Versionskontrolle: Git verwenden (Teil 1)

BBZW, Modul 426

Patrick Bucher

29.02.2024

Git: Hilfe

Globale Hilfe:

```
$ git help
```

Befehlsspezifische Hilfe:

```
$ git help [verb]
$ git [verb] --help
$ man git-[verb]
```

Beispiel: Hilfe für `git init`

```
$ git help init
$ git init --help
$ man git-init
```

Voraussetzungen

1. Sie haben Git *installiert*.
   - Windows: Git Bash
   - Linux/Mac OS: `git`-Paket
2. Sie haben Git *konfiguriert*.
   - `git config --global user.email [vorname]_[nachname]@sluz.ch`
   - `git config --global user.name "[Vorname] [Nachname]"`
3. Sie haben einen SSH-Schlüssel *erstellt* und in Ihrem Profil auf code.frickelbude.ch hinterlegt.
   - `ssh-keygen -t ed25519 -C [vorname]_[nachname]@sluz.ch`
   - `cat ~/.ssh/id_ed25519.pub` (Ausgabe im Profil hinterlegen)

Repository erstellen: `git init`

Wird `git init` in einem bestimmten Verzeichnis ausgeführt, wird ein Unterverzeichnis namens `.git` erstellt:

```
$ mkdir project
$ cd project
$ git init
```

Der Inhalt des Ordners `project` kann nun mit `git` als Repository verwaltet werden.

Aufgabe 1 (2 Minuten, Einzelarbeit)

1. Öffnen Sie die *Git Bash*.
2. Erstellen Sie ein Verzeichnis namens `git-exercises`.
3. Wechseln Sie in dieses Verzeichnis und initialisieren Sie ein neues Repository.

Repository klonen: `git clone`

Ein bestehendes Repository kann mittels `git clone` kopiert werden:

```
$ git clone git@code.frickelbude.ch:m426/meta.git
```

Hier wird der ganze Quellcode von Git inklusive Versionsgeschichte heruntergeladen.

Als zusätzlicher Parameter kann ein Name für den Zielordner definiert werden:

```
$ git clone git@code.frickelbude.ch:m426/meta.git m426-meta
```

Aufgabe 2 (2 Minuten, Einzelarbeit)

1. Besuchen Sie im Browser die Seite code.frickelbude.ch/m426.
2. Wählen Sie das Repository `meta` aus.
3. Klonen Sie das Repository einmal ohne einen lokalen Ordnernamen anzugeben...
4. ...und einmal mit dem lokalen Ordnernamen `m426-meta`.

Zustände: Konzepte

Eine Datei wird entweder durch Git verwaltet (*tracked*) oder nicht (*untracked*). Durch Git verwaltete Dateien können drei Zustände haben:

1. *unmodified*: unverändert (Zustand wie zuletzt festgeschrieben)
2. *modified*: verändert (Zustand im Arbeitsverzeichnis verändert)
3. *staged*: vorgemerkt (nach Veränderung für nächsten Zustand vorgemerkt)

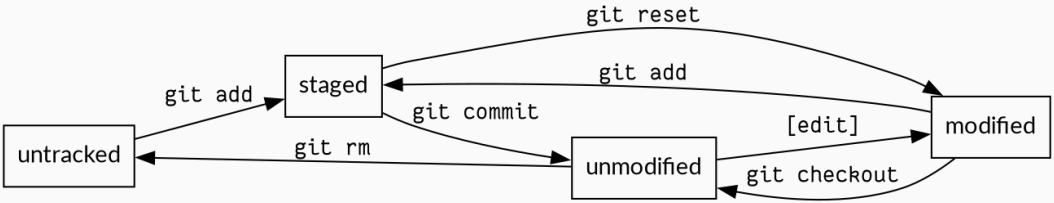

Abbildung 1: Lebenszyklus von Dateizuständen

Zustände: `git status`

Der aktuelle Zustand der Dateien kann mithilfe von `git status` angezeigt werden:

```
$ git status
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)
...
```

Die Ausgabe von `git status` ist sehr ausführlich. Die Ausgabe kann mit dem `--short`-Parameter kompakter erfolgen:

```
$ git status --short
```

Zustände: Ausgabe von `git status`

```
...
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   addition.c
        new file:   params.c
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   addition.c
        modified:   hello.c
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        LICENSE
```

Zustände: Ausgabe von `git status --short`

```
$ git status --short
AM addition.c
 M hello.c
A  params.c
?? LICENSE
```

Zustände: Interpretation (I)

Staging-Bereich	Arbeitsverzeichnis	Datei
`A`	`M`	`addition.c`
	`M`	`hello.c`
`A`		`params.c`
`?`	`?`	`LICENSE`

- Zwei Spalten:
  1. links: Staging-Bereich
  2. rechts: Arbeitsverzeichnis
- Drei Werte:
  1. `?`: *untracked*
  2. `A`: *staged* (`A` wie “added”)
  3. `M`: *modified*

Zustände: Interpretation (II)

- `addition.c`
  - die Datei wurde verändert (`[edit]`)
  - die Datei wurde hinzugefügt (`git add`)
  - die Datei wurde erneut verändert (`[edit]`)
- `hello.c`
  - die Datei wurde verändert (`[edit]`)
- `params.c`
  - die Datei wurde verändert (`[edit]`)
  - die Datei wurde hinzugefügt (`git add`)
- `LICENSE`:
  - die Datei wurde angelegt und ist *untracked*

`addition.c` hat drei Zustände gleichzeitig! (Repository, Arbeitsverzeichnis, Staging-Bereich)

Aufgabe 3 (5 Minuten, Einzelarbeit)

Für die weiteren Aufgaben wird das Repository `git-exercises` verwendet.

1. Erstellen Sie in Ihrem Repository die folgenden Textdateien mit dem jeweiligen Inhalt:
   - `foo.txt` (Inhalt: `foo`)
   - `bar.txt` (Inhalt: `bar`)
   - `qux.txt` (Inhalt: `qux`)
2. Führen Sie nun den Befehl `git status --short` aus und betrachten Sie die Ausgabe.
3. Führen Sie diesen Befehl erneut aus und leiten Sie die Ausgabe in die Datei `aufgabe-3.out` weiter.

Dateien hinzufügen: `git add`

Mithilfe von `git add` wird eine veränderte Datei zum Staging-Bereich hinzugefügt:

```
$ git add LICENSE
```

Je nach Kommandozeile können mehrere Dateien mit Glob-Patterns hinzugefügt werden:

```
$ git add *.c
$ git add utils/*.{h,c}
```

Vorsicht beim Hinzufügen aller Dateien!

```
$ git add *
```

Aufgabe 4 (5 Minuten, Einzelarbeit)

1. Führen Sie den Befehl `git add` auf die Dateien `foo.txt` und `bar.txt` (aber **nicht** auf `qux.txt`!) aus.
2. Führen Sie nun den Befehl `git status --short` aus und betrachten Sie die Ausgabe.
3. Führen Sie diesen Befehl erneut aus und leiten Sie die Ausgabe in die Datei `aufgabe-4.out` weiter.
4. Welche Unterschiede fallen Ihnen gegenüber Aufgabe 3 (`aufgabe-3.out`) auf? Dokumentieren Sie diese in der Datei `aufgabe-3-4-diff.txt`.

Dateien ignorieren: `.gitignore`

Manche grosse (Kompilate) oder sensitive (Keys, Tokens) Dateien sollen *nicht* mit Git verwaltet werden!

Diese können in der Datei `.gitignore` im Arbeitsverzeichnis mithilfe von Mustern definiert werden:

```
*.class    # keine .class-Dateien
bin/       # keine bin-Verzeichnisse
/tmp       # kein tmp-Verzeichnis direkt im Arbeitsverzeichnis
*.out      # keine .out-Dateien
!audit.out # Ausnahme für die Datei audit.out
```

`.gitignore` selber wird mit Git verwaltet!

Es gibt Vorlagen für verschiedene Programmiersprachen und Frameworks.

Aufgabe 5 (5 Minuten, Einzelarbeit)

1. Erstellen Sie eine leere Datei namens `foobar.exe`.
2. Führen Sie den Befehl `git status --short` aus.
3. Erstellen Sie die Datei `.gitignore` und definieren Sie darin ein Muster, um sämtliche Dateien mit der Endung `.exe` zu ignorieren.
4. Führen Sie den Befehl `git status --short` erneut aus. Wenn Sie den vorherigen Schritt richtig gemacht haben, sollte `foobar.exe` nun nicht mehr aufgelistet werden.
5. Wenden Sie den Befehl `git add` auf die Datei `.gitignore` an.

Zusatzfrage: Warum beginnt der Dateiname von `.gitignore` mit einem Punkt?

Inhaltliche Änderungen anschauen (I): `git diff`

`git status` gibt Auskunft über *veränderte Dateien*.

`git diff` hingegen zeigt die *inhaltlichen Änderungen* an. Es können verschiedene Zustände miteinander verglichen werden:

- Arbeitsverzeichnis mit Staging-Bereich: `git diff`
- Staging-Bereich mit Repository: `git diff --staged` (oder `--cached`)
- Früherer Commit mit Repository: `git diff [commit-hash]`
- Zwei frühere Commits: `git diff [old-commit-hash] [new-commit-hash]`

In der Regel genügen die ersten 4-8 Zeichen des Commit-Hashes (eindeutig).

Inhaltliche Änderungen anschauen (II): Ausgabe von `git diff`

```
diff --git a/hello.c b/hello.c
index cb76401..19611e3 100644
--- a/hello.c
+++ b/hello.c
@@ -1,8 +1,6 @@
-#include <stdio.h>
-
 int main(int argc, char *argv[])
 {
-    printf("Hello, World!\n");
+    puts("Hello, World!");

     return 0;
 }
```

Inhaltliche Änderungen anschauen (III): Ausgabe von `git diff`

Die ersten vier Zeilen beziehen sich auf die interne Arbeitsweise von `git diff`.

`@@ -1,8 +1,6 @@` gibt den Bereich an:

- alter Zustand: Zeilen 1 bis 8
- neuer Zustand: Zeilen 1 bis 6

Zeilen haben ein Präfix:

- `-`: diese Zeile wurde gelöscht
- `+`: diese Zeile wurde hinzugefügt
- `[leer]`: diese Zeile bleibt unverändert

Inhaltliche Änderungen anschauen (IV): Statistiken

Die Änderungsstatistiken (mengenmässig, nicht inhaltlich) kann mithilfe des `--shortstat`-Parameters angeschaut werden:

```
$ git diff --shortstat
 1 file changed, 1 insertion(+), 3 deletions(-)
```

- eine Datei wurde verändert
- eine Zeile wurde hinzugefügt
- drei Zeilen wurden entfernt

Aufgabe 6 (5 Minuten, Einzelarbeit)

1. Führen Sie den Befehl `git diff` aus. Welche Änderungen werden angezeigt?
2. Führen Sie nun den Befehl `git diff --cached` aus. Welche Änderungen werden *nun* angezeigt?

Änderung festschreiben: `git commit`

Mithilfe von `git commit` werden die vorgemerkten Änderungen im Staging-Bereich unwiderruflich festgeschrieben.

Jede Änderung muss mit einer sog. *Commit Message* kommentiert werden:

- Einzeilige Kommentare können mit dem Parameter `-m` mitgegeben werden:
  - `git commit -m 'use puts instead of printf'`
- Mehrzeilige Kommentare werden mit dem Texteditor verfasst:
  - `git commit`
  - `git config core.editor` definiert den Editor

Änderungen festschreiben: Commit Message (I)

Der Texteditor wird geöffnet und zeigt folgenden Überblick an (mit # beginnende Zeilen sind Kommentare und gehen *nicht* in den Kommentar):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Your branch is up to date with 'origin/master'.
#
# Changes to be committed:
#       modified:   hello.c
#
```

Änderungen festschreiben: Commit Message (II)

Ist der Kommentar verfasst, kann der Texteditor geschlossen werden (speichern nicht vergessen, hierzu wird eine temporäre Datei verwendet):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# ...
Use puts() instead of printf(), which has some benefits:
1. no explicit newline needed at the end
2. stdio.h is no longer needed
3. the program runs faster and becomes smaller
```

Form und Inhalt von Commit Messages sind je nach Projekt und Organisation oftmals vorgegeben.

Aufgabe 7 (5 Minuten, Einzelarbeit)

1. Führen Sie `git status --short` aus. Und betrachten Sie die Ausgabe.
2. Führen Sie den Befehl `git commit` aus. Beschreiben Sie Ihre Änderungen im Texteditor, speichern Sie ab, und schliessen Sie den Texteditor.
3. Führen Sie nun noch einmal `git status --short` aus. Welche Änderungen stellen Sie Gegenüber Schritt 1 fest?
4. Fügen Sie nun die verbleibenden Dateien mit `git add` hinzu.
5. Erstellen Sie einen Commit mit `git commit -m`, indem Sie hinter dem Parameter `-m` eine Commit Message angeben.
6. Führen Sie nun noch einmal `git status --short` aus. Wie sieht die Ausgabe jetzt aus?

Zentrale Repositories (I): `git remote`

Obwohl Git dezentral funktioniert, braucht man für die Zusammenarbeit mit anderen Entwicklern zentral gelagerte Repositories: *Remote Repositories* oder kurz *Remotes*.

Wird ein Repository gecloned, verfügt man standardmässig über ein Remote namens `origin` (der “Ursprung” oder die “Quelle”):

```
$ git clone https://code.frickelbude.ch/patrick/git-demo.git
$ cd git-demo
$ git remote
origin
$ git remote -v
origin  git@code.frickelbude.ch:patrick/git-demo.git (fetch)
origin  git@code.frickelbude.ch:patrick/git-demo.git (push)
```

Zentrale Repositories (II): `git remote add`

Weitere Remotes können hinzugefügt werden, etwa um einen Mirror einzurichten. (Auf GitHub muss das Repository zuerst manuell angelegt werden.)

Mit `git remote add [name] [url]` kann ein Remote hinzugefügt werden:

```
$ git remote add github git@github.com:patrickbucher/git-demo.git
```

Details von Remotes können mit `git remote show [remote]` angezeigt werden:

```
$ git remote show origin
```

Ein Remote kann auch umbenannt werden:

```
$ git remote rename origin the-one-and-only
```

Remotes synchronisieren (hochladen): `git push`

Mit `git push [origin]` können die am lokalen Repository vorgenommenen Änderungen auf das Remote Repository hochgeladen werden:

```
$ git push github
```

Wird das Remote (`github`) weggelassen, wird `origin` verwendet.

Aufgabe 8 (5 Minuten, Einzelarbeit)

1. Gehen Sie auf code.frickelbude.ch und erstellen Sie ein neues Repository.
2. Nennen Sie dieses `git-exercises`. (Sie brauchen keine weiteren Einstellungen vorzunehmen.)
3. Auf der nächsten Seite werden Hinweise angezeigt, u.a. der Befehl `git remote add […]`.
   3.1 Wechseln Sie in die Git Bash, wo Sie im Verzeichnis `git-exercises` sein sollten.
   3.2 Fügen Sie das Remote gemäss den Anweisungen hinzu.
4. Führen Sie den Befehl `git push origin master` aus.
   4.1 Laden Sie die Seite im Browser neu. Was fällt Ihnen dabei auf?

Remotes synchronisieren (herunterladen): `git fetch` und `git pull`

Mit `git fetch [remote]` werden alle Metadaten von einem Remote abgeholt:

```
$ git fetch origin
```

Mit `git pull [remote]` werden zusätzlich die Änderungen heruntergeladen:

```
$ git pull origin
```

Befehle im Überblick (I)

- `git config`: Git konfigurieren
- `git init`: Repository erstellen
- `git clone`: Repository kopieren
- `git status`: Zustand einsehen
- `git diff`: Unterschiede anzeigen
- `git add`: Datei(en) vormerken
- `git commit`: Änderungen festschreiben

Befehle im Überblick (II)

- `git remote`: Verwaltung von Remotes
- `git fetch`: Änderungen herunterladen
- `git pull`: Änderungen herunterladen und anwenden
- `git push`: Änderungen hochladen