GIT WORKSHOP

Wolfgang Höfer

Vorbemerkung

Ein Betrieb von *git* direkt auf Windows ist nicht sinnvoll. Es wird also ein System wie MacOS oder Linux vorausgesetzt.

Dieses Handout ist in folgende Sinnabschnitte unterteilt

- Ein minimaler *git*-Server
- Grundlagen der Arbeit mit *git*
- Installation eines vollwertigen Gitea-Servers

1 Erster Überblick

Eine Einführung in die Arbeit *git* ist etwas schwierig, weil auf einen Schlag so viele neue Konzepte zu verarbeiten sind.

Wir beginnen mit einem Überblick und gehen dann immer mehr in die Feinheiten.

1.1 Lokal – Remote

Du kannst ganz alleine auf deinem Rechner mit *git* arbeiten und hast dann ein eigenes Repository – und zwar ein lokales Repository.

Sind an einem Projekt mehr Leute beteiligt, dann wird im Normalfall auf einem Server ein zentrales Repository erstellt, von dem sich jeder Mitarbeiter eine Kopie als lokales Repository auf den eigenen Rechner holt (clont).

Der Unterschied beim Arbeiten in den beiden Varianten ist, dass die Änderungen der lokalen Repositories immer wieder ins zentrale Repository müssen (push) und
dass sich die anderen Mitarbeiter diese Änderungen auch alle holen müssen (pull).

Natürlich kann es dabei passieren, dass verschiedene Mitarbeiter die gleiche Datei verändern und dann müssen die entstehenden Konflikte *irgendwie* behoben werden.

In diesen wenigen Zeile stecken unwahrscheinlich viele Techniken, die wir im Rahmen dieses Workshops auf keinen Fall alle besprechen können.

1.2 Idealer Ablauf

Beschränken wir uns zuerst auf den lokalen Fall und eine ideale Welt.
In diesem Fall würdest du keine Fehler machen, immer genau wissen was du der Reihe nach machst – perfekte Welt eben.

Dein Arbeitsablauf wäre dann ganz einfach:

- Änderung programmieren
- Änderungen herrichten zum Protokollierung
- Änderung festschreiben im Protokoll

In der Sprache von *git* werden die letzten beiden Schritt als *add* und *commit* bezeichnet.

1.3 Realistischer

Wahrscheinlich musst du ab und zu Dinge ausprobieren und willst dafür deinen funktionierenden Code nicht gefährden.
In diesem Fall erstellst du einen parallelen Zweig (branch), in dem du in Ruhe arbeiten kannst. Wenn du mit deinem Ergebnis zufrieden bist, dann übernimmst du deine Entwicklung – wenn nicht, dann löschst du sie.

Man muss hier aber gleich sagen, dass das eine extreme Vereinfachung der Realität darstellt. Mit Branches gibt es viele Varianten, die zu besprechen wären.

1.4 Remote

Werfen wir noch einen kurzen Blick auf die Entwicklung im Team.
Die Wahrscheinlichkeit für das Auftreten von Konflikten ist hier natürlich viel größer und da ist es sehr hilfreich, dass *git* uns unterstützt. Im Prinzip werden bei der Konfliktlösung – sofern nicht automatisch möglich – beide (alle) Varianten in die betroffene Datei geschrieben werden. Dabei wird jeweils die Herkunft angegeben und der Zuständige für Konfliktlösung kann löschen oder übernehmen, was er für richtig hält.

2 Vorbereitung des lokalen Rechners

2.1 Vorüberlegung

Um mit *git* arbeiten zu können, brauchst du eine geeignete Client-Software auf deinem Computer[1]
Viele Entwicklungsumgebungen bringen ihren individuellen *git*-Client mit, was eine pauschale Behandlung erschwert.

Du solltest zuerst die typischen Abläufe direkt mit *git* einüben, bevor du dich mit grafischen Systemen beschäftigst.

2.2 Installation von Git

Bei der Installation von *git* werden automatisch Client und Server installiert. Du wirst hier aber nur den Client (=*git*-Bash) verwenden.
Dabei handelt es sich um eine extrem abgespeckte Linux-Umgebung, so dass wir auch die Grundbefehle im Terminal besprechen müssen.

Lade von gitforwindows.org die Installationsdatei herunter, beginne aber noch nicht mit der Installation!

Lade von notepad-plus-plus.org den Editor *notepad++* herunter und installiere das Programm. Du wirst bei der Installation von Git nach diesem Programm gefragt!

Starte nun die Installation von *git* durch Doppelklick auf die Installationsdatei. Die nachfolgenden Bilder zeigen den Ablauf in der Version vom Februar 2025.

TODO: Bilder

[1] Diese Aussage ist nur bedingt richtig, weil für manche Szenarien auch eine reine Online-Plattform genügt.

3 Hands On I

Du sollst in diesem *Hands on* die absolut einfachste Variante der Arbeit mit *git* kennenlernen. Davor kommen allerdings einige kleine Hürden in Form der Konfiguration eines lokalen Repositories und auch von *git* als solches.

Du wirst hier – wie im Großteil des Workshops – mit dem Terminal arbeiten, wen es um die *Bedienung* von *git* geht. Wenn du Inhalte in Dateien einfügen sollst, kannst du natürlich einen Editor deiner Wahl verwenden. Ich gebe aber auch immer die Terminal-Variante an, einfach weil es praktischer ist.

TIPP
Im Terminal kannst du mit der *Pfeil hoch* Taste durch die letzten Befehle blättern – damit gehen manche Dinge sehr schnell!

3.1 Konfigurieren von git

Öffne die *git*-Bash in dem Ordner, in dem du arbeiten willst. Das geht über die rechte Maustaste im Explorer (Dateimanager).
Sollte dir die Schrift im Fenter zu klein sein, kannst du sie mit strg + + vergrößern.

Aktuelle Einstellungen ansehen

```
# Eingabe
git config --list

# Ausgabe
user.email=
user.name=
      ...
```

Einstellungen anpassen

```
# eigene Phantasiedaten verwenden
git config --global user.name "Susi Sandmann"
git config --global user.email=susi@sandmann.de
```

3.2 Erste Schritt

Lege nun einen Arbeitsordner für dieses *Hands on* an:

```
git init projekt
cd projekt
```

Du erstellst nun eine erste Datei in diesem Ordner und gibst ihr den Namen *datei1.txt*. Wenn du das nicht über die *git*-Bash machen willst, dann achte darauf, dass du eine reine Textdatei erstellst – also nicht mit Word o.ä. sondern mit einem reinen Editor wie *notepad*, *notepad++* oder einer Entwicklungsumgebung.

Schneller geht das aber mit *git*-Bash.

```
echo "Version 1" >> datei1.txt
```

Diese Datei nimmst du dann in die Versionsverwaltung auf

```
git add datei1.txt
```

Im Anschluss erstellst du einen Commit um den aktuellen Stand deines *Projekts* festzuschreiben.

```
git commit -m "Erste Datei erstellt"
```

Diesen Zyklus aus *add* und *commit* führst du immer wieder aus, sobald du an eine wichtige Stelle im Projektverlauf kommst. Generell gilt: „Lieber mehr Commits, als zu wenige!“

Ein Commit ist ein Projektstand, an den du zurückkehren kannst, wenn etwas grob missglückt ist. Vergeht zu viel Zeit (=viel Code) zwischen zwei Commits, so entstehen größere Verluste.
Später im Workshop lernst du auch, wie du Commits nachträglich zusammenfassen kannst (=squash) um deine Entwicklungsschritte übersichtlich zu halten.

Nun änderst du deine Datei ab, indem du eine Zeile ergänzt:

```
echo "Version 2" >> datei1.txt
git add datei1.txt
git commit -m "Erläuterung"
```

Du kannst gleichzeitig auch mehrere Dateien in die Versionierung aufnehmen und als Projektstand festschreiben

```
# Aktuell passiert nichts - du hast ja nichts geändert!
git add . --all
git commit -m "Erläuterung"
```

TIPP
Das *adden* ist generell unabhängig vom *commit*. Es ist sogar üblich, immer erst einige Dateien anzusammeln, bevor du einen Commit machst.
Es ist auch völlig unproblematisch eine Datei zwischen zwei Commits mehrfach in der aktuellen Version mit *add* für den Commit zu registrieren.

In machen Projekte solltest du von *–all* Abstand nehmen. Vielleicht gibt es Dateien, die du absichtlich *nicht* in der Versionierung haben willst. Das ist oft bei Dateien mit Kennwörtern der Fall.

Mit diesen Befehlen ist bereits ein einfacher lokaler Entwicklungsprozess abbildbar. Allerdings gibt es noch viele weitere Feinheiten und Varianten um eventuelle Fehler zu beheben.

3.3 Ein genauerer Blick

Der Ordner eines Projekts wird von *git* in drei *logische* Bereiche unterteilt:

- Working-Directory (Arbeitsordner)
- Stage (auch Index oder Cache genannt)
- Repository

Leider werden die Begriffe nicht einheitlich verwendet und haben in unterschiedlichem Kontext auch noch abweichende Bedeutung – das betrifft uns hier aber nicht. Sollte eine Möglichkeit zur Verwechslung existieren, weise ich dich ausdrücklich darauf hin.

Mit der *logischen* Unterteilung ist gemeint, dass die unterschiedlichen Versionen der Dateien in einem speziellen Ordner innerhalb des Projektordners verwaltet werden. Der Ordner trägt den Namen `.git` und wird vom System ausgeblendet. Dieser Ordner bildet den *Stage* und das *Repository*. Allerdings bedeutet *logisch* auch, dass diese beiden Bereiche von *git* verwaltet werden und vom unbedarften Benutzer nicht einfach unterschieden werden können.

Betrachten wir einen Entwicklungszyklus im Detail.

Den aktuellen Stand des Projekts kannst du mit folgendem Befehl abfragen:

```
# Eingabe
git status

# Ausgabe (Zeilenumbrüche geändert)
Auf Branch master
nichts zu committen,
Arbeitsverzeichnis unverändert
```

Das bedeutet, dass du aktuelle einen *sauberen* Zustand deines Arbeitsordners ohne Änderungen hast.

Erstelle eine neue *datei2.txt* und füge die Zeile *Version 1* ein. In der Datei *datei1.txt* kommt die Zeile *Version 3* dazu.

```
# Eingabe (Lasse ich zukünftig weg)
echo "Version 3" >> datei1.txt
echo "Version 1" >> datei2.txt
git status

# Ausgabe
Auf Branch master
Änderungen, die nicht zum Commit vorgemerkt sind:
  (benutzen Sie "git add <Datei>...",
   um die Änderungen zum Commit vorzumerken)

  (benutzen Sie "git restore <Datei>...",
   um die Änderungen im Arbeitsverzeichnis zu verwerfen)
```

```
    geändert:       datei1.txt

Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...",
   um die Änderungen zum Commit vorzumerken)

    datei2.txt

keine Änderungen zum Commit vorgemerkt
(benutzen Sie "git add" und/oder "git commit -a")
```

Es wird dir angezeigt, dass Änderungen im *Workingdir* vorliegen.
Du siehst auch, dass *git* die *datei1.txt* bereits kennt (=geändert) und dass *datei2.txt* noch nicht in die
Versionsverwaltung aufgenommen wurde (=Unversioniert).

Nun werden die aktuelle Stände *auf den Stage/in den Stage* übertragen:

```
git add . --all
git status

# Ausgabe
Auf Branch master
Zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git restore --staged <Datei>..."
   zum Entfernen aus der Staging-Area)

    geändert:       datei1.txt
    neue Datei:     datei3.txt
```

Diese Dateien sind zum Commit vorgemerkt, also zum festen Eintrag in die Projekthistorie.
Der Status zeigt aktuell den *Stage* an.

Als Demonstration fügen wir beiden Dateien eine weitere Zeile hinzu und fragen wieder den Staus ab:

```
echo "Weitere Zeile" >> datei1.txt
echo "Weitere Zeile" >> datei2.txt
git status

# Ausgabe ( ---- Anmerkung ----)
Auf Branch master
Zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git restore --staged <Datei>..."
   zum Entfernen aus der Staging-Area)

    geändert:       datei1.txt
    neue Datei:     datei2.txt

--------------- Ende Stage ------------
--------------- Anfang Workdir --------

Änderungen, die nicht zum Commit vorgemerkt sind:
  (benutzen Sie "git add <Datei>...", um die Änderungen
```

```
    zum Commit vorzumerken)

   (benutzen Sie "git restore <Datei>...",
    um die Änderungen im Arbeitsverzeichnis zu verwerfen)

     geändert:       datei1.txt
     geändert:       datei2.txt
```

Beide Dateien liegen also in verschiedenen Versionen im Stage und im *Workingdir* vor. Ein Commit transferiert die Dateien aus dem Stage ins Repository, also in die Projekthistorie.

Hinweis
Wenn ich hier von *transferieren* spreche, so führt das zu einer falschen Vorstellung. In Wirklichkeit werden im *.git*-Ordner nur einige Verweise geändert – wobei das auch wieder nicht die ganze Wahrheit ist!

3.3.1 Etwas mehr Wahrheit

Dieser Abschnitt ist nur ein Einblick für Interessierte. Angenommen, du erstellst ein neues Repository

```
git init demo
cd demo

echo "Etwas HTML" >> homepage.html
```

Wenn du nun die Verzeichnisstruktur des Ordners *.git* genauer untersuchst, wirst du in etwa dies hier sehen:

```
.git/
   branches
   config
   description
   HEAD
   hooks
       applypatch-msg.sample
       commit-msg.sample
       fsmonitor-watchman.sample
       post-update.sample
       pre-applypatch.sample
       pre-commit.sample
       pre-merge-commit.sample
       prepare-commit-msg.sample
       pre-push.sample
       pre-rebase.sample
       pre-receive.sample
       update.sample
   info
       exclude
   objects
```

```
        info
        pack
    refs
        heads
        tags
```

homepage.htm[1] ist noch nicht versioniert, also stellt obige Ordnerstruktur ein leeres Repository dar. Sobald du die Datei zum Commit vormerkst

```
git add index.html
```

ändert sich im Ordner *.git* etwas. Ich gebe nur die relevante Stelle an:

```
    index            <<< neu
    info
        exclude
    objects
        ce          <<< neu, bei dir anders!
            1332fcf1795166c3c5cd0216b5e7dc2dce7998
        info
        pack
```

Die Datei *index* stellt den Stage dar. Diese Datei kannst du nicht direkt lesen. Das geht nur mit

```
git ls-files --stage
```

Den Hash aus dieser Ausgabe findet du oben in der Ordnerstruktur wieder.

```
100644 ce1332fcf1795166c3c5cd0216b5e7dc2dce7998 0   homepage.htm
```

Allerdings wird mit den ersten beiden Zeichen ein Ordner erstellt und in diesem Ordner eine Datei, deren Name aus den restlichen Zeichen besteht. Zusätzlich findet man den Dateityp (100644), den Status (Das ist komplexer! Hier 0) und den Dateinamen.

Den Hash kannst du dir auch so anzeigen lassen:

```
git hash-object homepage.htm
```

Aber gehen wir einen Schritt weiter und committen die Datei:

```
git commit -m "about.html added"
```

Nun sieht die Ordnerstruktur verändert aus – die Hooks habe ich in der Darstellung aus Platzgründen entfernt!

```
.git/
    branches
    COMMIT_EDITMSG
    config
    description
```

[1] Der Name wurde gewählt um keine Verwechslung zwischen index.html und dem Index von *git* zu provozieren!

```
HEAD
hooks
  ....
index
info
    exclude
logs
    HEAD
    refs
        heads
            master
objects
    3b
        028e4d0230fb8f3553f0f7579e68e6c4e27d3f
    94
        eed25c4ef6cfa1384df66f28308c09dc9bf28d
    ce
        1332fcf1795166c3c5cd0216b5e7dc2dce7998
    info
    pack
refs
    heads
        master
    tags
```

Wir konzentrieren uns auf die *objects* und das *log*.

Mit `git log --oneline` erfährst du etwas über den erfolgten Commit – zum Beispiel seinen Hash:

```
commit 3b028e4d0230fb8f3553f0f7579e68e6c4e27d3f (HEAD -> master)
Author: wolfgang <hoeferwolf@t-online.de>
Date:   Sun Jan 26 20:20:10 2025 +0100

    homepage.html added
```

Durch `git cat-file -t 3b028e4d0230fb8f3553f0f7579e68e6c4e27d3f`
siehst du, dass es wirklich ein Commit ist (unnötig),
durch `git cat-file -p 3b028e4d0230fb8f3553f0f7579e68e6c4e27d3f`
gibt es weitere Details:

```
tree 94eed25c4ef6cfa1384df66f28308c09dc9bf28d   << wichtig!
author wolfgang <hoeferwolf@t-online.de> 1737919210 +0100
committer wolfgang <hoeferwolf@t-online.de> 1737919210 +0100

homepage.html added
```

Am Commit hängt also der *tree* der betroffenen Dateien:

```
git cat-file -p 94eed25c4ef6cfa1384df66f28308c09dc9bf28d
```

Diese Ausgabe kennst du bereits aus dem Stage:

```
100644 blob ce1332fcf1795166c3c5cd0216b5e7dc2dce7998    homepage.html
```

und mit

```
git cat-file -p ce1332fcf1795166c3c5cd0216b5e7dc2dce7998
```

bekommst du den Inhalt der Datei:

```
Etwas HTML
```

Du bist jetzt über mehrere Stufen zum Inhalt der Datei gelangt. Eine kleine Stufe haben wir aber ausgelassen:

```
git cat-file -t ce1332fcf1795166c3c5cd0216b5e7dc2dce7998
```

liefert die Information `blob` (Binary large object). Es gibt einen Grund, warum ich dir diesen Umweg zumute! Di siehst hier ein grundlegendes Konzept von *git*, das beim Verständnis hilft:

Der Commit zeigt auf einen Tree und im Tree stehen Dateinamen mit den Hashes.
Das sind nur die Namen und nicht die Dateien! Hinter dem Hash verbirgt sich ein *blob*, also reine *Roh*daten **ohne** Dateinamen. Git verwendet das, um von verschiedenen Stellen aus auf die gleichen Rohdaten zu verweisen. Ändern sich Dateien zwischen zwei Commits nicht (Hash bleibt gleich), so werden sie nicht neu gespeichert. Der neue Commit zeigt auf einen neuen Tree in dem wieder die gleichen Hashes mit den gleichen Dateinamen stehen.

Tiefer will ich an dieser Stelle in dieses Thema nicht einsteigen!

3.4 Fehlerbehebung

Ein Versionsverwaltungsprogramm hat unter anderem die Aufgabe, ein Projekt in gewissem Umfang revisionssicher zu protokollieren. Aus diesem Grund müssen bestimmte *Fehler* in der History erhalten bleiben.
Bei anderen Fehlern ist es aber wichtig, dass sie schnell und einfach rückgängig gemacht werden können.

Relativ unproblematisch sind *Reparaturen* bei lokaler Arbeit, bevor ein Commit erfolgt ist:

- Neue, fehlerhafte Dateien kann man einfach löschen.
- Bereits versionierte Dateien mit Fehlern können durch `git restore <dateiname>` wieder mit der Version aus dem Stage überschrieben werden[2]

Vorher habe ich geschrieben, dass bei einem Commit die Dateien aus dem Stage in die Projekthistorie übertragen und aus dem Stage entfernt werden. Das stimmt so nicht ganz.

In Wirklichkeit zeigt `git status` diese Dateien nur nicht mehr an, was auf einen leeren Stage schließen lassen würde.

Wäre der Stage wirklich leer, so könnte nachfolgende Reparatur nicht funktionieren – was sie aber tut:

[2] Es gibt auch noch den Befehl `git restore <dateiname> --staged`, der das *Workingdir* unangetastet lässt und nur die Version aus dem Stage löscht.

```
echo "text" > datei.txt
git add datei.txt     # danach im Stage
git commit -m "gesichert"

git status   # Stage anscheinend leer

echo "falscher Text" >> datei.txt   #  Fehler
git restore datei.txt              # repariert
```

Hinweis
Im Netz findet man oft die Variante mit `checkout`, die früher üblich war. `restore` wurde eingeführt um eindeutigere Befehle zu schaffen.

Mit dem `restore`-Befehl kann man auch eine Datei aus einem beliebigen Commit restaurieren:

```
git restore --source=<HASH> <datei>
```

Die Fehlerbehebung kann auch das Löschen ganzer Entwicklungs- oder Testzweige bedeutet, was später noch besprochen wird.

4 Hands on II

Du wirst in diesem Abschnitt einige Dinge als redundant empfinden, weil sie auch bereits im ersten *Hands On* bereits verwendet wurden. Sieh es einfach als Übung und da du dieses PDF ohnehin nicht ausdrucken wirst, ist die minimal erhöhte Seitenzahl auch kein Problem.

Dieses Szenario ist etwas komplexer, da wir mit mehreren Dateien arbeiten und auch parallele Ideen verfolgen wollen (branching).
Da Programmieren zu lange dauert, erstellen wir in diesem Projekt ein kleines Buches mit Kurzgeschichten. Die Geschichten wurden von einer KI generiert um den Arbeitsaufwand zu minimieren.

Die Einzelteile der Geschichten befinden sich in einem *git*-Repository und müssen zunächst auf den lokalen Computer geclont werden. Das ist ein notwendiger Vorgriff auf den Abschnitt der Teamarbeit, den du hier einfach mal ausführst. Es handelt sich um ein öffentliches Repository, auf das auch ohne Benutzerkennung zugegriffen werden darf – aber eben nur lesend. Aus diesem Grund wird zum Clonen auch *https* und nicht *ssh* verwendet. Öffne die *git*-Bash (oder ein Terminal) und gib ein:

```
mkdir fortbildung
cd fortbildung
git clone https://TODO # Ziel: vorlage.git
cd vorlage
```

4.1 Labor

Es dürfte am einfachsten sein, wenn du zwei Terminals öffnest:

- Eines für den Vorlagenordner
- Eines für deinen Arbeitsordner

Du wirst ständig zwischen diesen Repositories wechseln müssen und da sind einzelne Fenster optimal (Wechsel z.B. mit alt + tab).

Um der Erklärungen besser folgen zu können, wäre es sinnvoll, wenn bei jedem Teilnehmer exakt die gleichen Hashwerte wie hier in der Anleitung entstehen würden. Leider ist das nicht so einfach, weil die Hashwerte nicht reproduzierbar sind – zumindest auf meinem Rechner! Aus diesem Grund wurden im Vorlagenverzeichnis die Commits getagged, so dass du über ihre Namen auf sie zugreifen kannst – wie das geht, siehst du gleich!

Erstellen eines Repositories

Öffne im Fortbilfungsordner eine *git*-Bash und erstelle ein Repository für das Buch.

```
# Eingaben
cd
cd fortbildung
git init buch
cd buch
```

Je nach Version von *git* kommt hier eine längere Ausgabe … oder auch nicht.

Im Ordner *fortbildungen* sollten nun also die Repositories `vorlage` und `buch` vorliegen.

Erste Datei anlegen
Wechsle in das Fenster mit dem Vorlagenordner und Checke *Step_1* aus. *Auschecken* wird später noch genauer erklärt!

```
git checkout Step_1
```

Kopiere die Datei *story1.md* in den Ordner *buch*.
Sie enthält (strategisch) nur einen Teil der Geschichte! Das Kopieren kann auch gerne mit der Maus im Explorer stattfinden. Wer allerdings nicht ständig zwischen Tastatur und Maus wechseln möchte, der kann auch folgenden Befehl verwenden:

```
# aktueller Ort ist das vorlagen-Fenster
cp story1.md  ../buch/
```

Wechsle in das Fortbildungsfenster

```
# Eingabe
git status

# Ausgabe
Auf Branch main                                                  ①

Noch keine Commits

Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...",
   um die Änderungen zum Commit vorzumerken)
    story1.md                                                    ②

nichts zum Commit vorgemerkt, aber es gibt unversionierte Dateien
(benutzen Sie "git add" zum Versionieren)                        ③
```

① Geänderter Name ist ok.
② Diese Datei ist betroffen, hier kann eine längere Liste stehen.
③ Empfohlener Befehl `git add story1.md`

Übertragen in den Stage

```
# Eingabe
git add story1.md
git status

# Ausgabe
Auf Branch main
```

```
Noch keine Commits

Zum Commit vorgemerkte Änderungen:                                  ①
  (benutzen Sie "git rm --cached <Datei>..."
   zum Entfernen aus der Staging-Area)
    neue Datei:     story1.md                                       ②
```

① Die Datei liegt auf dem Stage, bereit zum Commit
② Nur diese eine Datei.

Der Hinweis `git rm --cached <Datei>` bewirkt nicht das Löschen der Datei aus dem Arbeitsverzeichnis, sondern aus dem Stage!

4.1.1 Labor

Der erste Teil von *story1.md* ist nun in die Versionierung aufgenommen, es wurde aber noch kein protokollierter Punkt in der Projektgeschichte (=commit) erstellt.

Du schreibst an deiner Geschichte weiter und damit ändert sich der Stand zwischen Arbeitsverzeichnis und Stage. Für dich bedeutet das nun wieder foglende Schritte:

- Wechsle in den Vorlagenordner
- Richtigen Stand auschecken: `git checkout Step_2`
- Kopieren der Datei: `cp story1.md ../buch/`
- Status checken: `git status`

```
# Ausgabe
Auf Branch main

Noch keine Commits

Zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git rm --cached <Datei>..."
   zum Entfernen aus der Staging-Area)
    neue Datei:     story1.md                                       ①

Änderungen, die nicht zum Commit vorgemerkt sind:
  (benutzen Sie "git add <Datei>...",
   um die Änderungen zum Commit vorzumerken)
  (benutzen Sie "git restore <Datei>...", um die Änderungen im
   Arbeitsverzeichnis zu verwerfen)
    geändert:       story1.md                                       ②
```

① Die alte Version der Datei ist zum Commit vorgemerkt
② Die neuere Version ist noch nicht im Stage!

Wieder werden mögliche Befehle angegeben, die in diesem Zustand sinnvoll sein können!

Hinweis
Wenn du versuchst, die Datei mit `git rm --cached story1.md` aus dem Stage zu löschen, bekommst du eine Fehlermeldung. Es könnte nämlich sein dass der Vorgang zu

Datenverlust führt!

```
error: die folgende Datei hat zum Commit vorgemerkte Änderungen
      unterschiedlich zu der Datei und HEAD:
         story1.md

(benutze *-f*, um die Löschung zu erzwingen)
```

Bei `git restore story1.md` gibt es keine solche Warnung! Die Datei verbleibt dabei auch im Stage unverändert erhalten.

Nun kannst du erstmalig eine Differenz zwischen den Versionen der Datei erkennen:

```
git diff story1.md
```

4.1.2 Analyse der Differenz

TODO: Neu aufbauen

```
diff --git a/story1.md b/story1.md                              ①
index 9c818e7..3126a18 100644                                   ②
--- a/story1.md                                                 ③
+++ b/story1.md
@@ -25,3 +25,63 @@                                            ④
entdeckte sie den geheimnisvollen Tunnel.                       ⑤
  der gerade von einem seiner Flüge zurückgekehrt               ⑥
  war und sich auf dem Ast des Apfelbaums niederließ.
  +„Ich glaube, es ist ein Tunnel!" miaute Luna                 ⑦
  +aufgeregt. „Vielleicht führt er zu einem verborgenen
```

① Welche Dateien werden verglichen? `a/` und `b/` sind nur Kennzeichner der Dateien. `a/` ist die ältere (z.B. Stage) und `b/` die neuere Datei (z.B. Workingdir).
② Bei 9c818e7 und 3126a18 handelt es sich um die Hash-Ids der beiden Dateien, die verglichen werden (dazu unten mehr).
③ Soll erneut klarstellen `---` ist alt, `+++` ist neu.
④ Die Zeile heißt *Hunk Header*. Er beginnt und endet mit `@@`
⑤ In diesen Zeilen gab es keine Änderungen
⑥ Das ist vor der Änderung.
⑦ Diese Zeile steht in *b*.

Leider ist das Format vom *Hunk Header* etwas kryptisch! Er gibt immer an, in welcher Zeile der Vergleich beginnt und wie viele Zeilen betroffen sind (vorher und nachher). Bei `@@ -3,12 +3,14 @@` beginnt der relevante Bereich ab Zeile 3 und in der Ursprünglichen Version lagen 12 Zeilen vor, in der neuen Version sind es jetzt 14. Die Änderungen im Detail folgen in den nächsten Zeileen.

4.1.3 Hands on

Nun wird die Datei wieder zum Stage hinzugefügt und dann commitet:

Eingabe

```
git add story1.md
git commit -m 'Guter Ansatz'
```

TODO: Hash aktualisieren

Ausgabe

```
[main (Root-Commit) 5229c6b] Guter Ansatz
 1 file changed, 4 insertions(+)
 create mode 100644 story1.md
```

4.1.4 Labor

Dieser Arbeitszyklus von *add, status, diff, commit* stellt den Kern der Arbeit mit *git* dar, wenn man alleine arbeitet. Allerdings gibt es noch weitere, sehr wichtige Techniken, die auch später bei der Arbeit im Team eine große Rolle spielen werden.

Du hast nun *story1.md* so weit fertig gestellt und möchtest an anderen Kurzgeschichten arbeiten. Du beschließt, dass fertige Geschichten und Arbeitesversionen an unterschiedlichen *Orten* aufgehoben werden sollen.

In der Softwareentwicklung sprechen wir hier von *fertigen Produkten*, die an den Kunden ausgeliefert werden und von *Experimenten, Features, Bugfixes, …*, die parallel entwickelt werden müssen. Durch ein Experiment darf die aktuell lauffähige Version Nicht beeinträchtigt werden! Du brauchst einen *Entwicklungszweig* und einen *Auslieferungszweig*.

```
# Zweig erstellen
git switch -c entwicklung_story_2
```

Git teilt dir mit, dass du dich nun in diesem *Zweig* (=Branch) befindest. Du siehst hier immer noch deine aktuelle Story 1! Sie ist ebenfalls in dem Branch enthaltne, du hast aber nicht vor, an ihr zu arbeiten. Das wäre zwar möglich, würde aber mein Szenario hier stören!

Nun beginnst du mit Story 2, indem du sie aus dem Vorlagenordner kopierst.

- Wechsle in den Vorlagenordner
- Richtigen Stand auschecken: `git checkout Step_3`
- Kopieren der Datei: `cp story2.md ../buch/`
- Status checken: `git status`

Du bist mit deinem bisherigen Werk ganz zufrieden, hast aber zwei Ideen, wie man du weiter machen könntest. Aus diesem Grund erstellst du zunächst einen Commit

```
git add story2.md
git commit -m "Story 2 begonnen"
```

und legst für jede Idee einen Branch an:

```
git switch -c story2_idee_1
git switch entwicklung_story_2
git switch -c story2_idee_2
```

Nun nicht den Überblick verlieren! Im Branch *story2_idee_1* arbeitest du an deiner Idee 1 von *story2.md* weiter, im Branch *story2_idee_2* probierst du einen alternativen Verlauf.

Für dich heißt das:

- Im Fortbildungsordner: `git switch story_2_idee_1`
- Wechsle in den Vorlagenordner
- Richten Stand auschecken: `git checkout Step_4`
- Kopieren der Datei: `cp story2.md ../buch/`
- `git add story2.md`
- `git commit -m "Inhalt nach Idee1 fortgeführt"`

Da du aber noch nicht sicher bist, ob diese Version verwenden willst, arbeitest auch Idee 2 aus.

- Im Fortbildungsordner: `git switch story_2_idee_2`
- Wechsle in den Vorlagenordner
- Richten Stand auschecken: `git checkout Step_5`
- Kopieren der Datei: `cp story2.md ../buch/`
- `git add story2.md`
- `git commit -m "Inhalt nach Idee 2 fortgeführt"`

Nun hast du das *Problem*, dass du drei Versionen hast:

1. Version 1 im Entwicklungszweig.
   Sie ist die Stufe vor der Neubearbeitung.
2. Version 2 im Branch für Idee 1
3. Version 3 im Branch für Idee 2

Du bist unentschieden und siehst dir die 3 Versionen noch einmal in Ruhe an.

```
git switch story2_idee_1        # in Ruhe durchlesen
git switch story2_idee_2        # in Ruhe durchlesen
git switch entwicklung_story_2  # in Ruhe durchlesen
```

Dabei fällt dir ein *grober Rechtschreibfehler* im Entwicklungszweig auf, den du sofort korrigierst. Bei uns ist es das Wort *weiße*, das sich anstelle von *weise* in den Text geschummelt hat.

- In den Vorlagenordner
- `git checkout Step_6`
- `cp story2.md ../buch/`
- Im Fortbildungsordner
- `git add story2.md`
- `git commit -m "typo 'weiße' korrigiert"`

Du musst dich nun entscheiden, mit welcher Version du dein Projekt fortführen willst. In Wirklichkeit werden diese Branches natürlich deutlich länger sein, das Prinzip bleibt aber gleich!

Du entscheidest dich, dass du Idee 1 weiter verwenden willst und möchtest sie in den Entwicklungszweig übernehmen. In diesem Zweig solltest du aktuell gerade sein, überprüfst das aber:

```
git branch
# falls nötig
git switch entwicklung_story_2
```

Aus Sicherheitsgründen vergleichst du die Versionen kurz miteinander:

```
git diff --color-words entwicklung_story_2..story2_idee_1
```

Du siehst den Austausch des Wortes und den neu geschriebenen Text. Für *git* ist das problemlos!

Nun führst du die Zweige zusammen:

```
git merge story2_idee_1  # erster Merge
```

git erwartet von dir eine Begründung für den Merge. Gib also *Vorläufig ok* ein, speichere und schließe den Editor.

Du bist erfreut, dass dieser Merge problemlos funktioniert.

Allerdings bist du am Überlegen, ob das mit der Schildkröte so eine gute Idee ist. Du entscheidest dich, eine *Riesen*schildkröte daraus zu machen. Aus Sicherheitsgründen willst du das aber zuerst im Ideen-Zweig ändern!

Routinemäßig machst du vorher einen *diff*.

```
git diff --color-words entwicklung_story_2..story2_idee_1
```

Beachte, dass du aktuell die Dateiversionen vergleichst, die in den Commits stehen!

Den Unterschied *weise/weiße* siehst du immer noch, der ergänzte Text ist nun ja in beiden Dateien gleich.

Du wechselst in den Ideen-Branch und führst die Änderung durch:

```
git switch story2_idee_1
sed -i -e "s/Schildkröte/Riesenschildkröte/g" story2.md
```

Zur Kontrolle machst du den Diff erneut:

```
# Vergleich der Banches - also das was in den letzen Commits steht
git diff --color-words entwicklung_story_2..story2_idee_1
```

Seltsam … du siehst die Riesenschildkröte nicht im Diff. Aber eigentlich ist das klar, denn es werden ja die Versionen *in den Commits* verglichen und diese neue Version ist noch nicht im Commit. Momentan haben wir also die Situation:

```
# im Branch story2_idee_1
Work:   Riesenschildkröte
Stage:  LEER
Commit: Schildkröte
```

Ein einfaches

```
git diff
```

vergleicht die *aktuelle* Version im Workingdir gegen den Commit, weil der Stage aktuell leer ist (siehe folgender Abschnitt)! Hier siehst du die Riesenschildkröte und die einfache Schildkröte.

4.1.5 Genauerer Blick auf Diff

Diese Diffs sind verwirrend, betrachten wir einfachere Beispiele

Variante 1:
Im Commit, dem Stage und dem Workingdir befinden sich (unterschiedliche) Versionen einer Datei:

```
git init labor
cd labor
echo "Version commit" > datei.txt
git add datei.txt
git commit -m "Version 1"

echo "Version stage" > datei.txt
git add datei.txt

echo "Version work" > datei.txt
```

Nun wollen wir die einzelnen Versionen miteinander vergleichen. Das sind 3 Befehle:

1. work mit *stage*
   `git diff datei.txt`
   (DAS wird wichtig)
2. stage mit commit
   `git diff --cached datei.txt` oder
   `git diff --staged datei.txt`
3. work mit commit
   Wenn man den Hash (z.B. 1234) vom Commit kennt, z.B. durch
   `git log`, so kann man diesen verwenden
   `git diff 1234 datei.txt`
   Universeller ist `git diff HEAD datei.txt`

Variante 2:
Im Commit und dem Workingdir befinden sich (unterschiedliche) Versionen einer Datei. *Der Stage ist aktuell leer.*

```
git init labor
cd labor
echo "Version commit" > datei.txt
git add datei.txt
git commit -m "Version 1"

# auskommentiert
# echo "Version stage" > datei.txt
# git add datei.txt

echo "Version work" > datei.txt
```

1. work mit *commit*
   `git diff datei.txt`
   (Weil der Stage leer ist)
2. stage mit commit
   `git diff --cached datei.txt` oder
   `git diff --staged datei.txt`
3. work mit commit
   Wenn man den Hash (z.B. 1234) vom Commit kennt, z.B. durch
   `git log`, so kann man diesen verwenden
   `git diff 1234 datei.txt`
   Universeller ist `git diff HEAD datei.txt`

Hier sind die Fälle 1) und 3) also identisch.
Wie kann man also sicher sein, welche Dateien ein reines `git diff` vergleicht?
Liefert `git status` einen leeren Stage, so wird mit dem Commit verglichen. Ist eine Datei im Stage, so wird gegen diese Datei verglichen.

4.1.6 Labor

Die Version mit der Riesenschildkröte willst du nun wieder in den Entwicklungsstand mergen. Dafür musst du vorher wieder einen Commit machen.

```
git add story2.md
git commit -m "Riesenschildkröte statt Schildkröte"
```

Du machst einen erneuten Vergleich der Branches mit Diff

```
# Wieder Branch-Vergleich
git diff --color-words entwicklung_story_2..story2_idee_1
```

Da du den Schreibfehler ja nur im Entwicklungszweig verbessert hast, siehst du folgende Situation:

Die *weiße Riesenschildkröte* (Schreibfehler aber richtiges Tier) im Ideen-Branch und *weise Schildkröte* (Richtige Schreibung aber falsches Tier) im Entwicklungs-Branch.
In dieser Situation ist *git* überfordert. Jeder der beiden Texte ist falsch und deshalb musst du als Benutzer entscheiden, was am Ende in der Datei stehen soll – aber das sagt dir *git* beim Merge!

```
git switch entwicklung_story_2
git merge story2_idee_1
```

Nun siehst du eine fette Fehlermeldung, weil die Versionen nicht mehr zusammenpassen! Gerade für Anfänger ist das eine unheimliche Situation, in der man anscheinend viel kaputt machen kann … kann man ja auch. Es ist aber halb so tragisch!

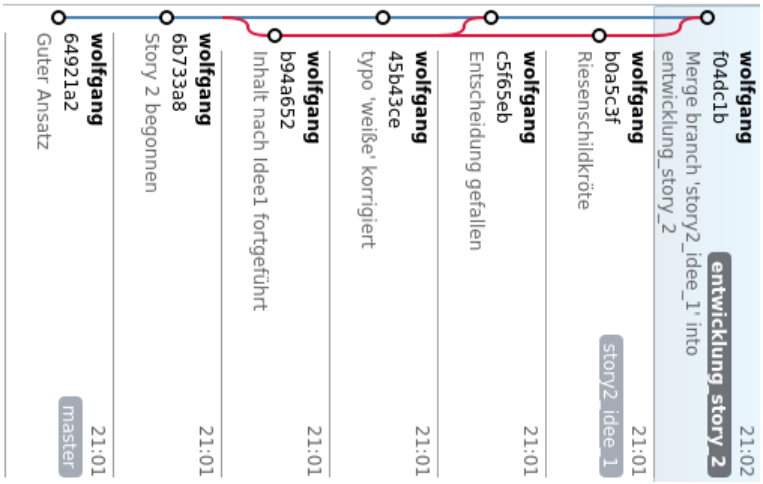
Öffne die entsprechende Datei im Editor. Du erkennst sofort die Stelle, an der Änderungen erfolgt sind:

```
Freddy war nicht allein. Seine beste Freundin,
<<<<<<< HEAD
Shelly, die weise Schildkröte, lebte ebenfalls am
=======
```

```
Shelly, die weiße Riesenschildkröte, lebte ebenfalls am
>>>>>>> story2_idee_1
Teich. Shelly war für ihre ruhige und besonnene
```

Lösche einfach die Markierungen und korrigiere den Text passend.
Wieder ein *add* und ein *commit* und das Problem ist vom Tisch!

Das nachfolgende Diagramm zeigt den Ablauf der Operationen zwischen den beiden Zweigen (Idee 2 wurde ausgeblendet, weil du mit ihr bisher nichts mehr gemacht hast).



Gut - das war ein langer Weg mit vielen Abzweigungen. Es ist normal, diesen Abschnitt mehrfach lesen zu müssen!

4.1.7 Fazit

Branches erlauben gefahrloses Arbeiten an Ideen. Baut man die Szenarien nicht zu komplex auf, dann gibt auch meist keine Probleme beim Merge.

Immer wenn du Änderungen in einem Zweig vornimmst, die du auch im anderen Branch menötigst (weiße/weise) kannst du den Merge auch in die andere Richtung ausführen. Das ist ein übliches Verfahren weil man als Entwickler immer auf dem aktuellen Stand des Auslieferungszweigs sein muss, auch wenn er weiter an seinem Experiment arbeitet.

Die Vergleiche mit *diff* sind definitiv gewöhnungsbedürftig. Machst du häufige Commits auch schon bei kleinen Änderungen, so bekommst du auch übersichtliche *diffs*.

```
git diff <commit>..<commit> <datei>
```

Der Nachteil von vielen Commits sind allerdings … viele Commits. Es gibt allerdings Möglichkeiten nachträglich Commits zusammenzufassen um die Entwicklungsgeschichte zu verschlanken. Diesen Vorgang nennt man *squashen*.

5 Merge, Rebase, Squash

Im letzten Abschnitt hast du einfaches Arbeiten mit Branches gelernt und wie man sie mit Hilfe eines *Merge* zusammengeführt. Das wollen wir uns jetzt genauer ansehen. Offen geblieben sind aber folgende Fragen:

- Was passiert mit alten Branches?
- Was mache ich mit zu vielen Commits?
- Wie sieht die Alternative zu *Merge* aus?

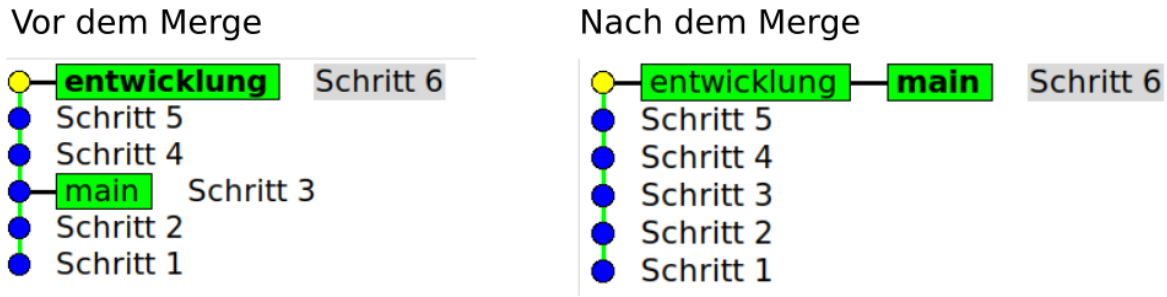
5.1 Verschiedene Merges

In der Praxis treten[1] meist zwei Fälle auf, da ich in der Regel alleine entwickle:

- Während ich am Entwicklungs-Branch arbeite, passiert auf dem Auslieferungsbranch nichts.
- Ich entwickle an einem völlig anderen Feature, das den Auslieferungsbranch gar nicht betrifft. Weil das oft aus Zeitgründen länger dauert, kann es im Auslieferungsbranch Änderungen geben, die im Entwicklungsbranch nicht interessieren.

5.1.1 Fast-Forward

Ohne Änderung im Auslieferungsbranch wird der der Entwicklungsbranch einfach (schnell davor) vor den letzten Commit des Auslieferungsbranchs gesetzt. In dieser Situation wird im Diagramm der Branch nicht einmal als visuelle Abzweigung dargestellt:

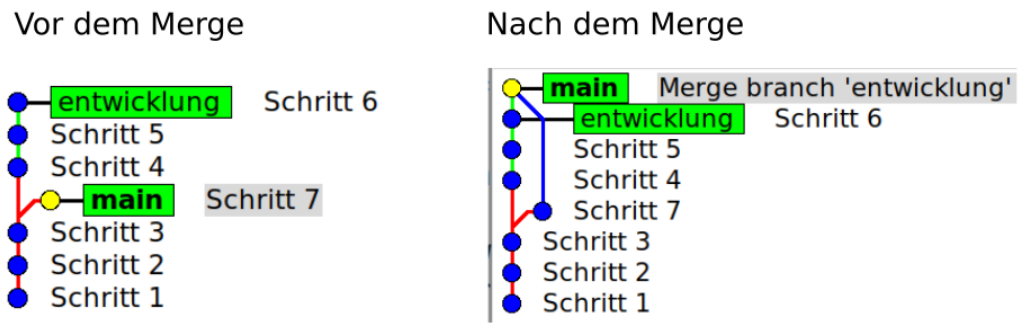


Erkennbar sind die grünen Rechtecke, die die Enden der Branches kennzeichnen. Im Bild links sind die Branches getrennt, im Bild rechts sind beide Branch-Enden nach dem Merge an der gleichen Stelle.
Wie du siehst, hat sich beim Merge die Anzahl Commits nicht verändert.

[1] Bei mir

5.1.2 Drei-Wege Merge

Wenn sich der Auslieferungsbranch parallel zum Entwicklungsbranch weiterentwickelt, sieht die Sitation ganz anders aus!



Du siehst, dass sich der *main*-Branch zu *Schritt 7* weiterentwickelt hat. Beim Merge entsteht ein neuer Commit, der die Spitze beider Branches darstellt. Du erkennst auch, dass im Diagramm die grünen Spitzen der Branches nicht am gleichen Commit sitzen!

Durch `git log --oneline --decorate --graph` (merken als „git log dog“) wird diese Sitation auch passend dargestellt:

```
*   70c1b21 (HEAD -> main) Merge branch 'entwicklung' into main
|\
| * 4a6e3a5 (entwicklung) Schritt 6
| * edaeaff Schritt 5
| * 508019b Schritt 4
* | 2ff4a45 Schritt 7
|/
* 29d2bd4 Schritt 3
* 964ee6f Schritt 2
* 66dcd4f Schritt 1
```

5.2 Alte Branches

Die Commits eines Branches stellen eine Dokumentation des Projektverlaufes dar, die für das Verständnis der erfolgten Änderungen eventuell notwendig sind. Zu viele Branches und Commits erschweren auf der anderen Seite den Überblick.

Es geht also darum, alte Branches zu löschen, die *wichtigen* Commits aber zu behalten. Für den schulischen ist das aber weniger relevant.

Um einen lokalen Branch zu löschen, gibt es folgenden Befehl:

```
git branch -d <name>
```

Er scheitert aber, wenn *git* den Branch noch nicht als abgeschlossen betrachtet.

Probieren wir das beim letzten Szenario aus. Sieh dir die Abbildung oben nich einmal an und dann lösche den Branche

```
git branch -d entwicklung
```

Du siehst eine minimale Änderung

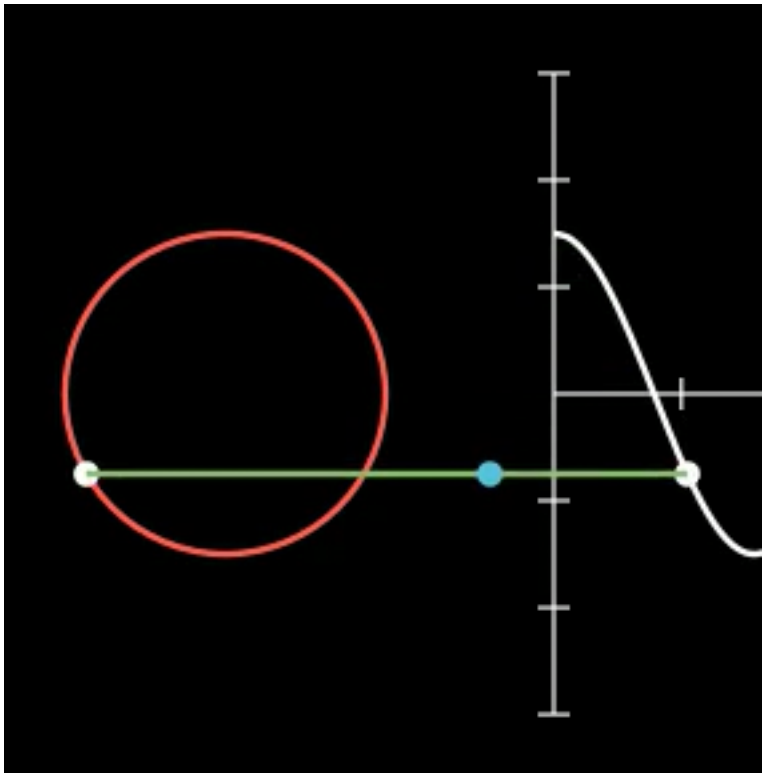
```
*   70c1b21 (HEAD -> main) Merge branch 'entwicklung' into main
|\
| * 4a6e3a5 Schritt 6   <<< hier fehlt (entwicklung)
| * edaeaff Schritt 5
| * 508019b Schritt 4
* | 2ff4a45 Schritt 7
|/
* 29d2bd4 Schritt 3
* 964ee6f Schritt 2
* 66dcd4f Schritt 1
```

Das *Etikett* vom Entwicklungsbranch wurde gelöscht, die Commits bleiben aber in exakt der gleichen Anordnung erhalten! Der Branch ist nicht mehr über seinen Namen zugänglich, die History ist aber weiterhin vorhanden.

5.3 Rebase

Neben dem *Merge* trifft man auch oft auf den *Rebase* um Branches zusammenzuführen. Er funktioniert allerdings deutlich anders und ist wegen seiner vielfältigen Möglichkeiten (z.B. Reihenfolge der Commits ändern, …) ein Werkzeug für fortgeschrittene Benutzer. Mit seiner Hilfe können auch mehrere Commits zusammengefasst werden, um die Branches zu verkürzen. Gerade bei der Zusammenarbeit im Team kann das aber sehr problematisch werden, wenn man nicht genau weiß, was man macht. Das liegt daran, dass sich die Hashwerte von Commits ändern können und wenn sich ein anderer Mitarbeiter den früheren Stand kopiert hat, dann kann das zu großem Chaos führen.

https://www.youtube.com/watch?v=CtyLg10aHN0 https://www.youtube.com/watch?v=1TNK-OkaeII



Bei einem Rebase wird der entsprechende Branch *verpflanzt* – d.h. gewissermaßen *ausgerupft* und an anderer Stelle wieder *angedockt*.

Bei diesem Vorgang werden die Hashwerte aller Commits im Branch geändert.

```
* 10f8d8d (HEAD -> arbeit) Neuer Inhalt 10
* e0ae3ce Neuer Inhalt 9
* 1eac6f3 Neuer Inhalt 8
* fa76537 Neuer Inhalt 7
* da22859 Neuer Inhalt 6
* 7903aaf Neuer Inhalt 5
* 33428f1 Neuer Inhalt 4
* 80db22a Neuer Inhalt 3
* 4a642a8 Neuer Inhalt 2
* a3645f3 Neuer Inhalt 1
| * 2c58be7 (main) Zwischenstopp
|/
* 7fea00c Start
```

Du siehst einen Branch *arbeit* mit mehreren Commits und einen neuen Commit im *main*-Branch.

Wenn du nun vom Branch *arbeit* einen Rebase auf den Branch *main* ausführst:

```
git switch arbeit
git rebase main
```

dann sieht das Ergebnis so aus:

```
* 1cad272 (HEAD -> arbeit) Neuer Inhalt 10
* 130c5b0 Neuer Inhalt 9
* 5b84ab1 Neuer Inhalt 8
* 56d74ea Neuer Inhalt 7
* 9255c1d Neuer Inhalt 6
* 412fa37 Neuer Inhalt 5
* cb91e55 Neuer Inhalt 4
* 3ab37b7 Neuer Inhalt 3
* dc4b1b0 Neuer Inhalt 2
* 9591035 Neuer Inhalt 1
* 2c58be7 (main) Zwischenstopp
* 7fea00c Start
```

Die Verzeigung ist verschwunden und der Branch *arbeit* hängt mit seinen Commits und neuen Hashwerten
am Commit mit dem Hash 2c58be7 aus dem Branch *main*.

Auch wenn es nicht so wirkt: Der Branch *arbeit* ist immer noch vorhanden! Das kannst du auch einfach testen, indem du auf den *main*-Branch wechselst und dort einen neuen Commit erstellst:

```
git switch main
echo "kontrolle" > kontrolle.txt
git add kontrolle.txt
git commit -m "Kontrolle"
```

Das Log zeigt dann

```
* 3508cce (HEAD -> main) kkkddd
| * f4ba9f7 (arbeit) kkk
| * 1cad272 Neuer Inhalt 10
....
| * dc4b1b0 Neuer Inhalt 2
| * 9591035 Neuer Inhalt 1
|/
* 2c58be7 Zwischenstopp
* 7fea00c Start
```

Die Branches laufen also wieder auseinander. Möchte man den Branch *arbeit* loswerden, dann würdest du hier eher den Rebase von *main* auf *arbeit* ausführen. *git* versucht dann, den *main*-Branch hinter den Branch *arbeit* zu hängen. Aus dieser Situation heraus kannst du dann auch den Branch *arbeit* löschen:

```
git branch -d arbeit
```

6 Arbeiten im Team

6.1 Herausforderungen

Das Arbeiten im Team stellt dich vor zwei Probleme

- Welche Server-Infrastruktur sollst du verwenden?
- Was ändert sich am Workflow in *git*

Beide Punkte sollen nachfolgend besprochen werden, wobei es zwangsläufig Überschneidungen geben wird.

6.1.1 Infrastruktur

Im Kern ist es egal ob wir vom Setup im Klassenzimmer oder im Internet sprechen – sobald ein Netzwerk beteiligt ist, sind höchstens Sicherheitsaspekte ein Thema, die Basistechnik bleibt gleich.

Für eine servbasierende Teamarbeit gibt es folgende Varianten

- einfacher *git*-Server
  Die Installation ist einfach, die Grundkonfiguration auch. Der Zugriff erfolgt über ssh. Soll das ohne Kennwort funktionieren, so müssen Schlüssel erzeugt und auf den Server eingepflegt werden. Sspäter führt um die Schlüssel aber kein Weg herum.
- *git*-Server mit kennwortlosem Zgriff über http.
  Funktioniert, ist aber keinesfalls für den Live-Einsatz über einen längeren Zeitraum im Internet zu empfehlen. Die entsprechenden Docker-Files befinden sich in den Materialien. (Basis: ynohat)
- Ein *git*-Server mit *Gitea*
  Gui-basiertes System mit vielen Optionen und weniger komplex als *GitLab*
- Ein *git*-Server mit *GitLab*

Bei allen Varianten ist im Prinzip eine direkte Installation auf den Server (nicht empfohlen) oder der Einsatz von Docker (sinnvoll) möglich. Die Anleitungen für die einzelnen Varianten befinden sich weiter hinten im Script.

6.1.2 Überblick

Bei der Arbeit im Team ändert sich beim lokalen Arbeiten zunächst wenig. Die Schritte *branch, add, commit, …* funktionieren wie auch bisher. Neu sind allerdings die Schritte

- Initiales Holen des Repos auf den eigenen Rechner (=clone)
- Regelmäßiges Abrufen des aktuellen Standes (=pull)
- Veröffentlichen des eigenen Standes (push)

Hierbei sind einige Spielregeln zu beachten, damit es nicht zu Problemen kommt. Diese Regeln werden meist als Workflow oder im Speziellen auch als GitFlow bezeichnet. Es gibt hier zwar gewisse – aber keinesfalls verbindliche – Standards. Jede Firma stellt hier ihre eigenen, fest vorgeschriebenenen Abläufe auf, an die sie die Mitarbeiter zu halten haben.

6.1.2.1 Gemeinsamkeiten

Auch diese Gemeinsamkeiten müssen nicht in jeder Firma oder jedem Projekt umgesetzt werden!

Verantwortung
Im Projekt gibt es einen *main*, der aber durch entsprechende Maßnahmen für *einfache* Mitarbeiter schreibgeschützt ist. Änderungen dürfen nur nach gründlichen Tests und Code-Reviews in diesen Zweig aufgenommen werden. In der Regel gilt hier auch ein *Mehr-Augen-Prinzip* oder noch weiter gestaffelte Zuständigkeitshierarchien.

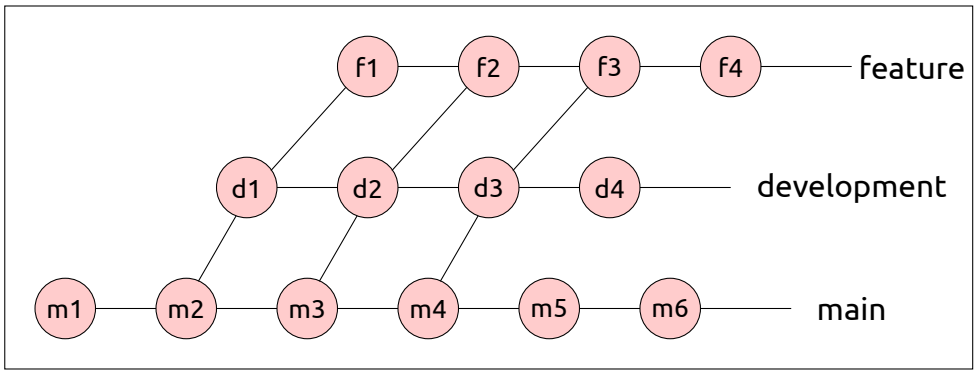
Arbeitsbranch
Je nach Größe des Projekt-Teams gibt es einen oder mehrere *development*-Branches. Die Mitarbeiter holen sich diesen Zweig und lassen von ihm ihre eigenen Arbeitszweige (Feature-Branches, Bugfix-Branches) ausgehen. Ist ihre Arbeit dort beendet, muss der Code getestet werden und dann erst erfolgt der Merge in den *development*-Branch.

6.1.2.2 Details

Da jetzt mehr Personen Änderungen zu unvorhersehbaren Zeitpunkten Änderungen am Code vornehmen, muss jeder Mitarbeiter dafür sorgen, immer die aktuellste Version vorliegen zu haben. Da er seine Entwicklung aber in seinem eigenen *Feature*-Branch vorantreibt, müssen diese Änderungen dort aber erst ankommen – dies geschieht durch Merges in die andere Richtung!

Beispiel

Aus Platzgründen zeichne ich das Branch-Diagramm hier waagrecht. Der *feature*-Branch ist auch noch nicht so weit fortgeschritten, als dass ein Merge auf den *development*-Branch erfolgt wäre.



Da noch keine wirklichen Fortschritte erzielt worden sind, dürfte der *development*-Branch auch noch nicht wieder auf den Server veröffentlicht worden sein (=push).

6.1.2.3 Pull und fetch

Die Grundidee von *holen* und *veröffentlichen* ist relativ einfach und eventuell muss man den Schülern auch nicht mehr erzählen. Da aber Code von anderen Entwicklern auf deinen Rechner kommt, solltest du diesen eventuell nicht ungesehen in deine Entwicklung aufnehmen. `git pull` macht aber genau das.
Willst du an dieser Stelle sicher gehen, so machst du zuerst `git fetch` und siehst dir die Änderungen zuerst an – siehe weiter unten. Im Anschluss kannst du sie dann mit `git merge` übernehmen.

6.2 Hands on

In den nachfolgenden Abschnitten wird oft das Wort *origin* auftreten. Wir sehen uns lieber gleich an, was es damit auf sich hat. Für die einzelnen Schritte gibt es Scripts in den Kursmaterialien!

6.2.1 origin

Origin.sh

```
#!/bin/bash

# Putzen
if [ -d labor1 ]
then
  rm -rf labor1
fi

# Anlegen
mkdir labor1
cd labor1

AKTUELL=$PWD  # Pfad merken

# Repo "entfernt.git" anlegen
git init --bare entfernt.git
# branch auf main umbenennen
cd entfernt.git
git branch -m main
cd ..

# Repo in Ordner "lokal" clonen
git clone $AKTUELL/entfernt.git lokal

# Überblick
cd lokal
git status
```

Das Script `origin.sh` erstellt dir

- Ein Repository *entfernt*, das einen Server darstellt
- Ein Repository *lokal*, das deinen Rechner darstellt, indem es eine Kopie von *entfernt* erstellt (=clone).

Öffne eine *git*-Bash in einem Spielordner und führe das Script `origin.sh` aus. Die Ausgabe wird bei dir minimal abweichen.

```
Leeres Git-Repository in /tmp/labor1/entfernt.git/ initialisiert
Klone nach 'lokal' …
warning: Sie scheinen ein leeres Repository geklont zu haben.
Fertig.
Auf Branch main

Noch keine Commits

nichts zu committen (erstellen/kopieren Sie Dateien und benutzen
Sie "git add" zum Versionieren)
```

Durch

```
git remote -v
```

siehst du, woher das Repository stammt. Bei dir ist natürlich ein anderer Pfad zu sehen!

```
origin  /tmp/labor1/entfernt.git (fetch)
origin  /tmp/labor1/entfernt.git (push)
```

Wenn also in den folgenden Ausgaben und Befehlen *origin* erscheint, dann ist dieser Pfad gemeint. *Pfad* kann allerdings auch eine Netzwerkverbindung zu einem Server sein – das ist sogar der Normalfall.

Mit *origin/main* ist immer der Branch im entfernten Repository gemeint, *main* alleine bezieht sich auf den Branch in deiner lokalen Kopie.

6.2.2 Zeiger

Der nachfolgende Setup erstellt

- einen Ordner
- im Ordner ein vollwertiges Repository (–bare)
- eine lokale Kopie des Repositories mit dem Namen *susi*
- eine lokale Kopie des Repositories mit dem Namen *max*

Öffen in einem geeigneten Ordner ein *git*-Bash und führe das Script `susi_und_max.sh` aus (./susi_und_max.sh}).

Beachte, dass aus in dem Script im oberen Teil etwas Overhead nötig ist, damit die Benutzer nicht mit einem leeren Repository konfrontiert werden – dort fehlt dann nämlich ein Branch!

```
#!/bin/bash
```

```
# Putzen
if [ -d labor2 ]
then
  rm -rf labor2
fi

# Anlegen
mkdir labor2
cd labor2

git init --bare entfernt.git
cd entfernt.git
git branch -m main
cd ..

ARBEIT=$PWD  # aktuellen Pfad merken
SUSI=$ARBEIT/susi  # Abkürzungen für den Überblick
MAX=$ARBEIT/max

# Overhead
git clone $ARBEIT/entfernt vorbereitung
cd vorbereitung
echo "Hi" > README.md
git add README.md
git commit -m "Init"
git push -u origin main
cd ..
rm -rf vorbereitung

git clone $ARBEIT/entfernt susi
git clone $ARBEIT/entfernt max

#
#cd $SUSI
#echo "Hallo" > datei.txt
#git add datei.txt
#git commit -m "Hallo geschrieben"

#git branch -m main
```

Öffne ein zweites Fenster im gleichen Ordner und entscheide dich, welches für *Susi* und welches für *Max* stehen soll. Wechsle jeweils in das Repository (`cd susi` und `cd max`).

Lasse dir in beiden Repositories den Status ausgeben

```
git status
```

Es sollte jeweils erscheinen

```
Auf Branch main
Ihr Branch ist auf demselben Stand wie 'origin/main'.

nichts zu committen, Arbeitsverzeichnis unverändert
```

Im Prinzip hat Susi jetzt eine identische Kopie des Repositories erstellt. Allerdings ist der *main*-Branch jetzt *Susis main*-Branch in *diesem* Repository. Die Statusmeldung sagt dir, beide Branches *main* (Susis) und *origin/main* (der im Original Repo) sind gleich.

Nachfolgend stehen die Bezeichnungen *Original* und *SUSI* bzw. *MAX* für die entsprechenden Repositories.

Susi erstellt in *SUSI* eine Datei und einen Commit:

```
echo "Hallo Welt" >> datei.txt
git add datei.txt
git commit -m "Begrüßung"
```

und betrachtet den Status erneut:

```
Auf Branch main
Ihr Branch ist 1 Commit vor 'origin/main'.
  (benutzen Sie "git push", um lokale Commits zu publizieren)

nichts zu committen, Arbeitsverzeichnis unverändert
```

Der *git*-Ordner führt also Buch über die Beziehung zwischen den Repositories und sagt ihr, dass *SUSI* einen Commit weiter vorne ist, als das *Original*.
Das macht *git*, indem es *Pointer* auf Commits setzt (also quasi eine Datei im *.git*-Ordner führt, wo der aktuelle Hash enthalten ist.)

```
...
   config
   description
   HEAD                                                    ①
   hooks
       applypatch-msg.sample
...
   packed-refs                                             ③
   refs
       heads
           main
       remotes
           origin
               HEAD                                        ②
       tags
```

① Enthält den Hash vom letzten Commit in der lokalen Kopie
② Enthält den Hash vom letzten Commit im Original
③ Siehe nachfolgender Text

Ein `git log --oneline` zeigt dir:

```
78db34e (HEAD -> main) Begrüßung
d46681d (origin/main, origin/HEAD) Init
```

Und ein Blick in die beiden Dateien liefert die Hashes – zumindest fast, denn *git* hat hier schon wieder optimiert und den Hash vom *origin/master* in die Datei *packed-refs* umgelagert.

Das *Original* weiß von diesem Commit noch nichts – die Repositories haben ja auch noch nicht miteinander kommuniziert! Wenn Max in seinem Repository *MAX* nachsieht kann er Susis Datei nicht sehen und daran würde auch ein `git pull` nichts ändern.

Susi führt nun einen Push aus:

```
git push origin main
```

Git kommentiert das:

```
Objekte aufzählen: 4, fertig.
Zähle Objekte: 100% (4/4), fertig.
Delta-Kompression verwendet bis zu 6 Threads.
Komprimiere Objekte: 100% (2/2), fertig.
Schreibe Objekte: 100% (3/3), 292 Bytes | 292.00 KiB/s, fertig.
Gesamt 3 (Delta 0), Wiederverwendet 0 (Delta 0), Pack wiederverwendet 0
To /tmp/labor2/entfernt
   d46681d..78db34e  main -> main
```

und jetzt liefert der Status:

```
Auf Branch main
Ihr Branch ist auf demselben Stand wie 'origin/main'.

nichts zu committen, Arbeitsverzeichnis unverändert
```

Die Zeiger in *.git* wurden also auf den gleichen Hash gesetzt.

Wechseln wir zu Max

Max hat noch sein ursprüngliches Repository und möchte auf Änderungen am Server prüfen.

```
git fetch
```

Als Ausgabe bekommt er

```
remote: Objekte aufzählen: 4, fertig.
remote: Zähle Objekte: 100% (4/4), fertig.
remote: Komprimiere Objekte: 100% (2/2), fertig.
remote: Gesamt 3 (Delta 0), Wiederverwendet 0 (Delta 0), Pack wiederverwendet 0
Entpacke Objekte: 100% (3/3), 272 Bytes | 272.00 KiB/s, fertig.
Von /tmp/labor2/entfernt
   d46681d..78db34e  main       -> origin/main
```

Sein Status sagt ihm

```
Auf Branch main
Ihr Branch ist 1 Commit hinter 'origin/main', und kann vorgespult werden.
  (benutzen Sie "git pull", um Ihren lokalen Branch zu aktualisieren)

nichts zu committen, Arbeitsverzeichnis unverändert
```

Mit *vorspulen* ist ein Merge gemeint. Max will aber wissen *was* sich geändert hat.

```
git log origin main
```

Max sieht den Commit, der noch nicht in seinen Branch integriert ist.

```
commit 78db34e3176c4b78d4b027eac06aa4dda5c50cdb (origin/main, origin/HEAD)
Author: wolfgang <susi@t-online.de>
Date:   Wed Jan 29 12:39:38 2025 +0100

    Begrüßung

commit d46681db23c9e2b3f5379a951c026e3d673fe3e0 (HEAD -> main)
Author: wolfgang <lehrer@t-online.de>
Date:   Wed Jan 29 12:34:48 2025 +0100

    Init
```

Schön sieht man das auch mit

```
git show-ref
```

Der lokale *main* befindet sich noch bei d46681, der entfernte *origin main* hingegen bei 78db34.
Durch folgenden Befehl sieht Max die Differenz der Versionen:

```
git show 78db34
```

Geht es einfach nur um den gesamten Inhalt:

```
git show 78db34e3:datei.txt
```

Max erkennt, dass es problemlose Änderungen sind und übernimmt sie in seinen Branch:

```
git merge
```

Die Hashes ziegen, was passiert:

```
Aktualisiere d46681d..78db34e
Fast-forward
 datei.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 datei.txt
```

Hätte Max größeres Vertrauen, so würde er sich diese Recherche sparen und einfach ein `git pull` ausführen. Im Prinzip ist dieser Befehl die direkte Nacheinanderausführung von `fetch` und `merge`.

6.2.3 Einige Fragen

Wie starte ich from Scratch
Es geht hier nicht um den Schlüsseltausch, sondern nur das Handling der Repositories.

```
# auf dem Server
git init --bare musterrepo
git branch -m main
```

```
# auf dem Client
# Besndere Schritte NUR auf dem ersten Client
git clone gituser@server:port/pfad/zu/repo.git
git switch -c main
echo "Welcome" > Readme.txt
git add Readme.txt
git commit -m "initial"
git push -u origin main
```

Jeder weitere Benutzer clont nun ein funktionsfähiges Repostory.

Muss ich das 'origin main' immer angeben?

Auch wenn der Branch *main* besonders klingt, ist er doch ein ganz normaler Branch. Seine Besonderheit
ist lediglich, dass er durch das Clonen auch gleichzeitig als *upstream-branch* definiert wurde – dehalb funktioniert statt `git pull origin main` auch die kurze Version `git pull` (analog für `git push`).

Wenn du einen neuen Branch anlegst und einen commit pushen willst:

```
git switch -c probe
echo "test" > probedatei.txt
git add probedate.txt
git commit -m "Datei ergänzt"
git push
```

dann gibt es eine Fehlermeldung

```
Schwerwiegend:
Der aktuelle Branch probe hat keinen Upstream-Branch.
Um den aktuellen Branch zu versenden und den Remote-Branch
als Upstream-Branch zu setzen, benutzen Sie

    git push --set-upstream origin probe

Damit das automatisch für Branches ohne Upstream-Tracking passiert,
siehe 'push.autoSetupRemote' in 'git help config'.
```

Willst du den Branch pushen, dann musst du obigen Befehl verwenden oder kürzer

```
git push -u origin probe
```

Theoretisch geht es auch, von einem *nicht upstream* Branch direkt auf einen Upstream-branch zu pushen – das läuft aber am lokalen Branch vorbei, dem dann ein Commit fehlt:

```
git push origin probe:main
```

Ist es egal, in welchem Branch ich fetch/pull/push ausführe?
Jein.

Beim Pushen kann nicht viel schief gehen. Upstream-Branches laufen automatisch und wenn sie nicht als upstream konfiguriert sind, dann geht der Push (standardmäßig) nicht.

Beim Pull sieht es anders aus. Du kannst jeden Remote-Branch in jeden lokalen Branch *pullen* – also *mergen*. Ist der aktuelle Branch als Upstream konfiguriert, holt sich ein kurzes `git pull` die richtigen Daten. Ein `git pull <branch>` holt aber genau das, was du angegeben hast! Das ist eine Fehlerquelle!

Wie hängen lokale und remote Branches zusammen?
Zunächst sind Branches lokal und werden beim `git push` auch nicht auf den Server übertragen.

7 Ein minimaler git-Server

7.1 Zu klärende Fragen

Ein einfacher *git*-Server ist lediglich ein Server mit einer rudimentären Installation von *git* und einer Erreichbarkeit über das Netz. Da gerade diese Erreichbarkeit bei Windows ein Problem darstellt, handelt es sich im Regelfall um einen Linux-Server. Als solcher besitzt er üblicherweise kein grafisches Benutzerinterface (GUI).

Je nach Szenario müssen auf dem Server Benutzer angelegt und verwaltet werden. Das kann ziemlich aufwändig sein und ist bei größeren Benutzerzahlen ohne Scripting kaum machbar. Es gibt Hilfsmittel in Form von Weboberflächen, die hier aber nicht weiter thematisiert werden.

Für einen reinen *git*-Server genügt allerdings bereits ein einziger Bunutzer – z.B. *git*.

Zweck des Servers
Geht es nur um *git* oder sollen noch Wordpress, VPN-Server, … dazu kommen. Für jedes Szenario wird die endgültige Konfiguration anders aussehen …

Wer nutzt den Server
Müssen auch andere Lehrkräfte (administrativen) Zugriff auf den Server besitzen? Werden Gruppen benötigt?

Rolle der Benutzer
Sollen die Benutzer nur für die Verwendung von *git* angelegt und konfiguriert werden, oder sollen sie *vollwertige* Benutzer sein, die den Server auch in anderen Szenarien sinnvoll nutzen dürfen?

Benennung der Benutzer
Sollen die Schüler mit Zugangsdaten versehen werden, die ihren Namen widerspiegeln – Probleme siehe unten – oder sollen Zugänge im Sinne von `user1` bis `user20` verwendet werden?

Bei Verwendung von richtigen Namen entstehen folgende Probleme:

- Sonderzeichen in den richtigen Namen erschweren das automatische Erstellen von Benutzernamen.
- Verschiedene Namen können den gleichen Login ergeben.
- Behandlung von Schülern mit gleichen Namen in gleichen / verschiedenen Klassen

Die Variante mit `user1` bis `user20` ist unpersönlich, leichter einzurichten, erschwert dem Lehrer aber eventuell eine Zuordnung zum echten Schüler.

Hierfür gibt es keine perfekte Lösung.

7.2 Nur git

Wenn ein Schüler nur *git* verwenden soll, dann braucht er kein vollwertiges Benutzerkonto, an dem er sich auch für andere Arbeiten anmelden kann. Für die Verwendung von *git* genügt es bereits, wenn man einen Benutzer mit Namen *git* (als Beispiel) erstellt. Nachfolgend sind dann diese Schritte nötig:

- Dem Benutzer *git* wird eine *echte* Anmeldung unmöglich gemacht.
- Schüler erstellen sich SSH-Schlüssel-Paare
- Sie schicken ihren Public-Key per Mail an den Lehrer
- Der Lehrer fügt den Schlüssel in die Datei `/home/git/.ssh/authorized_keys` ein.

7.2.1 Detaillierte Anleitung

Benutzer anlegen

```
useradd -m git
passwd git  # Das Kennwort wird 2x unsichtbar abgefragt.
```

Anmeldung deaktivieren

```
nano /etc/passwd
```

Den Benutzer *git* suchen und am Ende der Zeile aus `/bin/sh` den Eintrag `/bin/false` machen. Nun mit der Tastenkombination strg + o und ‘strg + x die Änderung abspeichern.

Erstellen eines Schlüsselpaars
Unter Linux / MacOS einfach eingeben:

```
ssh-keygen -t rsa -b 4096 -C "schueler@example.com"
```

Als Dateiname *schule* vergeben und die Fragen nach einer PassPhrase beide Male mit der Enter-Taste übergehen.

Unter Windows geht das ab Windows 10 mit dem gleichen Befehl aus der *cmd.exe* oder *powershell*. Frühere Versionen müssen PuTTYgen verwenden – Anleitung z.B. über KI. Bei der Vorbereitung des Workshops hat sich das Erstellen des Schlüsselpaares hier allerdings als ziemlich problematisch erwiesen. Mit Standardnamen im Standardordner ging es problemlos, bei Angabe eines anderen Namens wurde es kompliziert.

TODO: Überarbeiten Key unter WIndows.

Zuschicken der Datei `schule.pub` per Mail
Sollte klar sein. Durch die Wahl von *schule* wird das Schlüsselpaar auch nicht im versteckten Ordner `.ssh` angelegt, sondern sichtbar direkt unter `c:\user\BENUTZER`.

TODO: Das klappt noch nicht

Kopieren des Schlüssels
Der Schlüssel muss aus der eMail in die Datei `/home/git/.ssh/authorized_keys` eingetragen werden.

TODO: Klappt das über strg + c

Die Schüler sollten nun kennwortfrei mit *git* arbeiten können.
Beim Einsatz einer Weboberfläche (später im Script) müssen die Schüler diesen Schlüssel selbst dort im System registrieren.

Hintergrund
Nach der Anmeldung über ssh wird über die Einstellungen im Benutzerkonto entschieden, welche Art von Terminal der Benutzer erhält. Üblich ist die `bash, sh, csh, ...`. Wird hier ein *ungültiges* Terminal eingetragen, so ist eine Anmeldung auf diese Weise nicht mehr möglich. Für *git* spielt diese allerdings keine Rolle, da hierfür das Terminal nicht gestartet werden muss. Da trotzdem eine Art von Anmeldung für den Zugriff auf *git* erfolgen muss, wird das Public-Key-Verfahren verwendet.

Hier muss entschieden werden, wer für die Schlüssel zuständig ist.

Varianten

- Der Lehrer erstellt Schlüsselpaare für die Schüler, die sie auf den verwendeten Geräten (Schule, Zuhause) hinterlegen und die vom Lehrer auf dem Server freigeschaltet werden. Dies kann zu Problemen führen, falls Schüler bereits aus anderen Gründen mit ssh arbeiten und bereits Schlüssel besitzen[1].
- Der Schüler erstellt selbst das Schlüsselpaar oder bringt sein eigenes mit (ausdrücklich NICHT empfohlen, da der Private-Key dann auf einem Schulrechner liegt!). Er muss dann eigenständig seinen Public-Key auf dem Server hinterlegen. Das ist auch problematische, weil der Schüler zunächst das Kennwort für den Benutzer *git* besitzen muss. Nach dem Hinterlegen des Schlüssels ist es dann die Aufgabe des Lehrers, den Account zu sperren.
- Der Schüler erstellt für die Schule und zuhause unabhängige Schlüsselpaare. In diesem Fall müssen beide Schlüssel auf dem Server als gültig eingetragen werden.

7.3 Vorbereitung des Systems

Nachfolgend wird die Installation eines funktionsfähigen *git*-Servers beschrieben. Bevor man sich an die Installation auf einem Server im Internet macht, ist etwas *Training* in einer virtuellen Umgebung ganz sinnvoll. Nach Installation und Konfiguration eines Übungsservers in VirtualBox sind die Schritte isdentisch. Die Anleitung zur Installation von VirtualBox und dem entsprechenden Gastsystems *Ubuntu-24.04* befindet sich am Ende des Handouts. Gegebenenfalls also zuerst dort fortfahren und dann erst hierher zurückkehren.

7.4 Grundkonfiguration

Alle erforderlichen Konfigurationsschritte erfolgen über ssh im Terminal auf dem Server. In der virtuellen Maschine kann dies zwar auch direkt aus VirtualBox heraus erfolgen, zum *üben* ist es aber auch hier sinnvoll, sich mit ssh auf die VM zu verbinden.

[1] Auch das kann im Prinzip konfiguriert werden, stellt aber höhere Anforderungen an den Benutzer

7.4.1 SSH-Verbindung

Windows
Seit einiger Zeit kann auch Windows auch ohne Zusatzsoftware SSH-Verbindungen aufbauen. Allerdings unterliegt die `cmd.exe` einigen Einschränkungen (z.B. kopieren von Textinhalten), die bei der Verbindung mit Putty besser gelöst sind. Putty kann HIER als einzelne Datei heruntergeladen werden. Starten Sie dafür die Commandozeile `cmd.exe` und geben Sie folgenden Befehl ein (Benutzer und IP-Adresse müssen bekannt sein, ebenso das Kennwort):

TODO: Unterschied cmd und putty

```
ssh benutzer@ip-adresse
```

Es erscheint eine Sicherheitsabfrage, die mit `yes` und `enter` bestätigt werden muss:

TODO: Als Bild einbinden

```
The authenticity of host 'try.example.com (38.243.220.195)' can't be established.
ED25519 key fingerprint is SHA256:DSeSsfXDL2PkSlLYCt64krg9xa2vNr3og5SBJzZ/WNk.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

Hintergrund
Der Rechner merkt sich den Fingerabruck des Servers, zu dem die Verbindung aufgebaut wurde. Sollte später ein anderer Server diese IP-Adresse übernehmen (z.B. Man in the middle), so stimmt der Fingerabdruck nicht und es gibt eine Warnung.

Apple und Linux
Auch hier erfolgt der Verbindungsaufbau in gleicher Weise direkt aus dem Terminal.

Die Eingabe des Kennworts kann je nach Installationsvariante auch ohne sichtbare Zeichen erfolgen!

Eine ssh-Verbindung kann auf verschieden Weise wieder abgebaut werden:

- brutal, geht aber: Fenster schließen
- `exit()`
- `logout`
- strg + d

7.4.2 Software installieren

Bei *Ubuntu-24.04* ist in der Standard-Installation die Versionsverwaltung *git* bereits vorinstalliert und auch der ssh-Server sollte bereits funktionieren. Bei gemieteten Installationen muss das nicht so sein, ssh sollte aber funktionieren. Für die Verbindung mit ssh benötigt man die IP-Adresse. Von einem Mietserver kennt man die Adresse im Regelfall, in VirtualBox muss man sie erst im Terminal ermitteln, da sie vom DHCP-Server zugewiesen wird.

Hinweis
Falls man *git* im Klassenzimmer betreibt (was nicht besonders sinnvoll ist) und eine gewisse *Konstanz* in den Unterrichtsverlauf bringen möchte, dann sollte der Systemadministrator diese virtuellen Maschine in den DHCP-Server eintragen, damit sie immer die gleiche IP-Adresse bekommt!

```
ip addr show
```

Je nach Anbieter des Systems kann die Bildschirmausgabe etwas anders aussehen!

$$T \sim \sqrt{\frac{m}{D}}$$

$$\mathrm{T} \sim \sqrt{\frac{m}{D}}$$

Abbildung 7.1: Interfaces

Der veraltete Befehl `ifconfig` kann jederzeit nachinstalliert werden durch

```
sudo -i           # Admin werden
apt-get update    # Software-DB aktualisieren - dauert!
apt-get install net-tools  # mit y oder J bestätigen
```

Hinweis
Zum Installieren von Software gibt es auf Linux verschiedene Programme: `apt`, `apt-get`, `aptitude`, `snap`, `flatpak`, `appimages`, ... Eine detaillierte Beschreibung ginge hier zu weit. In diesem Workshop wird nur mit `apt`, `apt-get` und `aptitude` gearbeitet. Diese drei Programme verwenden die gleiche Datenbank für verfügbare Software und sind somit weitgehend identisch. `Aptitude` verfügt über eine – für mich – schönere Suchfunktion. Aus diesem Grund installieren wir es hier gleich nach.

```
apt-get install aptitude -y
```

Check auf installiertes git

```
git --version
```

Falls eine Fehlermeldung erscheint, muss *git* noch installiert werden:

```
apt update
apt get install git -y
```

7.4.3 Firewall

Je nach Anbieter ist auf dem Server eine Firewall aktiv oder auch nicht. Den Status erfragt man mit `ufw status`. Hierbei steht `ufw` für *uncomplicated firewall*. Bei Änderungen an der Firewall immer Vorsicht walten lassen! Man kann sich aussperren!

Ablauf

```
IN:   ufw status
OUT:  Status: inactive

IN:   ufw allow ssh
OUT:  Rules updated
      Rules updated (v6)

IN:   ufw enable
OUT:  Command may disrupt existing ssh connections.
      Proceed with operation (y|n)?
IN:   y
OUT:  Firewall is active and enabled on system startup
```

7.4.4 Ordner für Repository

Wo man diesen im Verzeichnisbaum von Linux anlegt, ist im Prinzip unwichtig – es gibt aber bessere und schlechtere Orte. Der Home-Ordner ist keine gute Idee – besser ist z.B. der Ordner `/srv`. Dort erstellen wir auch gleich ein Demo-Repository:

```
cd /srv
mkdir repositories
chown git:git repositories
cd repositories
git init --bare demo.git
```

Mit diesem Repository werden wir aber erst später arbeiten. Zunächst bleiben wir auf dem lokalen Rechner, um die Grundzüge zu lernen.

8 gitea

Die Installation von *gitea* ist relativ einfach und kann über kopieren und Einfügen erledigt werden. In den Kursmaterialien befinden sich die Scripte

- `setup_gitea.sh` - führt die Basisinstallation durch
- `selfreg.sh` - Selbstregistrierung ein/aus
- `tune.sh` - Feineinstellungen direkt nach der Installation

8.1 Setup-Script

In den folgenden Abschnitten sind die relevanten Passagen der Scripte dokumentiert. Die Befehle müssen nicht per Hand eingegeben werden. Das dient lediglich der Dokumentation. Die Scripte `setup_gitea.sh` und `tune.sh` müssen als `root` bzw. über `sudo` ausgeführt werden.

setup_gitea.sh

```
# Port 3000 muss für das Webinterface offen sein
ufw allow 3000/tcp

# Evtl. direkt auf der Seite https://dl.gitea.com/gitea die aktuelle
# Version checken  und hier anpassen.
wget -O gitea https://dl.gitea.com/gitea/1.22.6/gitea-1.22.6-linux-amd64

chmod +x gitea  # ausführbar machen

# Benutzer git ohne Anmeldeerlaubnis
adduser \
   --system \
   --shell /bin/bash \
   --gecos 'Git Version Control' \
   --group \
   --disabled-password \
   --home /home/git \
   git

# nötige Verzeichnisse erstellen
mkdir -p /var/lib/gitea/{custom,data,log}
chown -R git:git /var/lib/gitea/
chmod -R 750 /var/lib/gitea/
mkdir /etc/gitea
chown root:git /etc/gitea
```

```
chmod 770 /etc/gitea

# Programm an richtigen Ort verschieben
mv gitea /usr/local/bin/gitea

# Autostart einrichten

# Im richtigen Script minimiert!!!!!
cat << EOF > /etc/systemd/system/gitea.service

[Unit]
Description=Gitea (Git with a cup of tea)
After=network.target
###
# Don't forget to add the database service dependencies
###
#
#Wants=mysql.service
#After=mysql.service
#
#Wants=mariadb.service
#After=mariadb.service
#
#Wants=postgresql.service
#After=postgresql.service
#
#Wants=memcached.service
#After=memcached.service
#
#Wants=redis.service
#After=redis.service
#
###
# If using socket activation for main http/s
###
#
#After=gitea.main.socket
#Requires=gitea.main.socket
#
###
# (You can also provide gitea an http fallback and/or ssh socket too)
#
# An example of /etc/systemd/system/gitea.main.socket
###
##
## [Unit]
## Description=Gitea Web Socket
## PartOf=gitea.service
##
## [Socket]
## Service=gitea.service
## ListenStream=<some_port>
```

```
## NoDelay=true
##
## [Install]
## WantedBy=sockets.target
##
###

[Service]
# Uncomment the next line if you have repos with lots of files and get a HTTP 500 error beca
# LimitNOFILE=524288:524288
RestartSec=2s
Type=simple
User=git
Group=git
WorkingDirectory=/var/lib/gitea/
# If using Unix socket: tells systemd to create the /run/gitea folder, which will contain th
# (manually creating /run/gitea doesn't work, because it would not persist across reboots)
#RuntimeDirectory=gitea
ExecStart=/usr/local/bin/gitea web --config /etc/gitea/app.ini
Restart=always
Environment=USER=git HOME=/home/git GITEA_WORK_DIR=/var/lib/gitea
# If you install Git to directory prefix other than default PATH (which happens
# for example if you install other versions of Git side-to-side with
# distribution version), uncomment below line and add that prefix to PATH
# Don't forget to place git-lfs binary on the PATH below if you want to enable
# Git LFS support
#Environment=PATH=/path/to/git/bin:/bin:/sbin:/usr/bin:/usr/sbin
# If you want to bind Gitea to a port below 1024, uncomment
# the two values below, or use socket activation to pass Gitea its ports as above
###
#CapabilityBoundingSet=CAP_NET_BIND_SERVICE
#AmbientCapabilities=CAP_NET_BIND_SERVICE
###
# In some cases, when using CapabilityBoundingSet and AmbientCapabilities option, you may wa
# set the following value to false to allow capabilities to be applied on gitea process. The
# value if set to true sandboxes gitea service and prevent any processes from running with p
# in the host user namespace.
###
#PrivateUsers=false
###

[Install]
WantedBy=multi-user.target
EOF

# Autostart aktivieren
sudo systemctl enable gitea

# Gitea einmalig per Hand starten
sudo systemctl start gitea

echo Einrichten über Web  `http://<ip-adresse>:3000`
```

```
echo SQLITE als Datenbank einstellen
echo **WICHTIG:** Administrator einrichten!
```

8.2 Nacharbeiten

Im Nachgang sollten einige Schreibberechtigungen entzogen werden, damit nur noch `root` Änderungen vornehmen darf:

```
chmod 750 /etc/gitea
chmod 640 /etc/gitea/app.ini
```

Dieses Script deaktiviert Anmeldungen über OpenID und setzt neue Sicherheitsschlüssel.

tune.sh

```
# OpenID deaktivieren
INI="/etc/gitea/app.ini"
KEY1="ENABLE_OPENID_SIGNIN = "
KEY2="ENABLE_OPENID_SIGNUP = "
sed -i "s/${KEY1}true/${KEY1}false/g" $INI
sed -i "s/${KEY2}true/${KEY2}false/g" $INI

# Sicherheit

for KEY in INTERNAL_TOKEN SECRET_KEY JWT_SECRET LFS_JWT_SECRET
do
  VAL=$(gitea generate secret ${KEY})
  sed -i "s/^$KEY[[:space:]]*=.*/$KEY = $VAL/" $INI
done

systemctl restart gitea
```

In der Datei `/etc/gitea/app.ini` kann das wieder rückgängig gemacht werden, indem der Abschnitt `[openid]` so abgeändert wird:

```
[openid]
ENABLE_OPENID_SIGNIN = true
ENABLE_OPENID_SIGNUP = true
```

Selbstregistrierung an/aus

Standardmäßig können sich die Benutzer selbst auf Gitea registrieren. Das vereinfacht die Arbeit für den Lehrer enorm, ist aber bei einer öffentlich zugänglichen Instanz problematisch. Es ist also sinnvoll, diese Einstellung nach dem erfassen der Benutzer zu deaktivieren (kann kederzeit wieder aktiviert werden).
Dazu muss in der Datei `/etc/gitea/app.ini` die folgende Zeile im Abschnitt `[service]` angepasst werden:

```
ORIGINAL:  DISABLE_REGISTRATION = false
ANGEPASST: DISABLE_REGISTRATION = true
```

Dies kann über `nano /etc/gitea/app.ini` gemacht werden (Speichern: `strg + o`, Beenden `strg + x`).

Danach muss Gitea neu gestartet werden:

```
systemctl restart gitea
```

Das Setup-Script des Kurses kopiert das Script `selfreg.sh` nach `/usr/local/bin`, so dass die Registrierung einfach aktiviert und deaktiviert werden kann:

```
selfreg.sh on
selfreg.sh off
```

8.3 SSH

Für den Client muss der ssh-Zugriff möglich sein (clone, push, pull). Es kommt hier der normalen ssh-Server auf Port 22 zum Einsatz[1].

Für den kennwortlosen ssh-Zugriff für clone, push, pull wird ein ein ausreichend langer ssh-Schlüssel benötigt. Die üblichen 2048 Bit genügen hier nicht! Dieser *Deploy-Key* wird später dann über die graphische Benutzeroberfläche in *Gitea* eingefügt (s.u.).

Schlüssel erzeugen

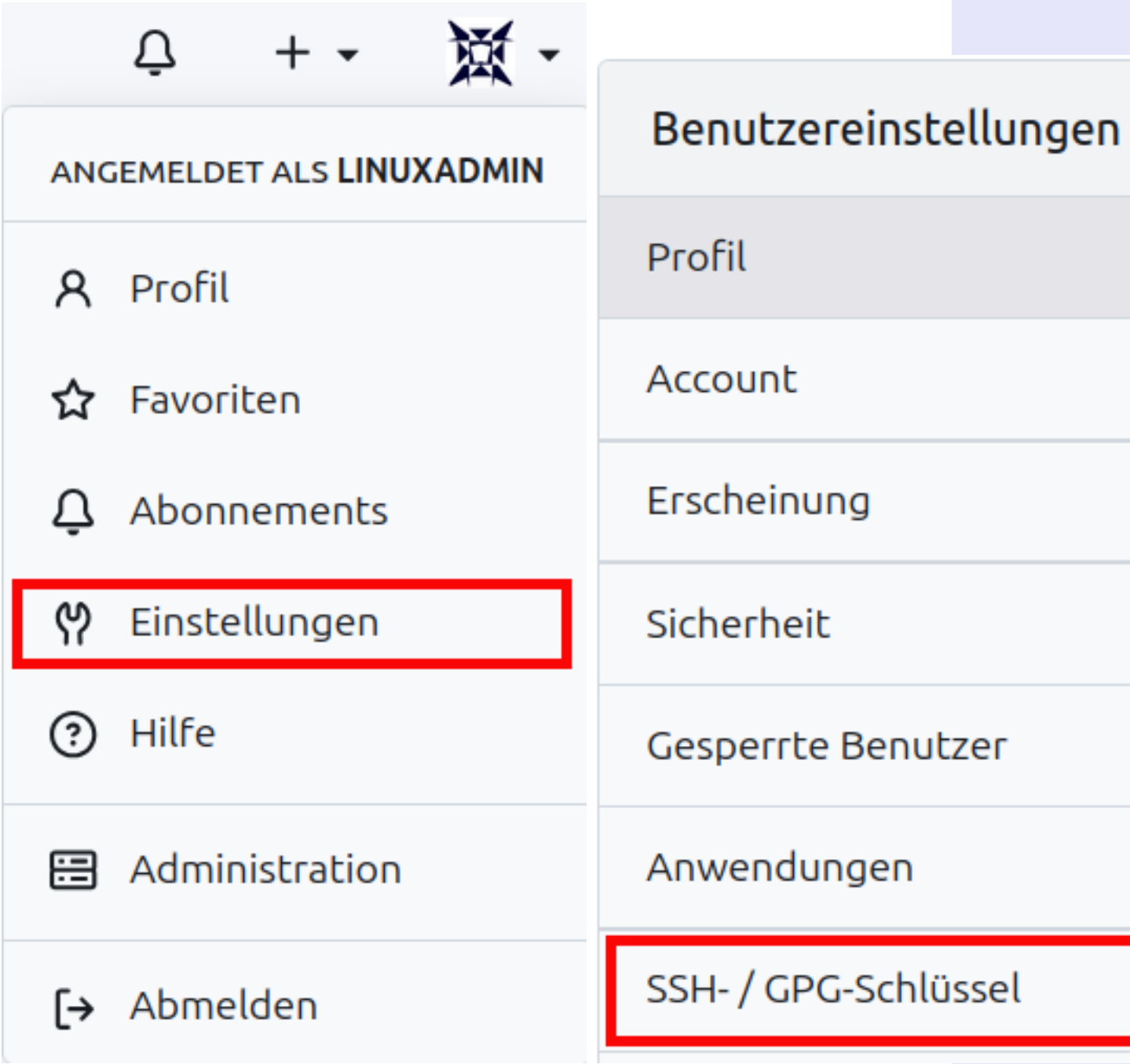
```
IN:  ssh-keygen -t rsa -b 4096
OUT: Generating public/private rsa key pair.
     Enter file in which to save the key
     (/home/linuxadmin/.ssh/id_rsa):
IN:  gitea_rsa
OUT: Enter passphrase (empty for no passphrase): ... leer lassen Enter
OUT: Enter same passphrase again: ... leer lassen Enter
OUT: Your identification has been saved in gitea_rsa
     Your public key has been saved in gitea_rsa.pub
     The key fingerprint is:
     SHA256:ty5F............Q0 linuxadmin@gitserver
     The key's randomart image is:
     +---[RSA 4096]----+
     |      . ..       |
     |.      o  .      |
     | E. . o  o o     |
     |..*. o o+ O .    |
     | =... *+SB.o     |
     | ..  o =O.o.     |
     |..     =.=.      |
     |++ .    *+       |
     |+ =.    .=+      |
     +----[SHA256]-----+
```

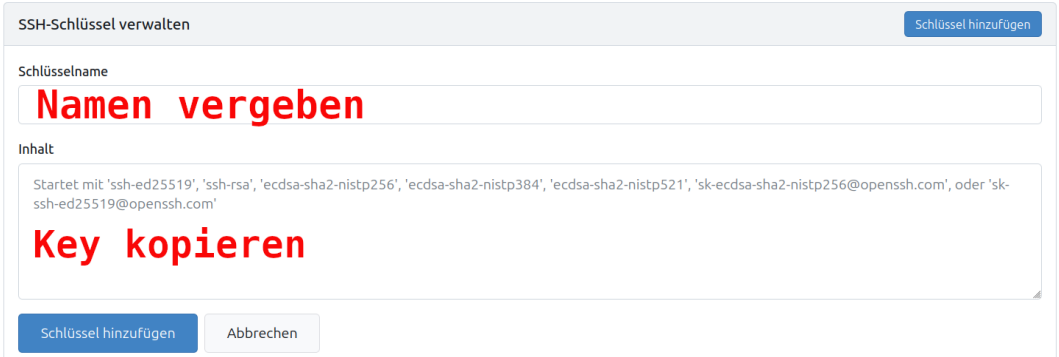
Der Inhalt von `gitea_rsa.pub` kann durch folgenden Befehl im Terminal angezeigt werden:

[1] Gitea würde über einen eigenen ssh-Server verfügen, den man in einigen speziellen Fällen verwenden könnte.

```
cat ~/.ssh/gitea_rsa.pub
```

Die mehrzeilige Ausgabe in einem Stück in die Oberfläche von Gitea kopieren:





Damit ist die Grundkonfiguration beendet. Es sollte nun noch eingestellt werden, welche Anmelde- und Registrierungsvarianten erlaubt sind, um nicht plötzlich fremde Personen in

seiner Gitea-Instanz zu haben.

8.4 HTTPS

Aktuell ist die Instanz nur über `http` und Ip-Adresse zugänglich. Es ist möglich, Gitea auch für `HTTPS` zu konfigurieren. Hierfür gibt es verschiedenen Lösungen, von denen aber eigentlich nur der Einsatz eines Reverse-Proxy mit einem entsprechenden Domain-Name sinnvoll ist. Eine derartige Konfiguration würde den Rahmen dieses Workshops aber sprengen! Zur Erinnerung: Es ging um eine möglichst einfache Installation eines funktionsfähigen Systems.

8.5 API

Oft ist es effizienter, wenn Dinge über Kommandozeile erledigt werden. Wenn z.B. 50 Benutzerkonten erstellt und mit Passwort abgesichert werden sollen, weil die Selbstregistrierung nicht gewünscht ist. In diese Fall verwendet man die API von Gitea. Dies kann von jedem Rechner aus erfolgen, auf Gitea zugreifen kann. Dafür braucht es aber noch zwei Anpassungen an der `/etc/gitea/app.ini`:

1. Ergänzen Sie am Ende der Datei:

```
[api]
ENABLE_ADMIN = true
```

2. Erweitern Sie den Abschnitt `security` falls Sie die Accounts mit sehr einfachen Kennwörtern vorbelegen wollen:

```
[security]
... bisheriger Inhalt ...
PASSWORD_COMPLEXITY = off
MIN_PASSWORD_LENGTH = 6
```

Melden Sie sich als Administrator in Gitea an und rufen Sie das Menu *Einstellungen* des Benutzers auf. Unter *Anwendungen* können *Admin-Token* erzeugt werden, die einen kennwortlosen Zugriff auf das System erlauben. Stellen Sie im aufklappbaren Detail-Abschnitt ein, welche Berechtigungen erforderlich sind:

- Admin - Lesen und Schreiben
- Organisation
- Repository
- User

Kopieren Sie die lange Zahl (=Token) dann weg – sie wird nur einmal angezeigt!

Auf der Konsole (Linux / MacOS) kann dann z.B. folgender Befehl abgeschickt werden, der die Benutzerin Susi anlegt:

```
curl -X POST "http://<ip-adresse>:3000/api/v1/admin/users" \
-H "Authorization: token <kopiertes Token>" \
-H "Content-Type: application/json" \
-d '{
  "username": "susi",
```

```
  "email": "susi@sandmann.com",
  "password": "lalelu",
  "send_notify": false,
  "must_change_password": true
}'
```

Verfeinert wird dieser Befehl in ein Script verpackt – z.B. `gitea_user_anlegen.sh`, das man mit einer Namensdatei als Parameter aufrufen kann, in der folgende Spalten auftreten:

```
# login,kennwort,email  <- Diese Zeile entfernen
susi,lalelu,susi@sandmann.de
max,foobla,max@mustermann.de
...
```

8.5.1 Sinnvolle Szenarien

8.5.1.1 Organisationen

Hier bin ich mir nicht sicher, ob das kein Overkill ist. Eine Arbeit über Teams erscheint mir bei der Größe unserer Szenarien deutlich geeigneter. Das Problem ist nämlich, dass man über die API einen Benutzer nur löschen kann, wenn er in keiner Organistation mehr ist!

Es könnte sinnvoll sein, wenn sich die einzelnen Kollegen als *Organisationen* sehen. Dann müssen Schüler aber mehrfach ins System eingetragen werden und das bereits bei den eMail-Adressen Probleme.

Sinnvoller erscheint eine Namenskonvention beim Erstellen von Teams!

Organisation erstellen

```
token=1234
name=G10a
curl -X POST "http://192.168.3.195:3000/api/v1/admin/organizations" \
-H "Authorization: token ${token}" \
-H "Content-Type: application/json" \
-d '{
  "username": "${name}",
  "full_name": "${name}",
  "description": "Organisation für die Schüler der Klasse ${name}",
  "visibility": "private"
}'
```

Benutzer hinzufügen

```
token=1234
name=G10a
user=susi
curl -X PUT "http://192.168.3.195:3000/api/v1/orgs/${name}$/members/${user}" \
-H "Authorization: token ${token}"
```

User löschen

```
token=1234
user="susi"
# In welchen Organistaionen ist der User
curl -X GET "http://192.168.3.195:3000/api/v1/users/${user}/orgs" \
    -H "Authorization: token ${token}"

# Einzeln austragen, sonst ist er nicht löschbar
orga=organization1
curl -X DELETE "http://192.168.3.195:3000/api/v1/orgs/${orga}/members/${user}" \
-H "Authorization: token ${token}"

# Benutzer löschen
curl -X DELETE "http://192.168.3.195:3000/api/v1/admin/users/${user}" \
-H "Authorization: token ${token}"
```

8.5.2 Teams

Auch ein Team erfordert eine Organisation – hier würde ich einfach die Schule anlegen.

Team erstellen

```
schule="GGP"
token=1234
team=G10a
curl -X POST "http://192.168.3.195:3000/api/v1/orgs/${schule}/teams" \
-H "Authorization: token ${token}" \
-H "Content-Type: application/json" \
-d '{
  "name": "${team}",
  "description": "Team für das erste Projekt",
  "permission": "write"
}'
```

Benutzer hinzufügen

```
schule=GGP
token=1234
team=G10a
user=susi
curl -X PUT "http://192.168.3.195:3000/api/v1/orgs/${schule}/teams/${team}/members/${user}"
-H "Authorization: token YOUR_ADMIN_TOKEN"
```

TODO: GPT shgt Team_id statt Team_name!

9 Docker

Docker ist eine besonders effiziente Art der Virtualisierung. Es ist möglich, auf einem physikalischen Server viele virtuelle Server zu erstellen, diese miteinander kommunizieren und daten austauschen zu lassen und sie auch über das Internet zugänglich zu machen. Es ist also kein Problem, mehrere eigenständige Datenbankserver parallel zu betreiben und damit Übungssystem für Schüler bereit zu stellen. Wie groß der Server dimensioniert sein muss, hängt dann wieder von den speziellen Anforderungen ab (mit 6 Cores und 16 GB RAM geht schon einiges, aber sicher nicht alles!)

9.1 Idee

Nachdem Docker auf dem Server installiert ist (s.u.), kann man sich einfach ein Betriebssystem-Image aus dem Netz herunterladen und als Container starten. Natürlich ist es eine Vertrauensfrage, welche Images man verwendet. Es gibt allerdings *offizielle* Images von Firmen und Organisationen, denen man vertrauen kann:

```
docker search gitlab

NAME                              DESCRIPTION     STARS     OFFICIAL
alpinelinux/gitlab                Alpine...       11
okteto/gitlab                                     3
gitlab/gitlab-runner              GitLab ...      960
vulhub/gitlab                                     1
gitlab/gitlab-ce                  GitLab...       4310
gitlab/gitlab-runner-helper                       50
gitlab/gitlab-ee                  GitLab ...      549
```

In dieser gekürzten Ausgabe ist also anscheinend kein *offizielles* Image dabei. `gitlab/gitlab-ce` ist aber trotzdem offiziell. Dafür recherchiert man kurz auf dockerhub und dort ist gitlab verlinkt, wo die Informationen stehen.

Wir werden übrigens nicht *gitlab* sondern *gitea* verwenden!

Von einem Image kann man dann beliebig viele Instanzen erzeugen, die auch fast ohne Zeitverlust startbar sind. Der längste Teil ist der Download des Image.

Hintergrund
Images sind in Schichten konstruiert, die nacheinander heruntergeladen werden. Diese Schichtenstruktur ist wichtig, wenn man eigene Images erstellen möchte. Der Erstellungsprozess macht nämlich immer auf der Schicht weiter, ab der Änderungen am Image vorgenommen wurden (weitere Software, Dateiänderungen, …). Auf diese Weise wird viel weniger Zeit benötigt, als bei einer komplett neuen Erstellung!

9.2 Installation

Die Befehle befinden sich auch bei den Kursmaterialien!

TODO - Docker install Script

```
sudo -i
apt update
apt install apt-transport-https curl -y

curl -fsSL https://download.docker.com/linux/ubuntu/gpg  \
     | gpg --dearmor -o /etc/apt/keyrings/docker.gpg

echo "deb [arch=$(dpkg --print-architecture) \
      signed-by=/etc/apt/keyrings/docker.gpg] \
      https://download.docker.com/linux/ubuntu \
      $(. /etc/os-release && echo "$VERSION_CODENAME") \
      stable" | tee /etc/apt/sources.list.d/docker.list > /dev/null

apt update

apt install docker-ce docker-ce-cli \
    containerd.io docker-buildx-plugin \
    docker-compose-plugin -y


compose_url="https://github.com/docker/compose/releases/download"
curl -L  $compose_url/1.28.5/docker-compose-`uname \
     -s`-`uname -m` -o /usr/local/bin/docker-compose

chmod +x /usr/local/bin/docker-compose
```

Aktuell läuft Docker jetzt als Benutzer *root* (=Administrator), was man im professionellen Firmenumfeld lieber vermeidet. Es gibt Anleitung, wie man das ändert – ich selbst verwende es aber auch so.

Teste die Installation durch folgende Befehle:

```
docker --version
docker-compose --version
```

9.3 Kurzeinführung

Es gibt sehr viele verschiedene Szenarien, wie Docker verwendet werden kann und entsprechend sind auch die Befehle zahlreich und unterschiedlich kompliziert.

Nachfolgend ein kleines *Work-along*, wo einige Befehle verwendet und erklärt werden. Tipp einfach mal folgenden Befehl:

```
docker run -d -p 8090:80 --name it-tools corentinth/it-tools
```

Die Ausgabe (außer die Hashes) sollte dann so aussehen:

```
Unable to find image 'corentinth/it-tools:latest' locally
latest: Pulling from corentinth/it-tools
43c4264eed91: Pull complete
45a30f47e80f: Pull complete
4c64d3291c88: Pull complete
9dc0279166b1: Pull complete
d3b17590914c: Pull complete
50d6cfdb81c6: Pull complete
6592d833752c: Pull complete
f4cab7bcfad1: Pull complete
65e7766bfa53: Pull complete
c5f5268086b8: Pull complete
Digest: sha256:8b8128748339583ca951af03dfe02a9a4d7363f61a216226fc28030731a5a61f
Status: Downloaded newer image for corentinth/it-tools:latest
bd072e349cab6408a3b5f2d7040d586a9149107d68031089f846d0526c028f45
```

Sieht nicht spktakulär aus, bis man `http://SERVERADRESSE:8090` aufruft. Du hast hier einen kompletten Server voll mit digitalem Spielzeug erstellt - in no time!

Natürlich kannst du den Container einfach weiter laufen lassen – wir wollen uns aber einige Dinge ansehen:

Befehl	Wirkung
docker run	versucht einen Container zu erstellen
-d	im Hintergrund
-p 8090:80	Von außen 8090, im Container eigentlich Port 80
–name	sehen wir gleich
corentinth/it-tools	Name des Images

Es gibt nun zwei Stellen (oberflächlich) mit Informationen zu diesem Server.

```
docker image ls
```

Zeigt uns das heruntergeladene Image:

```
REPOSITORY             TAG       IMAGE ID       CREATED         SIZE
corentinth/it-tools    latest    bb7ba9626731   2 months ago    56.2MB
```

Mit nachfolgendem Befehl überprüft man den Status des Containers:

```
docker ps
```

Ausgabe (gekürzt)

```
CONTAINER ID   IMAGE                 COMMAND                  gekürzt ... NAMES
bd072e349cab   corentinth/it-tools   "/docker-entrypoint.…"   gekürzt ... it-tools
```

Weitere Befehle
|Befehl|Wirkung| |——|——-| |docker stop it-tools| Container stoppen| |docker stop bd072e349cab| geht auch| |docker start it-tools|Container starten| |docker start bd072e349cab| geht auch| |docker rm it-tools|Container entfernen – Image bleibt!| |docker rmi “corentinth/it-tools”|Entfernt auch das Image|

Damit ein Container gelöscht werden kann, muss er beendet sein, damit ein Image gelöscht werden kann, muss der Container gelöscht sein.

9.4 Detaillierter

Für komplexere Szenarien wird der *docker run* Befehl aber etwas unübersichtlich. Man kann nun in zwei Stufen die Komplexität erhöhen:

- Anlegen einer Datei *Dockerfile*, in der die Parameter für *docker run* stehen
- Verwenden von *docker-compose* mit einer zusätzlichen Datei *docker-compose.yml* um z.B. mehrere Server im Verbund zu managen. Dieses Verfahren werden wir bei der abschließenden, vollständigen Installation von Gitea verwenden.

Da sowohl Syntax als auch Varianten von beiden Stufen für eine schnelle Behandlung zu komplex sind, verweise ich auf entsprechende Tutorial auf youtube oder eines der vielen Bücher zum Thema – z.B. von Michale Kofler.

10 Gitea mit Docker

10.1 Basics

Wie im letzten Kapitel besprochen, wird der Einsatz von Docker durch vorgefertigte Dockerfiles und Composefiles sehr vereinfacht. Erstellt man selbst derartige Dateien, so sollte man Chatgpt nur sehr bedingt vertrauen. In seinen Kreationen tauchen oft Konfigurationsparameter auf, die es entweder gar nicht gibt, oder deren Schreibweise falsch ist. Letztendlich führt kein Weg an der Herstellerdokumentation vorbei. Die Dockerimages im Netz funktionieren aber gut, so dass man keine eigenen bauen muss.

Bei einer vollwertigen Gitea-Installation brauchen wir (mindestens) drei Container:

- Gitea
- Datenbankserver
- Runner (Erklärung folgt)

Mit dem nachfolgenden `docker-compose.yaml` können diese erstellt und konfiguriert werden.

Was ist ein Runner?
In fortgeschrittenen Szenarien kann Gitea so konfiguriert werden, dass ein Push auf einen bestimmten Branch zu nachgelagerten Aktionen auf dem Server führt. Das können Unittests sein, aber auch Compilierungen, Erstellung von Internetseiten, … was mit dem versionierten Code schlussendlich geplant ist. Wir werden unser Buch-Projekt so anpassen, dass aus unseren Dokumenten automatisch ein druckreifes PDF erstellt wird. Hier sind wir dann also bei der ursprünglichen Bedeutung der *Online-Kollaboration* angelangt.

Damit diese nachgelagerten Aktionen gestartet werden können, läuft eine *Runner*-Programm in einem eigenen Container. Dieses Programm kommuniziert mit dem Gitea-Container und führt bei Bedarf die entprechende Aktion aus. In eigentlich fast allen Fällen wird zum Ausführen ein weiterer Docker-Container erstellt, der für die eigentliche Aktion zuständig ist. Die Details dazu folgen weiter unten.

10.2 Die Dateien

Natürlich kann man für jeden benötigten Container eine eigene `docker-compose.yaml` erstellen, es ist aber sinnvoll, alle drei Container gemeinsam zu konfigurieren, da dann auch Abhängigkeiten zwischen den Containern autmatisch berücksichtigt werden. Wir besprechen die verwendete `docker-compose.yaml` zuerst Zeile für Zeile. Die vollständige Version steht am Ende des Kapitels und ist Teil der Kursmaterialien.

10.2.1 Allgemeine Einstellungen

```
version: "3"

networks:                                                    ①
  gitea:
    external: true                                           ②

volumes:                                                     ③
  gitea_data:
    driver: local
  gitea_db_data:
    driver: local
  gitea_runner_data:
    driver: local
```

① Alle drei Container müssen miteinander kommunizieren und werden deshalb in ein gemeinsames Netz mit dem Namen *gitea* eingebunden.
② Der Basis-Setup funktioniert hier auch mit `false`. Sobald der Runner aber einen neuen Container erstellt, klappt das nur noch mit `true`.
③ Die Daten der Benutzer und die Einstellungen sollen auch über Neustarts und Upgrades hinweg erhalten bleiben. Dafür dürfen sie nicht Teil des Containers sein, der dabei komplett neu erstellt wird. Es werden auf diese Weise drei Ordner mit den angegebenen Namen erstellt, die außerhalb des Containers liegen und in diesen hinein verlinkt werden.

ToDo: Was hat es genau mit dem exteral:false auf sich? ToDo: Welche andere Driver haben Volumens noch?

10.2.2 Services

```
services:
  server:
    ...
  db:
    ...
  runner:
    ...
```

Dieser schematische Aufbau beschreibt die drei anzulegenden Container. Wir besprechen die einzelnen Abschnitte nachfolgend getrennt.

10.2.3 Server

Teil 1

```
  server:
    image: docker.io/gitea/gitea:1.22.6                      ①
    container_name: gitea
    hostname: gitea
    environment:
```

```
      - USER_UID=1000
      - USER_GID=1000
      - GITEA__database__DB_TYPE=mysql                    ②
      - GITEA__database__HOST=db:3306                     ③
      - GITEA__database__NAME=gitea                       ④
      - GITEA__database__USER=gitea
      - GITEA__database__PASSWD=gitea
```

① Hier könnte auch `gitea:latest` stehen, was Upgrades vereinfacht.
② Möglich sind auch *postgresql* und *sqlite*. Sqlite ist eine Alternative für kleine Installationen, da dabei kein eigener Datenbankserver benötigt wird.
③ Name des Datenbankservers:Port
④ Hochsicherheitskennwörter für die Fortbildung

Teil 2

```
    restart: always                                       ①
    networks:                                             ②
      - gitea
    volumes:                                              ③
      - gitea_data:/data
      - /etc/timezone:/etc/timezone:ro
      - /etc/localtime:/etc/localtime:ro
    ports:
      - "4000:3000"                                       ④
      - "222:22"                                          ⑤
    depends_on:                                           ⑥
      - db
```

① Container immer automatisch neu starten, wenn etwas schief geht oder der Server neu gestartet wird.
② Festlegen des Netzes in dem der Container läuft
③ Die Daten von Gitea liegen im Ordner `gitea_data`, genauer unter `/var/lib/docker/volumes/gitea_data`. Über Volumes wird aber auch der Zugriff auf Ressourcen des Servers gesteuert, hier der Zugriff die Eistellungen der Zeitzone. (`:ro` nur lesender Zugriff)
④ IM Container läuft Gitea auf Port 3000, der Zugriff von außen (Browser) erfolgt über Port 4000. Ist hier eher als Demo gedacht, 3000:3000 ginge auch.
⑤ Der SSH-Server (für pull, push, …) läuft im Container auf Port 22, Zugriff von außen über Port 222. Das ist nötig, da der Server (also der gemietete Server) seinen ssh-Server bereits auf Port 22 laufen hat. Das kann man zwar ändern, muss aber etwas aufpassen, dass man sich nit versehentlich aussperrt, weil man die Firewall nicht angepasst hat.
⑥ Der Container wird erst gestartet, wenn der Datenbankserver erfolgreich hochgefahren ist.

10.2.4 db

Alle relevanten Zeilen sind analog zum obigen Beispiel zu lesen.

```
  db:
    image: docker.io/library/mysql:8
    restart: always
    container_name: db
```

```
    hostname: db
    environment:
      - MYSQL_ROOT_PASSWORD=gitea
      - MYSQL_USER=gitea
      - MYSQL_PASSWORD=gitea
      - MYSQL_DATABASE=gitea
    networks:
      - gitea
    volumes:
      - gitea_db_data:/var/lib/mysql
```

10.2.5 runner

Es werden hier nur neue oder wichtige Einstellungen erklärt!

```
  runner:
    image: docker.io/gitea/act_runner:latest
    container_name: gitea_runner
    hostname: gitea_runner
    environment:
      CONFIG_FILE: /config.yaml                          (1)
      GITEA_INSTANCE_URL: "http://gitea:3000"
      GITEA_RUNNER_NAME: "Probe-runner"
      GITEA_RUNNER_REGISTRATION_TOKEN: <ausfüllen>       (2)
    restart: always
    depends_on:                                          (3)
      - server
      - db
    networks:
      - gitea
    volumes:
      - ./config.yaml:/config.yaml                       (4)
      - gitea_runner_data:/data
      - /etc/timezone:/etc/timezone:ro
      - /etc/localtime:/etc/localtime:ro
      - /var/run/docker.sock:/var/run/docker.sock        (5)
```

(1) Dazu kommt unten mehr!
(2) Füllen wir später noch aus
(3) Dieser Container startet erst wenn die beiden anderen laufen
(4) Ein Volume kann auch nur eine einzige Datei sein. Beachte den Unterschied bei der Schreibung von `./config.yaml` und `gitea_runner_data`. Das `./` am Anfang bedeutet, dass diese Datei im gleichen Ordner liegt wie die `docker-compose.yaml`.
(5) Dieser Container braucht auch selbst Zugriff auf Docker, damit er selbst neue Container starten kann.

Das Config-File
Bei vielen Containern kann die Konfiguration durch Einträge in der `docker-compose.yaml` erfolgen - wie z.B. die Kennwörter. Gerade für Kennwörter gibt es noch sicherere Varianten, das führt aber zu weit. Für den Runner werden aber noch Einstellungen nötig, die man so (anscheinend) nicht übergeben kann. Aus diesem Grund wird oben der Parameter

CONFIG_FILE übergeben. Diese Datei befindet sich ebenfalls in den Kursmaterialien und wird hier nicht abgedruckt. Für eine frische Datei kann man folgenden Befehl aufrufen:

```
docker exec -it gitea_runner act_runner generate-config
```

oder direkt als Datei

```
docker exec -it gitea_runner act_runner generate-config > config.yaml
```

Hinweis
Vielleicht ist dir aufgafallen, dass einmal im Environment Spiegelstriche vorkommen und einmal nicht. Das ist Absicht, denn die Server wollen das unterschiedlich haben. Intern wird bei der Spiegelstrich-Variante eine Liste der Art `["key=value", "key=vlaue", ...]` erstellt, die der Server verarbeiten muss. Der Runner hätte aber lieber ein Dictionary, d.h. eine Darstellung{key:value, key:value, ...}‘ und das passiert **ohne** die Spiegelstriche.

Das Registrierungs-Token
Bist du in Gitea eingelogged, so kannst du über das Einstellungsmenü (Das geht auf Instanz-, Repository- oder Userebene) nach *actions* suchen, dort auf *runner* und dann oben rechts auf *neuen runner anlegen* (o.ä.). Dort erscheint dann das Token, das in die `docker-compose.yaml` eingetragen werden muss. Beim Neustart registriert sich der Runner dann damit bei Gitea.

10.2.6 Starten

Damit der Runner gestartet werden kann, muss zunächst Gitea laufen – sonst kann man kein Token erstellen.

```
docker-compose up gitea db
```

Später wird der Befehl mit einem zusätzlichen `-d` gestartet, damit der Start im Hintergrund erfolgt. Ohne `-d` sieht man eventuelle Fehlermeldungen.

Nun kannst du dich bei Gitea im Browser anmelden und den ersten Benutzer einrichten, den du dann auch als Administrator verwenden kannst.

TODO: Anmeldescreen

Im Prinzip kann man auch Mailregistrierung und andere Dinge konfigurieren – für einen einfachen Betrieb genügt aber das Einrichten des Benutzers.

Nach dem oben beschriebenen Erstellen eines Tokens kannst du den Gitea und den DB-Server mit `ctrl-c` zunächst wieder beenden. Das Token wird nun in die `docker-compose.yaml` kopiert.

Nun kann auch der Runner gestartet werden. Auch jetzt zu Testzwecken zuerst ohne `-d`:

```
docker-compose up
```

Hat das ohne Fehler geklappt, dann wieder `strg + c` und dann ein richtiger Start:

```
docker-compose up -d
```

Gitea sollte jetzt verwendbar sein!

10.2.7 Einrichten

Im Prinzip ist Gitea bereits voll funktionsfähig. Wenn die Schüler **nur** über den Browser mit dem System arbeiten, sind auch keine weiteren Einstellungen nötig. Wird aber vollwertiger Zugriff (clone/pull/push) benötigt, so müssen die Benutzer noch ihren ssh-Schlüssel im System hinterlegen.

10.2.8 Schlüssel erstellen

Auf Linux und MacOS

```
ssh-keygen -t rsa -b 4096
```

Im weiteren Verlauf wird nach einer Keyphrase gefragt – leer lassen – und ein Name für den Schlüssel abgefragt – gitea passt gut.

Diesen Schlüssel lässt man sich ausgeben:

```
cat ~/.ssh/gitea.pub
```

In den Benutzereinstellungen auf Gitea kann nun ein neuer Schlüssel hinzugefügt werden.

TODO: Eventuell Screenshot

Auf Windows
… ach ja … Windows …

10.2.9 Clonen

Das Clonen eines öffentlichen Repositories über Https sollte auch ohne Schlüssel funktionieren. Für einen vollwertigen Zugriff verwendet man aber lieber *ssh* und zwar mit folgendem Befehl:

```
git clone ssh://git@mein_server:222/BENUTZER/REPO.git
```

Die Angabe von 222 ist nötig, da wir nicht Port 22 verwenden – man muss also die Adresse, die man in Gitea über dem Repository sieht, noch entsprechend anpassen.

TODO: Screenshot

11 Actions

Dieses Thema ist nun so groß und speziell, dass wir an unsere Genzen stoßen. Ich möchte nur kurz das Vorgehen für unser Szenario der Bucherstellung besprechen.

12 Clients

Generell gilt natürlich, dass im Informatikunterricht ein Tablet (oder Handy) kein geeignetes Werkzeug ist. Wir sprechen im Folgenden also von Laptops oder Desktops und diese können mit Windows, MacOS, Linux oder ChromeOS ausgestattet sein. Für jedes dieser Systeme gibt es viele verschiedene Git-Clients. Die Kommandozeile (Terminal, Bash, Git-Bash, …) ohne grafische Benutzeroberfläche ist gerade bei Anfängern nicht besonders beliebt, aber gerade im Umfeld von Servern sehr verbreitet. Entwickler, die mir den IDEs von JetBrains, VS-Code, o.ä. arbeiten, verwenden oft die dort eingebauten Git-Clients. Leider sind diese Entwicklungsumgebungen dür den Einsatz im Unterricht ein Overkill. Bei einer kleineren IDE kann ein graphischer Git-Client diese Lücke schließen.
Probleme gibt es allerdings traditionell bei der heimischen Installation der entsprechenden Software (GitKraken, …). Mit Hilfe von *portable* Versionen kann das aber deutlich vereinfacht werden. Diese Software-Variante wird nicht installiert, sondern lediglich entpackt. Der entstehende Ordner kann sogar auf einen USB-Stick ausgelagert werden. Damit sind auch die Rechner der Eltern problemlos benutzbar.

Portable Git

13 Autograding

Der Workflow für das Autograding von Schülerlösungen unterscheidet sich deutlich von der normalen Arbeit mit git/gitea, weil man den *kreativen Wissenstransfer* doch möglichst vermeiden möchte.
Beim Autograding geht es im Prinzip darum, dass die Schüler ihre Aufgaben / Prüfungen auf eine Art und Weise schreiben, dass sie mit automatisierten Test überprüft werden können. Hierbei können sichtbare Tests (der Schüler sieht sofort, ob seine Lösung passt) und versteckte Tests verwendet werden.
Technisch gesehen führt der Schüler einen Commit aus, der die Überprüfung triggert. Es ist auch möglich, dass sich der Lehrer die Schülerlösungen holt und dann erst testet.

13.1 Vorarbeiten

- Die Schüler müssen im System sein.
- Erstellen eines Teams
- Schüler dem Team zuordnen
- Ein Repository für Hausaufgaben oder spezielle Prüfungen erstellen und die benötigten Materialien hinein kopieren.

Das Script aus den Kursmaterialien führt dann für alle Teammitglieder nachfolgende Schritte aus * Das Repository für den Schüler *forken*, d.h. eine vollwertige Kopie erstellen * Die Berechtigungen so setzen, dass kein Schüler den Fork eines Mitschülers sehen kann

13.2 Workflow Schüler

Das hängt nun von der genauen Aubeitsweise ab. Im Prinzip muss der Schüler die Aufgabenstellung in der Aufgabendatei erledigen, die im Original-Repository erstellt wurde. Nach dem Erstellen der Lösung wird die Arbeit committet und auf den Server gepusht. Dort greifen dann die nachfolgende Automatismen.

Da das gleiche Repo (Hausaufgaben) über einen längeren Zeitraum genutzt werden soll, muss der Schüler regelmäßig die Änderungen im Origina-Repo in seinen Fork übertragen. Der Lehrer har dafür zu sorgen (Anleitung, …) dass es Dateien gibt (Namenskonvention), die der Schüler nicht anfasst. Auf diese Weise werden Merge-Konflikte vermieden, wenn Original-Repo und Fork zusammengeführt werden.

```
git pull


## Ablauf
```

```
* Schüler einrichten
* CI/CD System einrichten - hier wird auf jenkins verweisen? Geht auch otter-grader?
* Vorlagen für verschiedene Aufgaben erstellen ?????
* Anleitung durch README.md


## Beginn des Kurses

* Der Lehrer erstellt ein Repo
* Die Schüler forken das über die GUI
* Die Schüler Clonen ihr eigenes Repo
```

git clone https://gitea.example.com/student1/haupt-repository.git

* Jeden Tag wird das Repo gepullt

cd haupt-repository git pull origin main

* Die Hausaufgaben werden committed und gepusht
```bash
git add . --all
git commit -m "erledigt"
```

- Der Lehrer sammelt die Arbeien ein

```
#!/bin/bash

STUDENT_REPOS=("https://gitea.example.com/student1/hausaufgaben.git" "https://gitea.example.
LOCAL_DIR="student_submissions"

mkdir -p $LOCAL_DIR
cd $LOCAL_DIR

for REPO in "${STUDENT_REPOS[@]}"; do
  REPO_NAME=$(basename $REPO .git)
  if [ -d "$REPO_NAME" ]; then
    cd $REPO_NAME
    git pull origin main
    cd ..
  else
    git clone $REPO
  fi
done
```

14 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

15 Summary

In summary, this book has no content whatsoever.

References

Knuth, Donald E. 1984. „Literate Programming“. *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.