin https://github.com/Florgol/GitFeatureExploration
```

Um zu bestätigen, dass die Verbindung erfolgreich war, kann man den Befehl „git remote -v“ benutzen. Dieser zeigt einem an, welche URLs Git für die Befehle „push“ und „fetch“ verwenden wird.

```
e\GitFeatureExploration>git remote -v
origin  https://github.com/Florgol/GitFeatureExploration (fetch)
origin  https://github.com/Florgol/GitFeatureExploration (push)
```

git add

Mit „git add“ kann man eine spezifische Datei, mehrere speziefische Dateien, oder alle veränderten Dateien zur „staging area“ hinzufügen.

Eine spezifische Datei zur „staging area“ hinzufügen: **git add <file>**

```
e\GitFeatureExploration>git add macaroni.html
```

Mehrere spezifische Dateien zur „staging area“ hinzufügen: **git add <file1> <file2>** ..

Alle veränderten Dateien zur „staging area“ hinzufügen: **git add .** ..oder **git add -A**
(beide Befehle tun das gleiche)

git commit

Wenn sich veränderte Dateien in der „staging area“ befinden, kann man diese Veränderungen mit „git commit“ im Repository speichern. Man kann ebenfalls eine kurze beschreibende Nachricht mitangeben, damit man später schnell erkenn kann welche Veränderungen man dort „comitted“ hat.

Der Befehl hat dieses Format: **git commit -m “commit message“**

```
e\GitFeatureExploration>git commit -m "erster commit"
[master (root-commit) d2f492a] erster commit
 1 file changed, 19 insertions(+)
 create mode 100644 macaroni.html
```

Man kann die Übertragung in die „staging area“ auch überspringen und Veränderungen direkt mit diesem Befehl comitten: **git commit -a -m “commit message“**

git log

Mit dem „git log“-Befehl kann man die „commit history“ einsehen. Es gibt verschiedene Möglichkeiten diese darzustellen und zu filtern. Ich werde hier kurz ein paar beschreiben.

git log – Standardbefehl ohne filter

```
e\GitFeatureExploration>git log
commit d2f492ab7cd8f8c52048753039c9392fb275891a (HEAD -> master)
Author: Florgol <38015664+Florgol@users.noreply.github.com>
Date:   Fri Jun 16 07:13:29 2023 +0200

    erster commit
```

Anmerkung: Die lange Nummer neben „commit“ ist ein „commit hash“. In der Theorie sollte dieser global einzigartig sein (Es ist sehr unwahrscheinlich, dass 2 commits den gleichen „commit hash“ haben) .

git log --oneline – Mit diesem Befehl kann man die „commit history“ übersichtlicher darstellen. Ein commit hat hier nur noch eine Zeile.

```
e\GitFeatureExploration>git log --oneline
cc42dbd (HEAD -> master) Added main heading
d2f492a erster commit
```

Anmerkung 1: Die Beschreibung „(HEAD → master)“ bedeutet, dass dies der letzte commit war. Es gibt also immer nur einen commit mit dieser Beschreibung.

Anmerkung 2: Ich habe bemerkt, dass ich leider die Sprachen Englisch und Deutsch in meinem Repository vermischt habe. Daraufhin dachte ich, dass es vielleicht möglich sei die erste commit-Nachricht nachträglich zu verändern. Ich habe recherchiert, und es ist möglch, allerdings wird es als bad practice angesehen dies zu tun wenn zu mehreren an einem Projekt arbeitet (wegen Synchronisationsschwierigkeiten). Ausserdem ist es relativ umfangreicher workaround. Deswegen habe ich ihn hier nicht benutzt und nicht dokumentiert.

git log p – dieser Befehl zeigt alle Unterschiede für jeden commit an. Da dies sehr umfangreiche informationen sind, werden diese im „pager“ dargestellt. Man hat hier weitere Navigationsmöglichkeiten innerhalb seiner shell. Um den Pager zu verlassen kann man „q“ für quit eingeben. Mann kann hier z.B. auch mit den „nach oben“- und „nach unten“-Pfeilen scrollen.

git log -n – Man kann hier für n eine Zahl eingeben und sich die letzten n commits anzeigen lassen.

git log --author=“Florgol“ – Man kann sich hier nur die commits vom Author „Florgol“ anzeigen lassen.

git fetch

Mit dem „git fetch“-Befehl holt man sich die Veränderungen des verbundenen „remote repositores“. Man vereinigt diese noch nicht dem lokalen repository. Dafür benutzt man später den „git merge“-Befehl. Die allgemeine Syntax sieht so aus: **git fetch <remote-repository-name>**

git pull

Der „git pull“-Befehl ist eine Kombination aus dem „git fetch“ und dem „git merge“ befehl. Er führt beide diese Befehle hintereinander aus.
Syntax: **git pull <remote-repository-name> <branch>**
(Bedeutung: siehe „git fetch“ und „git merge“)

```
e\GitFeatureExploration>git pull origin main
```

Anmerkung: Das hatte leider bei mir nicht so funktioniert, da ich meine Repositories separat voneinander erstellt hatte und diese keine gemeinsame „commit history“ hatten. Git erlaubt dann keinen pull. Doch es gibt einen Befehl um einen pull trotzdem zu erlauben:

```
e\GitFeatureExploration>git pull origin main --allow-unrelated-histories
```

git push

Mit dem „git push“-Befehl kann man seine lokalen commits auf ein „remote repository“ übertragen. Man muss natürlich bereits ein remote repository mit seinem lokalen repository verbunden haben.

Der Befehl sieht wie folgt aus: **git push <repository-name> <branch>**

Anmerkung 1: Als ich den „git push“-Befehl ausführen wollte gab es ein Problem, denn auf meinem lokalen repository arbeite ich auf dem „master“ branch. Das „remote repository“, dass ich verbunden habe hat allerdings nur den „main“ branch. Ich löste das Problem indem ich den branch meines lokalen repositories in „main“ umbenannte:

```
e\GitFeatureExploration>git branch -m master main
```

Anmerkung 2: Ausserdem erlaubt Git erst einen push nach einem erfolgreichen pull durchzuführen - also tat ich dies.

Daraufhin konnte ich den „git push“-Befehl erfolgreich durchführen:

```
e\GitFeatureExploration>git push origin main
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Delta compression using up to 12 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (11/11), 1.22 KiB | 1.22 MiB/s, done.
Total 11 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/Florgol/GitFeatureExploration
   522c912..7a39ecd  main -> main
```

git rm

Mit dem „git rm“-Befehl kann man Dateien aus seinem repository löschen. Man sollte dies immer mit diesem Befehl machen und nicht auf manuelle Weise im Ordner tun, damit dies gut dokumentiert ist und es keine Synchronisationsprobleme gibt.

Syntax: **git rm <file>**

```
e\GitFeatureExploration>git rm Garganelli.html
rm 'Garganelli.html'
```

git mv

Der „git mv“-Befehl kann genutzt werden um Dateien zu verschieben oder Dateien umzubenennen. Wie auch beim „git rm“-Befehl sollte man dies immer mit diesem Befehl machen und nicht auf andere Weise, damit dies gut dokumentiert ist und es keine Synchronisationsprobleme gibt.

Syntax um Datei umzubenennen: **git mv <old-file-name> <new-file-name>**

```
e\GitFeatureExploration>git mv macaroni.html Macaroni.html
```

Syntax um Datei in einen Unterordner zu verschieben: **git mv <file> <sub-directory/>**

```
e\GitFeatureExploration>git mv Macaroni.html Macaroni/
```

Syntax um Datei aus einem Unterordner in den Ordner in welchem man sich befidet zu verschieben: **git mv <subdirectory/file> .**

```
e\GitFeatureExploration>git mv Macaroni/Macaroni.html .
```

Anmerkung: Git bemerkt keine leeren Ordner. Diese müssen also nicht mit einem extra git-Befehl gelöscht werden. Sie sind nicht Teil des Projekts (wenn sie leer sind), und können manuell gelöscht werden.

git revert

Der „git revert“-Befehl macht die Veränderungen eines spezifischen commits rückgängig. Es wird ein commit erstellt, der genau die Veränderungen, die in einem angegebenen commit gemacht wurden wieder rückgängig macht.

Syntax: **git revert <commit>**

Bei <commit> setzt man den Hashcode des gewünschten commits, den man rückgängig machen möchte, ein.

```
e\GitFeatureExploration>git revert c48129f
[main 3cdb8c4] Revert "moved Macaroni.html back to root directory"
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename Macaroni.html => Macaroni/Macaroni.html (100%)
```

Wenn man diesen Befehl ausführt öffnet sich der default code editor und man kann dort noch eine eigene „commit“-Nachricht vermerken.

Anmerkung: Der „git revert“-Befehl wird in der „commit history“ vermerkt, denn es ist nur ein weiterer commit, der genau die Veränderungen rückgängig macht, die man in einem angegebenen commit getätigt hat. Dies ist anders beim „git reset“-Befehl, der vergangene commits tatsächlich zerstört (siehe git reset).

git reset

Der „git reset --hard“-Befehl ist gefährlich! Er zerstört die gesamte „commit history“ zeitlich nach einem angegebenen commit und macht alle Veränderungen der zerstörten commits rückgängig.

Syntax: **git reset --hard <commit>**

Ich werde diesen Befehl an der folgenden „commit history“ testen:

```
f45c293 (HEAD -> main, origin/main) added a another maybe not so useful paragraph
32e5700 added a maybe not so useful paragraph
3cdb8c4 Revert "moved Macaroni.html back to root directory"
c48129f moved Macaroni.html back to root directory
2aec182 Moved Macaroni.html int Macaroni folder
5e09446 Changed case of the macaroni.html to Macaroni.html
ac954ac removed Garganelli Page
```

```
e\GitFeatureExploration>git reset --hard 3cdb8c4
HEAD is now at 3cdb8c4 Revert "moved Macaroni.html back to root directory"
```

Die „commit history“ nach dem ausgeführten „git reset –hard“-Befehl:

```
3cdb8c4 (HEAD -> main) Revert "moved Macaroni.html back to root directory"
c48129f moved Macaroni.html back to root directory
2aec182 Moved Macaroni.html int Macaroni folder
5e09446 Changed case of the macaroni.html to Macaroni.html
ac954ac removed Garganelli Page
daec262 Added Garganelli Page
```

Als ich danach die Veränderung auf das „remote repository“ übertragen wollte, erlaubte mir das Git nicht, da sich die lokale „commit history“ offensichtlich an einem früheren Zeitpunkt befindet.

Man kann diese Veränderung allerdings mit dem folgenden Befehl erzwingem:
git push <repository> <branch> --force

Anmerkung: Man kann „--force“ auch durch „-f“ als Kurzform ersetzen.

```
e\GitFeatureExploration>git push origin main -f
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/Florgol/GitFeatureExploration
 + f45c293...3cdb8c4 main -> main (forced update)
```

git checkout

Mit dem „git checkout“-Befehl kann man zwischen branches wechseln, man kann aber auch einen neuen branch erstellen. Ebenfalls kann man Dateien auf den stand eines angegebenen commits bringen – darauf gehe ich hier aber nicht ein.

Wechseln zwischen branches: **git checkout <branch-name>**

Erstellen eines neuen branches: **git checkout -b <branch-name>**

Im folgenden erstelle ich einen neuen branch:

```
e\GitFeatureExploration>git checkout -b tortelloni-feature
Switched to a new branch 'tortelloni-feature'
```

Mit git branch kann ich mir die verschiedenen branches anzeigen lassen. Der aktuelle wird durch „*“ gekennzeichnet:

```
e\GitFeatureExploration>git branch
  main
* tortelloni-feature
```

Hier wechsel ich zurück zu dem „main“ branch:

```
e\GitFeatureExploration>git checkout main
Switched to branch 'main'
```

```
e\GitFeatureExploration>git branch
* main
  tortelloni-feature
```

git merge

Mit dem „git merge“-Befehl kann man 2 branches zusammenführen. Es gibt dabei einen branch, der dem anderen hinzugefügt wird. Normalerweise fügt man dem „main branch“ (oder master) einen anderen branch, z.B. ein feature, hinzu.

Für den „git merge“-Befehl bedeutet das, dass man sich im branch in den man den anderen mergen möchte (also z.B. im main branch) befinden muss.

Syntax: **git merge <branch-name>**

```
e\GitFeatureExploration>git merge tortelloni-feature
Updating 3cdb8c4..19fb94f
Fast-forward
 Tortelloni.html | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Tortelloni.html
```

Github Pull Request

von Florian Goldbach

Ich werde folgend die Schritte erklären, die notwendig sind um eine Github „pull request“ für ein „fremdes“ Repository zu erstellen.

1. Fork des Repository

Eine „fork“ ist eine Kopie eines Repository. Der „Fork“-Button befindet sich oben rechts. Das kopierte Repository ist nun bei den eigenen Repositories zu finden.

2. Das geforkte Repository klonen

Wir klonen das geforkte Repository auf unserem lokalen Rechner um Veränderungen vorzunehmen.

Wir führen den „git clone“-Befehl in unserem gewünschten lokalen Ordner aus:
git clone https://github.com/DEIN_BENUTZERNAME/REPOSITORY_NAME.git

3. Einen Branch erstellen (optional)

Mit dem „git checkout -b“-Befehl können wir einen neuen branch erstellen:
git checkout -b <branch-name>

4. Änderungen vornehmen

Alle Änderungen im lokalen Repository vornehmen.

5. Änderungen comitten

Wir „comitten“ die Änderungen mit den dazugehörigen Befehlen:
git add .
Git commit -m “Beschreibende Nachricht“

6. Wir übertragen die Änderungen auf unser remote Repository (push)

Mit diesem „git push“-Befehl:
git push origin <branch-name>

7. Wir erstellen einen Pull request

In unserem geforkten Repository (in unserem Account) auf Github wechseln wir zu dem branch, auf dem wir die Veränderungen vorgenommen haben. In der oberen Leiste klicken wir auf „Contribute“ und dann auf „Open Pull Request“. Auf der nächsten Seite ist eine Zusammenfassung der Änderungen und man kann dort einen Titel und Kommentar zu seinem „pull request“ angeben, bevor man diesen abschickt.

Mein "pull request“ für die ESA: **#428**