GIT auf der Konsole

Inhalt

Quelle: Diese Dokumentation und die darin enthaltenen Bilder beruhen alle auf dem Material von Frank Schiebel (https://info-bw.de/faecher:informatik:oberstufe:git:start). Dort finden sich auch weitergehende Anleitungen für die Arbeit mit GIT auf der Konsole.

Nach der Installation von GIT stehen eine GIT-Konsole und eine einfache Git-GUI zur Verwaltung der Repositorys zur Verfügung. Für die produktiven Arbeit werden in der Regel in die Entwicklungsumgebungen (z. B. bei IntelliJ oder Visual Studio Code) integrierte GIT-Clients verwendet. Es gibt auch spezielle Git-Verwaltungsprogramme, die eine umfangreichere Verwaltung der GIT-Projekte zulassen (z. B. Git-Cola, Smart-Git).

Im Unterricht kann es sinnvoll sein, zunächst mit der GIT-Konsole zu starten. In der GIT-Konsole müssen alle Befehle von Hand eingetippt werden. Daher wird jeder Befehl mit mehr Bedacht ausgeführt. In einer GUI ist schnell mal ein falscher Klick gemacht, der nur schwer rückgängig gemacht werden kann. Hat man allerdings die Arbeitsweise von GIT verstanden, ist es wesentlich komfortabler, mit einer GUI zu arbeiten.

1. Konfiguration

Bevor mit git gearbeitet werden kann, müssen Name und Mailadresse festgelegt werden. Dazu sind in einer Shell die folgenden Kommandos auszuführen. Unter Windows ist das Programm "Git Bash" zu starten, in Linux/MacOS reicht ein gewöhnliches Terminal aus. Name und Mailadresse sind durch passende Werte zu ersetzen.

```
$ git config --global user.name "Max Mustermann"
$ git config --global user.email max@example.org
```

Sollte keine Berechtigung bestehen, diese Einstellungen systemweit ("global") vorzunehmen, z.B. an den PCs in der Schule, ist es erforderlich, sie für jedes Repository einzeln festzulegen, **nachdem** dieses initialisiert wurde:

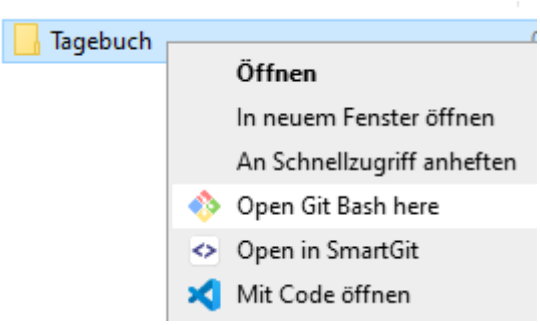
```
$ git config user.email "meine@mail.adresse.hier"
$ git config user.name "John Doe"
```

Diese Befehle speichern die Einstellungen nur für das Repository, in dem gerade gearbeitet wird. Für ein weiteres Repository muss die Email-Adresse erneut konfiguriert werden.

2. Erste Schritte mit Git

2.1. Initialisieren

Um die Abläufe und die Funktionsweise zu erproben, soll zunächst ein Verzeichnis unter Versionskontrolle gestellt werden, in dem ein Tagebuch angelegt wird. Es wird also ein Verzeichnis `tagebuch` erstellt und dort ein Git-Repository initialisiert:




```
$ git init
Initialized empty Git repository in F:/Tagebuch/.git/
```

Nun steht das Verzeichnis `tagebuch` unter Versionskontrolle. Das lokale Git-Repository befindet sich im Unterverzeichnis `.git`:

```
$ ls -la
total 52
drwxr-xr-x 1 XXX 197121 0 Oct  9 14:19 ./
```

```
drwxr-xr-x 1 XXX 197121 0 Oct  9 14:18 ../
drwxr-xr-x 1 XXX 197121 0 Oct  9 14:19 .git/
```

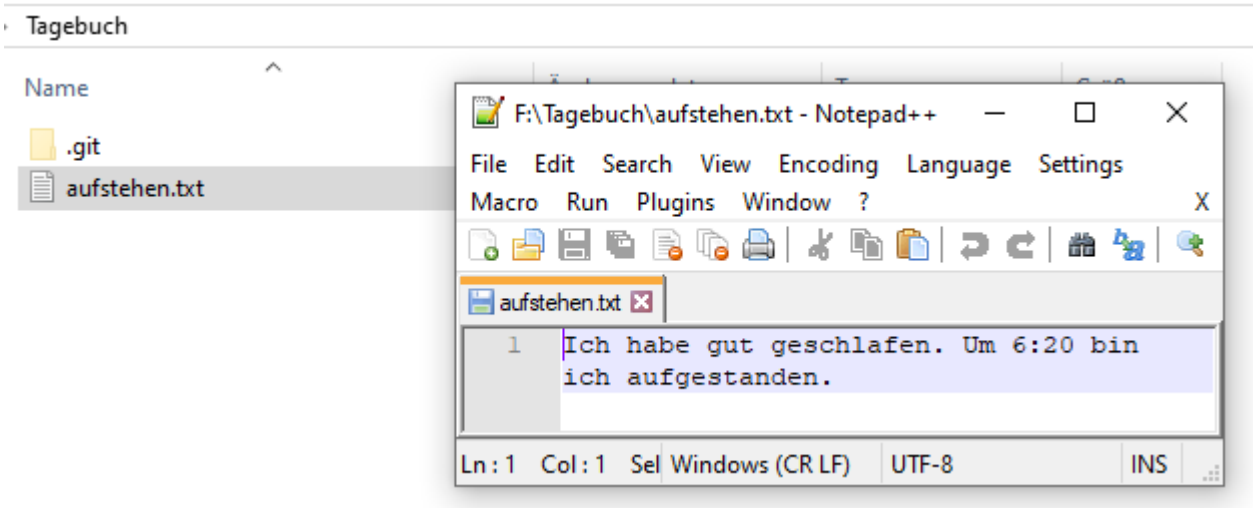
2.2. Repository Status anzeigen lassen

Das Verzeichnis `tagebuch` ist jetzt ein "git-Repository" - es wird von git "beobachtet", so dass Änderungen in diesem Verzeichnis und seinen Unterverzeichnissen nachverfolgt werden können. Mit dem Befehl `git status` kann der aktuelle Status des "Repos" angezeigt werden[1] :

```
$ git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

2.3. Ein erster Tagebucheintrag

Mit einem Texteditor (z. B. notepad++ bei Windows oder kate bei Linux, **nicht** mit Word oder Writer!) wird eine Datei `aufstehen.txt` angelegt. In diese Datei kann beispielsweise hineingeschrieben werden, wie geschlafen wurde und wann aufgestanden wurde. Wichtig ist, dass die Datei im Verzeichnis `tagebuch` abgespeichert wird.



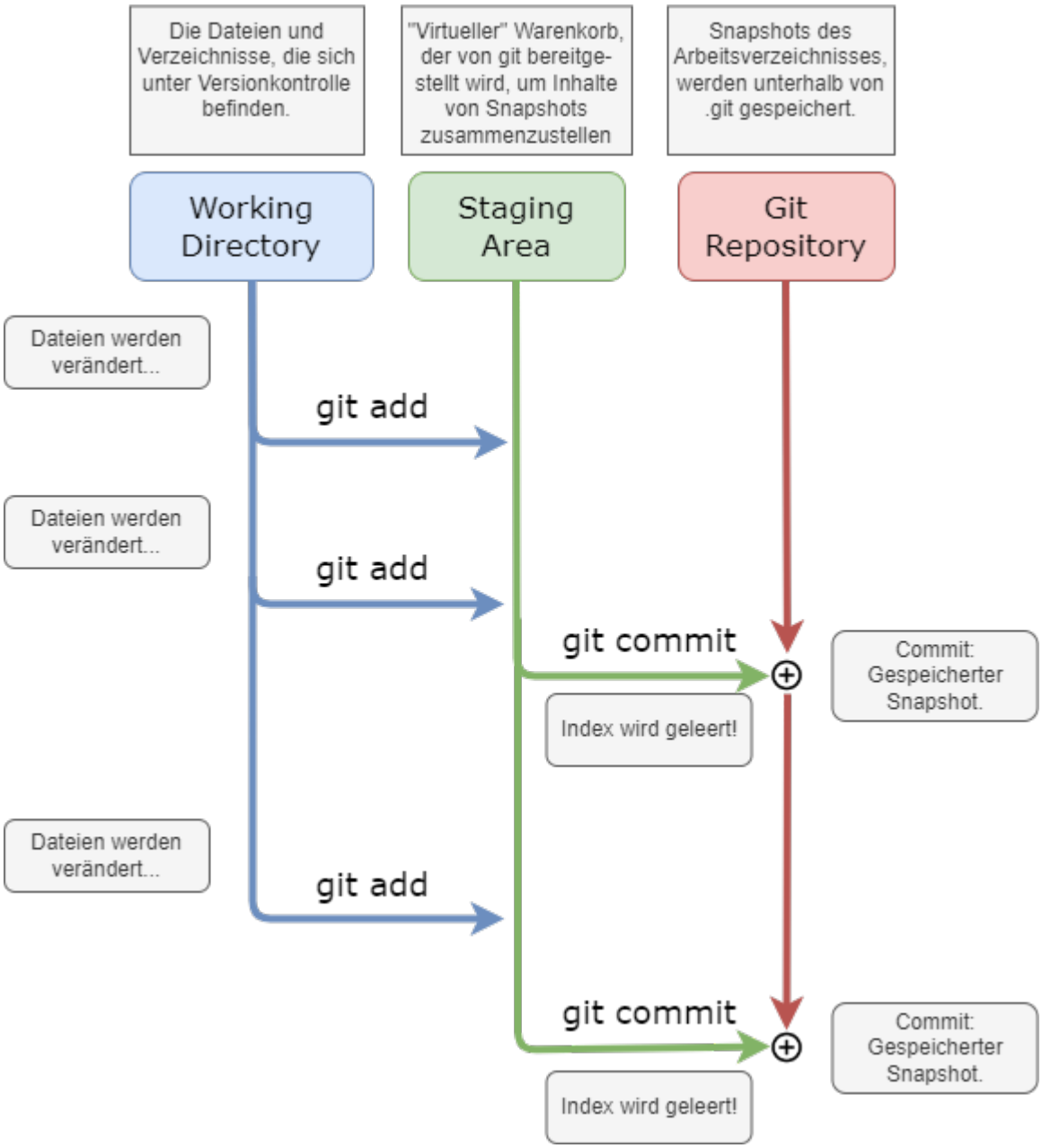
Das Tagebuch enthält nun einen Eintrag in `aufstehen.txt`. Es soll jetzt der Zustand des Tagebuchs an dieser Stelle so in der Versionsverwaltung festgehalten werden, dass er später wieder verwendet werden kann.

2.4. Ein erster Commit

Um den git-Workflow zu verstehen, müssen drei Begriffe unterschieden werden: Das Arbeitsverzeichnis ("Working Directory"), die Staging Area ("Index") und das eigentliche

Repository.

1. **Arbeitsverzeichnis ("Working Directory"):** Das ist das Verzeichnis, welches zuvor mit `git init` unter Versionskontrolle gestellt wurde, mit allen seinen Dateien und Unterverzeichnissen, so wie es auf der Festplatte vorgefunden wird. Das "spezielle" Verzeichnis `.git` wird dabei ignoriert, es dient der internen Verwaltung der Abläufe durch git.
2. **Staging Area ("Index")**: In der Staging Area werden zunächst alle Dateien zusammengetragen, die in einem nächsten Schritt zu einem Snapshot zusammengefasst und im Repository gespeichert werden sollen. Es gibt zahlreiche Anwendungsfälle, bei denen nicht alle Änderungen des Arbeitsverzeichnisses in einem Snapshot festgehalten werden möchten: Es kann z.B. sinnvoll sein, die Änderungen auf mehrere Snapshots aufzuteilen oder Dateien auszuschließen, die gar nicht unter Versionskontrolle gestellt werden sollen, beispielsweise Compilate von Java Programmen (class-Dateien). Daher ist es sinnvoll, zunächst über die Staging Area auszuwählen, welche Dateien gesichert werden sollen.
3. **Repository:** Wenn in der Staging Area alle Dateien für den nächsten Snapshot zusammengestellt wurden, kann ein neuer Snapshot erstellt werden. Ein solcher Snapshot heißt **Commit** und wird durch einen Hashwert identifiziert, außerdem werden Metainformationen wie Zeit und Name des Commiters festgehalten. Ein Commit wird mit dem Befehl `git commit` durchgeführt. Bei einem Commit wird außerdem die Staging Area wieder geleert, da alle Änderungen, die dort vorgemerkt waren, in den Snapshot überführt wurden. Für den nächsten Commit müssen die Dateien dort wieder hinzugefügt werden.



2.5. Commit: Schritt für Schritt

Neue Dateien befinden sich zunächst "nur" im Arbeitsverzeichnis und werden von git ignoriert. Mit `git status` kann dies überprüft werden, solche Dateien tauchen dort in der Liste der "Unversionierten Dateien" auf, für das Tagebuch sieht das so aus:

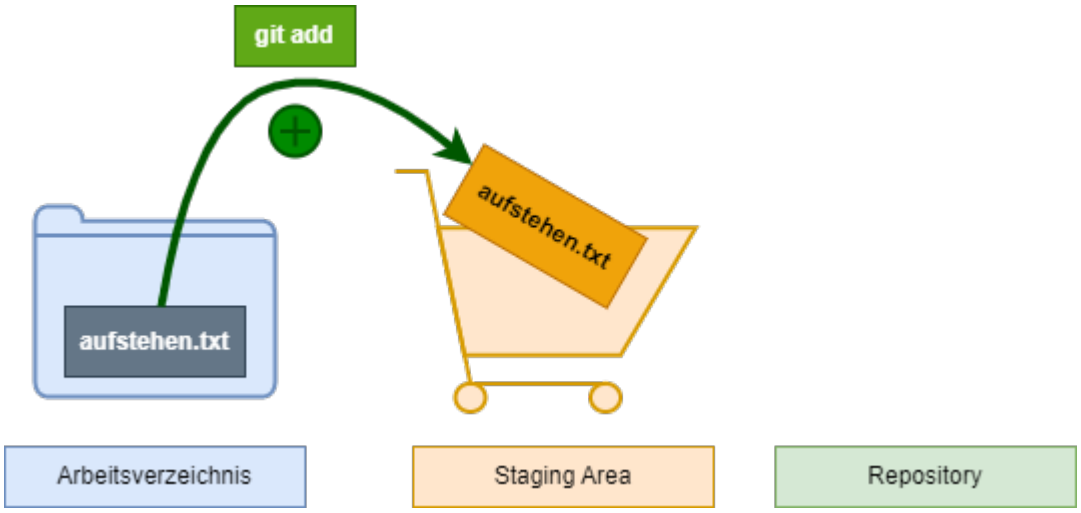
```
$ git status
On branch main

No commits yet

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

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

Mit dem Befehl `git add` wird eine Datei im Index vorgemerkt - dies kann man sich wie ein Einkaufswagen vorstellen, in dem neue Dateien und Änderungen gesammelt werden, bis ein Punkt erreicht ist, den man sich "merken" möchte.

Im Folgenden wird die einzige Datei `aufstehen.txt` zum Index hinzugefügt:

```
$ git add aufstehen.txt
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   aufstehen.txt
```

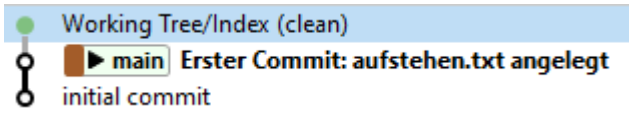


Wenn man mit den im Index vorgemerkten Änderungen zufrieden ist, kann ein "Commit" durchgeführt werden, um den Zustand aller im Index befindlichen Dateien zu speichern. Mit dem Befehl `git commit -m "Erster Commit: aufstehen.txt angelegt"` wird ein Commit mit einer Commit-Message angelegt (Parameter `-m`).

```
$ git commit -m "Erster Commit: aufstehen.txt angelegt"
[main (root-commit) 9eefa56] Erster Commit: aufstehen.txt angelegt
 1 file changed, 1 insertion(+)
 create mode 100644 aufstehen.txt
```

Wenn der Status des Arbeitsverzeichnisses jetzt erneut abgefragt wird, erhält man folgende Ausgabe:

```
$ git status
On branch main
nothing to commit, working tree clean
```



Man erkennt, dass der Index wieder leer ist ("nichts zum Commit vorgemerkt"). Nun können weitere Änderungen im Tagebuch vorgenommen werden und zu allen wichtigen Zeitpunkten der Zustand der Dateien in einem Commit festgehalten werden.

2.6. Frühstück

Aufgabe:

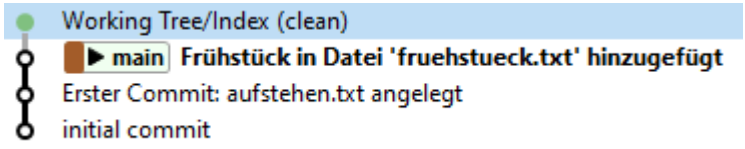
1. Halten Sie in der Datei `fruehstueck.txt` fest, was es zum Frühstück gab.
2. Kontrollieren Sie mit `git status`, dass die Datei jetzt existiert, sie jedoch noch nicht unter Versionskontrolle steht.
3. Fügen Sie die Datei `fruehstueck.txt` mit dem Befehl `git add fruehstueck.txt` zum Index hinzu.
4. Erstellen Sie einen Commit für das Frühstück. Vergessen Sie die Commit-Message nach der Option `-m` nicht.
5. Überprüfen Sie den Zustande des Repositorys erneut.

Es wurde nun ein zweiter Commit erstellt:

Es wurde zwar nur die Datei `fruehstueck.txt` zum Commit vorgemerkt und anschließend mit `git commit` "commited", **ein Commit beinhaltet jedoch stets den Zustand aller unter Versionskontrolle stehender Dateien im Arbeitsverzeichnis**, in diesem Fall ist in dem zweiten Commit also auch die (unveränderte) Datei `aufstehen.txt` enthalten!

Ein Commit kann also wie im Bild dargestellt als Archivbox betrachtet werden, in dem jeweils der Zustand aller versionierten Dateien festgehalten ist. Ein Commit wird durch einen hexadezimalen "Hashwert" identifiziert, das ist gewissermaßen die eindeutige Nummer eines Commits, z.B. `2f40bf7`. Mit dem Befehl `git log` können die Commits aufgelistet werden:

```
$ git log
commit 2f40bf7011e453259db1621014979353003262df (HEAD -> main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt

commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200

    Erster Commit: aufstehen.txt angelegt
```



Man erkennt hier auch, dass die eigentlichen Commit-Hashes sehr viel länger sind, als das Beispiel oben vermuten lässt; für die Identifizierung eines Commits reichen die ersten 7 Stellen des Hashes aus.

2.7. Mittagessen

Aufgabe:

1. Fügen Sie dem Tagebuch den Eintrag `mittagessen.txt` als Datei hinzu, zunächst ohne diese zu versionieren.
2. Ändern Sie die Datei `fruehstueck.txt` und schreiben Sie zusätzlich `Schokolade` in die Datei.
3. Überprüfen Sie mit `git status` den Zustand des Repositorys. Das Repo sollte ungefähr so aussehen:

```
$ git status
On branch main
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:   fruehstueck.txt

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

no changes added to commit (use "git add" and/or "git commit -a")
```

Es wurden jetzt **zwei** Dinge geändert: In der Datei `fruehstueck.txt` wurde eine Änderung vorgenommen. Die Datei `mittagessen.txt` wurde neu hinzugefügt.

Wenn nun der nächste Commit vorbereitet wird, kann mit dem Befehl `git add` wieder ausgewählt werden, welche Änderungen in den Commit übernommen werden. Um dies zu demonstrieren, werden die beiden vorgenommenen Änderungen im Folgenden auf zwei Commits aufgeteilt.

```
$ git add fruehstueck.txt

$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   fruehstueck.txt

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

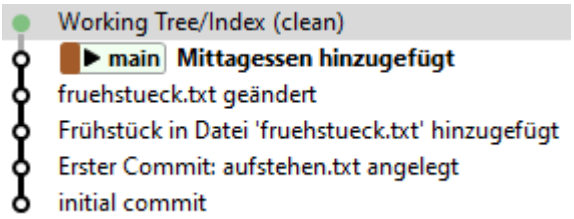
Jetzt wurden die Änderungen von `fruehstueck.txt` für den nächsten Commit vorgemerkt, die neue Datei `mittagessen.txt` wird allerdings nicht in diesen übernommen. Mit `git commit -m`

`"fruehstueck.txt geändert"` wird der Commit ausgeführt:

```
$ git commit -m "fruehstueck.txt geändert"
[main 1c669a2] fruehstueck.txt geändert
 1 file changed, 2 insertions(+), 1 deletion(-)

$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        mittagessen.txt

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

Für den nächsten Commit wird jetzt die Datei `mittagessen.txt` übernommen:

```
$ git add mittagessen.txt

$ git commit -m "Mittagessen hinzugefügt"
[main 311cb29] Mittagessen hinzugefügt
 1 file changed, 1 insertion(+)
 create mode 100644 mittagessen.txt

$ git status
On branch main
nothing to commit, working tree clean
```



Sie können nun Dateien selektiv in einem Verzeichnis unter Versionskontrolle stellen. Mit jedem Commit wird ein **Snapshot** des Zustands erzeugt, den die versionierten Dateien zum Zeitpunkt des Commits haben. Dateien, die nicht mit `git add` unter Versionskontrolle gestellt wurden, werden von git nicht beachtet. Im folgenden Kapitel wird die Versionsgeschichte genauer untersucht und Zeitsprünge durchgeführt, um ältere Versionen zu betrachten.

3. Versionshistorie untersuchen

3.1. Blick in die Vergangenheit

Nachdem einige Commits gemacht wurden, kann nun das Repository untersucht werden:


```
$ git log
commit 311cb296f5c7099ed88cfa6336c089eb03ddbb35 (HEAD -> main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:09:01 2024 +0200

    Mittagessen hinzugefügt

commit 1c669a21d0b0ad1422e285a745780d7e99c9034e
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:07:56 2024 +0200

    fruehstueck.txt geändert

commit 2f40bf7011e453259db1621014979353003262df
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt

commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200
```

```
    Erster Commit: aufstehen.txt angelegt
```

Es werden einige Informationen für jedem Zeitpunkt angezeigt, an dem ein Snapshot der unter Versionskontrolle stehenden Dateien im Arbeitsverzeichnis erstellt wurde:

- Den Hashwert des Commits
- Den Autor
- Datum und Zeit, zu der der Snapshot erstellt wurde
- Commit-Message

Außerdem wird durch den Zeiger `HEAD` angezeigt, in welchem Zustand sich das Arbeitsverzeichnis befindet.

3.2. Änderungen zwischen Commits ansehen

Um den Unterschied des aktuellen Arbeitsstandes (HEAD) zu vorhergehenden zu untersuchen, kann folgender Befehl verwendet werden: `git diff HEAD~1`. Dies bedeutet: "Zeige alle Unterschiede im Verzeichnis zwischen dem Commit, auf den HEAD gerade zeigt, und dem vorigen Commit" - die Ausgabe ist zunächst etwas gewöhnungsbedürftig:

```
$ git diff head~1
diff --git a/mittagessen.txt b/mittagessen.txt
new file mode 100644
index 0000000..54777bb
--- /dev/null
+++ b/mittagessen.txt
@@ -0,0 +1 @@
+Blumenkohl
\ No newline at end of file
```

Die Ausgabe sagt, dass in der Datei mittagessen.txt neu angelegt wurde und in diese eine Zeile eingefügt wurde: Blumenkohl.

Der Vergleich kann auch mit weiter zurückliegenden Commits durchgeführt werden: z. B. zwei Commits in die Vergangenheit: `git diff HEAD~2`:

```
$ git diff head~2
diff --git a/fruehstueck.txt b/fruehstueck.txt
index 307fc9b..b7a56aa 100644
--- a/fruehstueck.txt
+++ b/fruehstueck.txt
@@ -1,3 +1,4 @@
 Müsli
 Kaffee
```

```
-Brot
\ No newline at end of file
+Brot^M
+Schokolade
\ No newline at end of file
diff --git a/mittagessen.txt b/mittagessen.txt
new file mode 100644
index 0000000..54777bb
--- /dev/null
+++ b/mittagessen.txt
@@ -0,0 +1 @@
+Blumenkohl
\ No newline at end of file
```

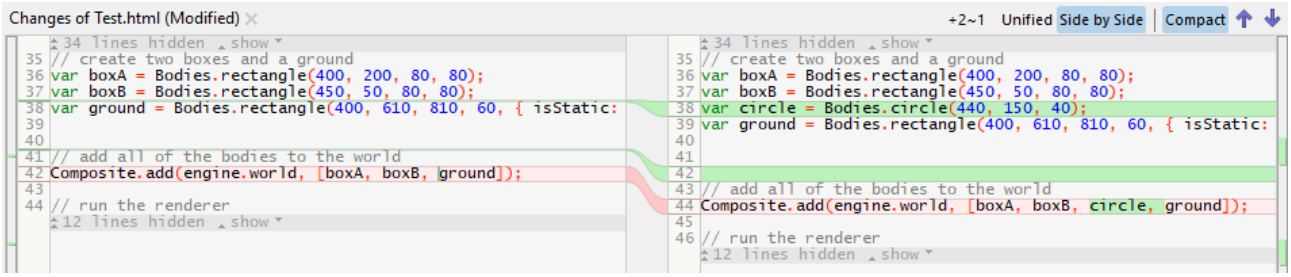
Die Ausgabe sagt, dass in den letzten zwei Commits bis zum aktuellen HEAD die folgenden Änderungen im Repo stattgefunden haben:

Es wurde eine neue Datei angelegt - mittagessen.txt und in diese eine Zeile eingefügt: Blumenkohl. In der zuvor bereits vorhandenen Datei fruehstueck.txt wurde nach den drei schon vorhandenen Zeilen eine weitere Zeile Schokolade eingefügt. Am Ende der Zeile Brot wurde außerdem ein Zeilenumbruch hinzugefügt.

Aufgabe:

1. Untersuchen Sie die Unterschiede in Ihrem Repo zwischen dem HEAD auf main und einigen vorigen Commits.
2. Erstellen Sie auf dem main Branch einen weiteren Commit, bei dem in einer Ihrer Dateien eine Zeile entfernt wird. Untersuchen Sie, wie die Ausgabe von git diff jetzt aussieht - woran erkennt man, dass die Zeile entfernt wurde?
3. Erstellen Sie einen Commit, bei dem eine Datei entfernt wird (git rm <Dateiname>, dann einen Commit erstellen). Untersuchen Sie, wie die Ausgabe von git diff jetzt aussieht.

Für eine bessere Darstellung der Unterschiede sollte eine GUI für GIT genutzt werden.



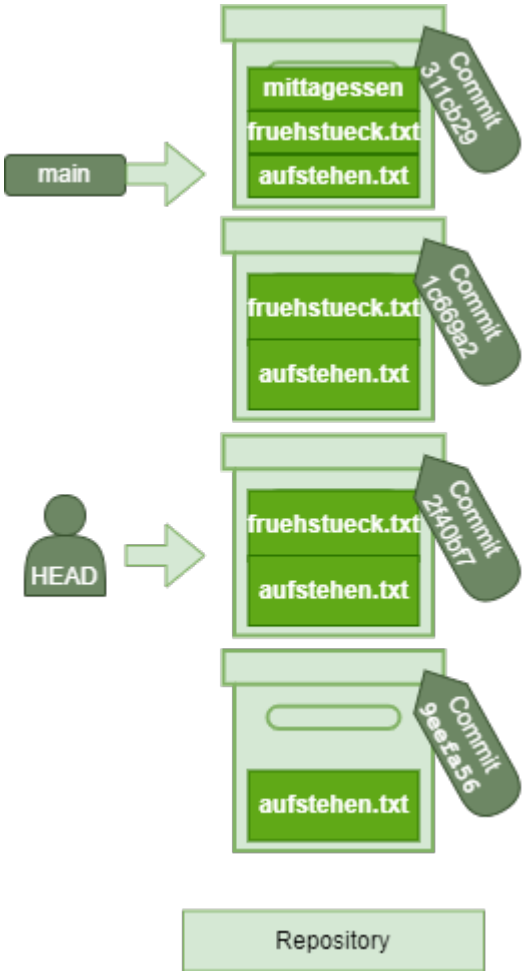
3.3. Reise in die Vergangenheit

Man sollte nur dann zu einem älteren Stand der Dateien zurückkehren, wenn das Arbeitsverzeichnis keine nicht committeten Änderungen beinhaltet.

("Sauberes Arbeitsverzeichnis")

```
$ git status
On branch main
nothing to commit, working tree clean
```

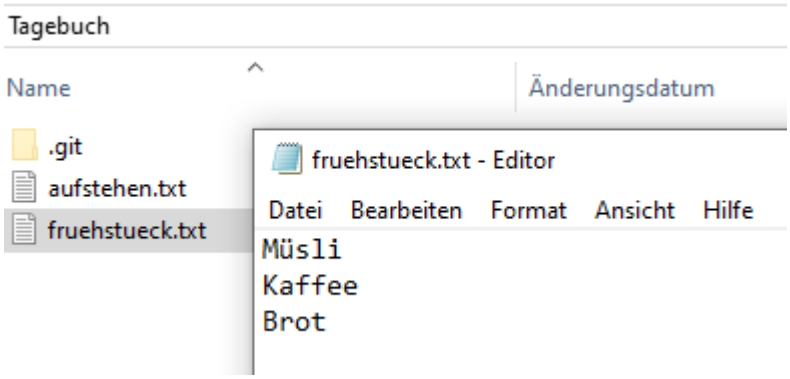
Der Zustand des Arbeitsverzeichnisses kann aus den Commits wiederhergestellt werden. Es ist also möglich, das Arbeitsverzeichnis in genau den Zustand zurückzuversetzen, in dem es beim Commit `2f40bf7011e453259db162101497935300 3262df` - "Frühstück in Datei 'fruehstueck.txt' hinzugefügt" war – oder eben zu jedem anderen Zeitpunkt, an dem der Zustand des Arbeitsverzeichnisses als Snapshot commitet wurde. Der Befehl dazu ist **`git checkout <Commit-Hash>`**. Es reicht dabei, die ersten 7 Zeichen des Hashwertes, z.B. `2f40bf7`, anzugeben:




```
$ git log
commit 311cb296f5c7099ed88cfa6336c089eb03ddbb35 (HEAD -> main)
...
commit 2f40bf7011e453259db162101497935300 3262df
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt
```

```
...

$ git checkout 2f40bf7
Note: switching to '2f40bf7'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:
  git switch -c <new-branch-name>

Or undo this operation with:
  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 2f40bf7 Frühstück in Datei 'fruehstueck.txt' hinzugefügt

$ git status
HEAD detached at 2f40bf7
nothing to commit, working tree clean
```

Was ist denn jetzt passiert?



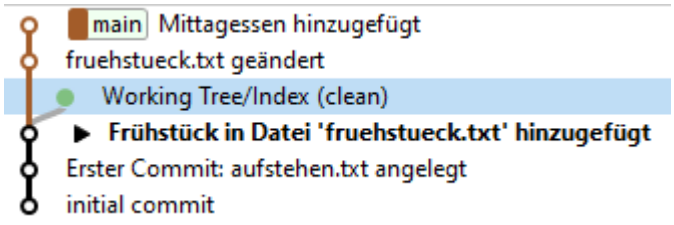
1. Zunächst einmal befinden sich die Dateien im Verzeichnis jetzt in dem Zustand, in dem sie waren, als der Commit `2f40bf7` erstellt wurde. Überprüfen Sie das.
2. Der HEAD ist zum Commit `2f40bf7` gewandert. HEAD ist ein Zeiger. Er springt in gewisser Weise in der Versionsgeschichte herum und zeigt immer auf denjenigen Commit, der im Arbeitsverzeichnis gerade "ausgecheckt" ist.
3. Die Versionsgeschichte ist kürzer geworden:

   ```
   $ git log
   commit 2f40bf7011e453259db162101497935300326 2df (HEAD)
   Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
   ```

```
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt

commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200

    Erster Commit: aufstehen.txt angelegt
```

Git zeigt mit `git log` standardmäßig die Versionsgeschichte an, die zu dem Commit geführt hat, den man im Arbeitsverzeichnis gerade angezeigt bekommt. Die anderen Commits sind jedoch nicht verloren. Die Option `--all` zeigt dies. Es kann also auch in die Gegenwart zurückgekehrt werden.

```
commit 311cb296f5c7099ed88cfa6336c089eb03ddbb35 (main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:09:01 2024 +0200

    Mittagessen hinzugefügt

commit 1c669a21d0b0ad1422e285a745780d7e99c9034e
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:07:56 2024 +0200

    fruehstueck.txt geändert

commit 2f40bf7011e453259db16210149793530032 62df (HEAD)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt

commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200

    Erster Commit: aufstehen.txt angelegt
```



Neben dem HEAD gibt es einen weiteren Zeiger `main`, der auf den aktuellen Stand der Arbeit[2] zeigt - hier auf den Commit `311cb29`.

Aufgabe:

1. Wechseln Sie durch einen Checkout zu den verschiedenen Zeitpunkten in der Versionsgeschichte und untersuchen Sie, welche Dateien vorhanden sind und welchen Text sie enthalten.

3.4. Manipulation der Vergangenheit

Was passiert, wenn nun die Vergangenheit geändert wird? Welche Auswirkungen hat dies auf die aktuelle Gegenwart? Die Antwort auf die zweite Frage ist kurz: keine! Wenn in der Versionsgeschichte zurückgegangen und Änderungen durchgeführt werden, wird immer eine parallele Zeitschiene begonnen. Diese parallele Zeitschiene hat eine neue Zukunft. Jede Zeitschiene ist ein sogenannter **Branch**. `Main` ist der Name des Standard-Branches. In der Schule sollte man nicht mit mehreren Branches arbeiten, da dies leicht zur Verwirrung führen kann.

Wenn mit checkout in die Vergangenheit zurückgekehrt wird, wird lediglich der HEAD verschoben. Es wird kein neuer Branch erstellt. Der HEAD ist also losgelöst (**detached HEAD**) von einem Branch. Man kann sich umsehen und auch Dateien verändern; wenn anschließend jedoch wieder in der Versiongeschichte "gesprungen" wird, gehen diese Änderungen verloren.

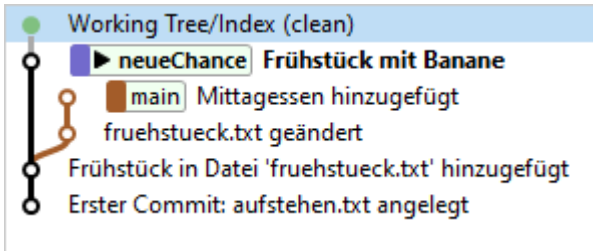
Wenn eine neue Zeitschiene begonnen wird, kann man sich entscheiden, was mit der aktuellen Zeitschiene passieren soll:

1. Sie ist weiterhin verfügbar, dann muss aber ein neuer Branch abgezweigt werden. (git switch)
2. Sie wird gelöscht und steht nicht mehr zur Verfügung (git reset).

Variante 1 (nur für Experten zu empfehlen): neuer Branch

```
$ git switch -c "neueChance"
Switched to a new branch 'neueChance'
$ git log --all
commit 311cb296f5c7099ed88cfa6336c089eb03ddbb35 (main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:09:01 2024 +0200

    Mittagessen hinzugefügt

commit 1c669a21d0b0ad1422e285a745780d7e99c9034e
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 15:07:56 2024 +0200

    fruehstueck.txt geändert

commit 2f40bf7011e453259db1621014979353003262df (HEAD -> neueChance)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200
```

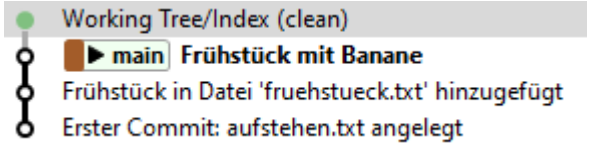
```
    Frühstück in Datei 'fruehstueck.txt' hinzugefügt

commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200

    Erster Commit: aufstehen.txt angelegt
```



Neben main gibt es jetzt auch noch den Branch "neueChance". Der HEAD zeigt auf diesen Branch. Damit können Änderungen nun in diesem Branch commited werden. Eine parallele Zeitschiene beginnt. Mit `git switch <branchname>` kann zwischen den verschiedenen Zeitschienen hin- und hergewechselt werden.

Variante 2: die aktuelle Zeitschiene löschen Soll unwiderruflich zu einem alten Arbeitsstand zurückgekehrt werden, können die letzten Änderungen verworfen werden:

```
$ git reset --hard 2f40bf7
HEAD is now at 2f40bf7 Frühstück in Datei 'fruehstueck.txt' hinzugefügt

$ git log --all
commit 2f40bf7011e453259db1621014979353003262df (HEAD -> main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:35:50 2024 +0200

    Frühstück in Datei 'fruehstueck.txt' hinzugefügt

commit 9eefa5687ebae993ce5b7ca0f597159418f1d7cd
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Wed Oct 9 14:31:24 2024 +0200

    Erster Commit: aufstehen.txt angelegt
```

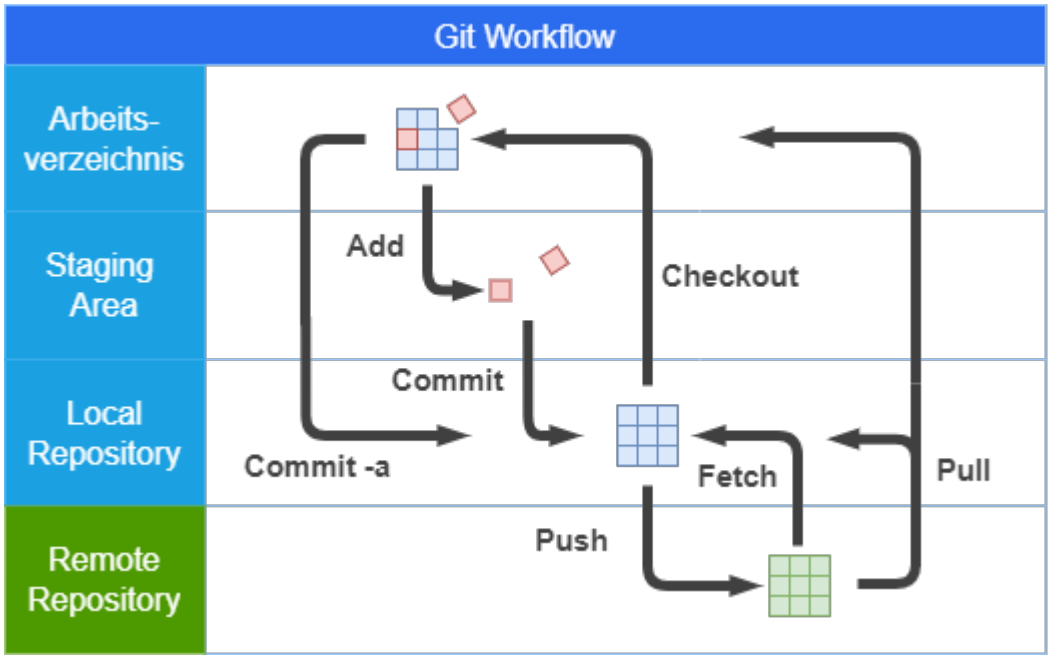


Der HEAD ist nicht losgelöst, da er weiterhin auf main zeigt. Die nachfolgenden Commits sind nicht mehr vorhanden und daher auch bei der Option `--all` nicht mehr zu sehen.

4. Speichern in der Cloud

Zunächst ist ein Git-Repo eine vollkommen lokale Angelegenheit - alle wichtigen Informationen und die Snapshots werden im .git-Verzeichnis gespeichert.

Um besser zusammenarbeiten zu können, ist es möglich, ein Repo über einen Cloud-Speicher anderen zur Verfügung zu stellen. Das Land Baden-württemberg stellt für diesen Zweck eine datenschutzkonforme Plattform "Gitcamp" bereit.



4.1. Klonen eines Repositorys

Ein so veröffentlichtes Repo kann man "klonen". Dabei wird ein Repository aus der Cloud auf den eigenen Computer kopiert. Es wird in ein Unterverzeichnis kopiert, das genauso heißt, wie das Repository. Git merkt sich, von welcher Quelle das Repository stammt (origin) und auf welchem Arbeitsstand die Quelle ist (origin/main, origin/HEAD).

```
$ git clone https://fortierung.gitcamp-bw.de/Thomas.Schaller/UebungVorlage
Cloning into 'UebungVorlage'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), 7.17 KiB | 152.00 KiB/s, done.

$ git log
commit 33f53e800b45c072c1d914b2ab0c0a55e5abf977 (HEAD -> main, origin/main,
origin/HEAD)
Author: Thomas Schaller <thomas.schaller@noreply.Domain>
Date:   Tue Oct 1 09:57:04 2024 +0200
```

```
    Initial commit
```

Der Begriff des Klonens ist hier wörtlich zu nehmen - jetzt existiert eine vollständige Kopie des Repos auf dem lokalen Rechner, die alle Commits des ursprünglichen Repos nachverfolgbar enthält.

4.2. Änderung auf den Server zurückkopieren

Nun kann man mit dem Repo lokal ganz normal arbeiten, der wesentliche Unterschied zum ausschließlich "lokalen" Repository ist, dass dieses Repository "weiss", woher es kommt – das ermöglicht es, Änderungen auch wieder auf den entfernten Server zurück zu übertragen. Der dazu verwendete Befehl lautet `git push`.

Zunächst bearbeitet man lokal Dateien im Repo und erzeugt einen (oder mehrere Commits) :

```
$ git log
commit d81cb957d62bccb2f68378aea7f6232fc2f81026 (HEAD -> main)
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Mon Oct 14 16:40:29 2024 +0200

    Neue Datei angelegt

commit 33f53e800b45c072c1d914b2ab0c0a55e5abf977 (origin/main, origin/HEAD)
Author: Thomas Schaller <thomas.schaller@noreply.Domain>
Date:   Tue Oct 1 09:57:04 2024 +0200

    Initial commit
```

Das Cloud-Verzeichnis (origin) bleibt dabei auf dem vorherigen Stand. Nur das lokale Repository (HEAD→main) kennt die Änderungen. Durch einen push werden diese Änderungen hochgeladen (origin ist nun auch auf dem neusten Stand):

```
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 324 bytes | 162.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://fortbildung.gitcamp-bw.de/Thomas.Schaller/UebungVorlage
   33f53e8..d81cb95  main -> main

$ git log
commit d81cb957d62bccb2f68378aea7f6232fc2f81026 (HEAD -> main, origin/main,
origin/HEAD)
```

```
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Mon Oct 14 16:40:29 2024 +0200

    Neue Datei angelegt

commit 33f53e800b45c072c1d914b2ab0c0a55e5abf977
Author: Thomas Schaller <thomas.schaller@noreply.Domain>
Date:   Tue Oct 1 09:57:04 2024 +0200

    Initial commit
```

Damit landen die Änderungen auf dem Server, von dem das Repo zuvor geklont wurde. Es darf aber natürlich nicht jeder auf jedes im Internet zugänglich Repository Änderung zurückspielen, sondern nur diejenigen, die dazu berechtigt sind.

Andere Personen oder man selbst an einem anderen Rechner kann diese Änderungen nun herunterladen und in das eigene, vorher geklonte Repository integrieren. Dazu gibt es den Befehl `git pull`.

```
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 260 bytes | 9.00 KiB/s, done.
From https://fortbildung.gitcamp-bw.de/Thomas.Schaller/UebungVorlage
   d81cb95..2ce4edf  main       -> origin/main
Updating d81cb95..2ce4edf
Fast-forward
 neu.txt | 1 +
 1 file changed, 1 insertion(+)

$ git log
commit 2ce4edf2df9254891d16dce195dc0a51fc0db357 (HEAD -> main, origin/main,
origin/HEAD)
Author: Thomas.Schaller <thomas.schaller@noreply.Domain>
Date:   Mon Oct 14 16:48:51 2024 +0200

    neu.txt aktualisiert

commit d81cb957d62bccb2f68378aea7f6232fc2f81026
Author: Thomas Schaller <thomas.schaller@zsl-rska.de>
Date:   Mon Oct 14 16:40:29 2024 +0200

    Neue Datei angelegt

commit 33f53e800b45c072c1d914b2ab0c0a55e5abf977
Author: Thomas Schaller <thomas.schaller@noreply.Domain>
```

```
Date:   Tue Oct 1 09:57:04 2024 +0200

    Initial commit
```

Hier wurden von einer anderen Person an der Datei `neu.txt` Änderungen vorgenommen und gepusht. Diese Änderungen stehen nun im lokalen Repository auch zur Verfügung und werden sofort in das Arbeitsverzeichnis kopiert. (HEAD → main, origin/main, origin/HEAD zeigen alle auf den gleichen Commit). Der Befehl **`git fetch`** lädt das Repository auch herunter, überträgt die Änderungen aber nicht sofort ins Arbeitsverzeichnis.

Nehmen mehrere Personen gleichzeitig Änderungen an einem Repository vor, kommt es in der Regel zu sogenannten Konflikten: mehrere Personen haben die gleiche Datei bearbeitet. Die verschiedenen Versionen müssen nun zusammengeführt werden (merge). Als Git-Neuling sollte man diesem Problem aus dem Weg gehen, indem man zunächst alleine an einem Repository arbeitet und bei der Arbeit an mehreren Rechnern, vor jedem Arbeitsschritt ein Pull durchführt und am Ende einer Arbeitsphase sofort einen Push macht.

[1] Bei älteren git-Versionen heißt der Hauptbranch, der hier angezeigt wird, gelegentlich `master`, in diesem Fall muss man sich stets `main` durch `master` ersetzt denken.

[2] Genau genommen zeigt **`main`** auf den aktuellen Stand des Branches mit dem Namen main; arbeitet man mit mehreren Branches, gibt es weitere Zeiger.