GIT

28.10.2025

Warum eigentlich GIT?

WER hat WAS, WANN und WARUM geändert?

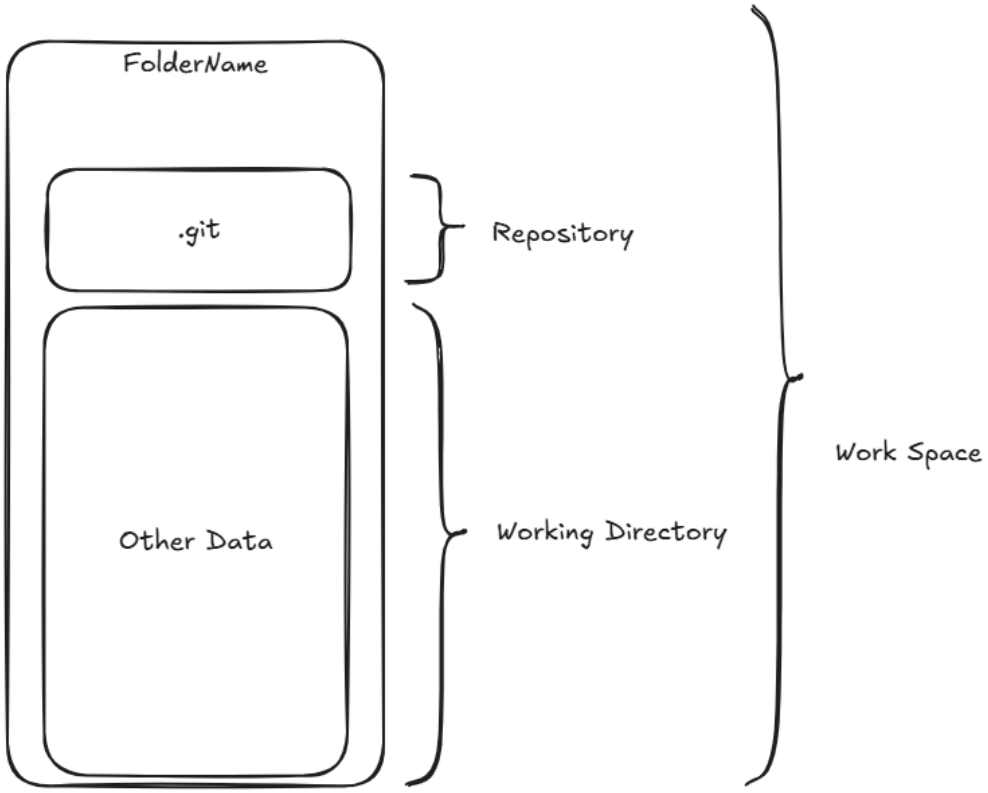
Versionsverwaltung ist relevant weil Software stetig von unterschiedlichen Entwicklern weiterentwickelt und neue Features getestet werden müssen (und auch Fehler passieren).

Was wird gespeichert?:

- WER? Autor eines Commits
- WAS? Unterschiede bzw Dateien im Commit
- WANN? Timestamp des Commits
- WARUM? Commitbeschreibung

Ein **VersionControlSystem (VCS)**

- Protokoliert Änderungen
- Ermöglicht Arbeiten in Branches
- Ermöglicht zurückspringen auf ältere Versionen.



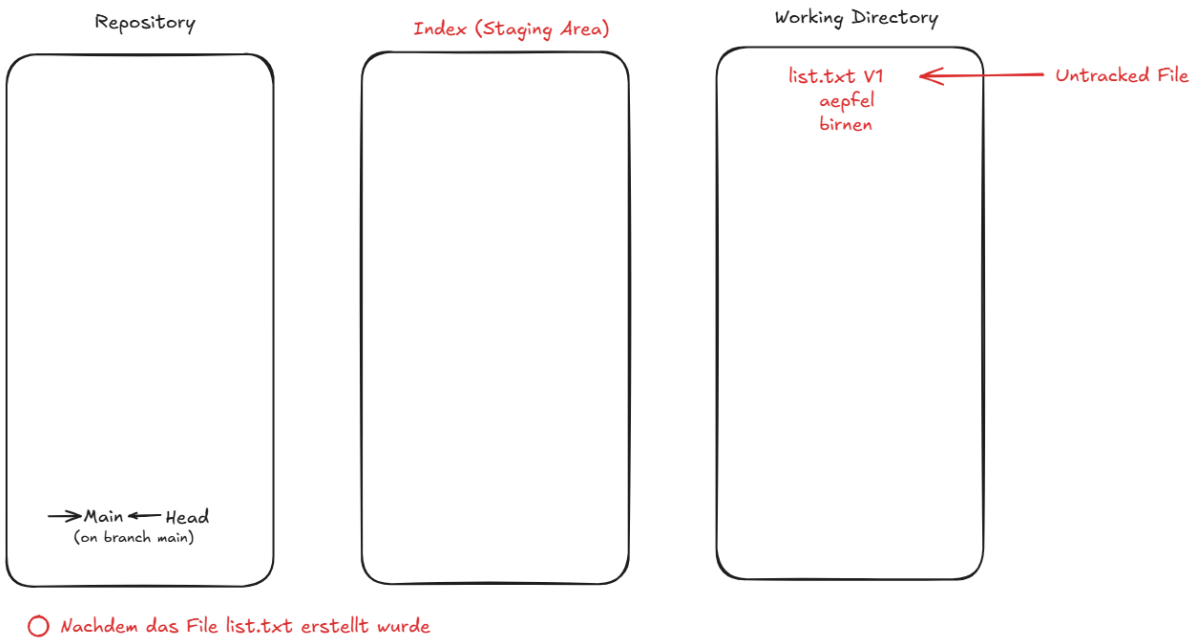
Git hat 3 Layers von Einstellungen

- SYTEM-Einstellungen (Standard)
  - USER-Einstellungen (Usereigene Einstellungen (überschreibt System))
    - LOKAL-Einstellungen (Repositoryspezifische Einstellungen (überschreibt Usereinstellungen))
      Allesamt sind in jeweils eigenen configfiles gespeichert.

Bei neuer Git-Installation ist es wichtig __Autorinformationen anzugeben (Benutzername & E-Mail)

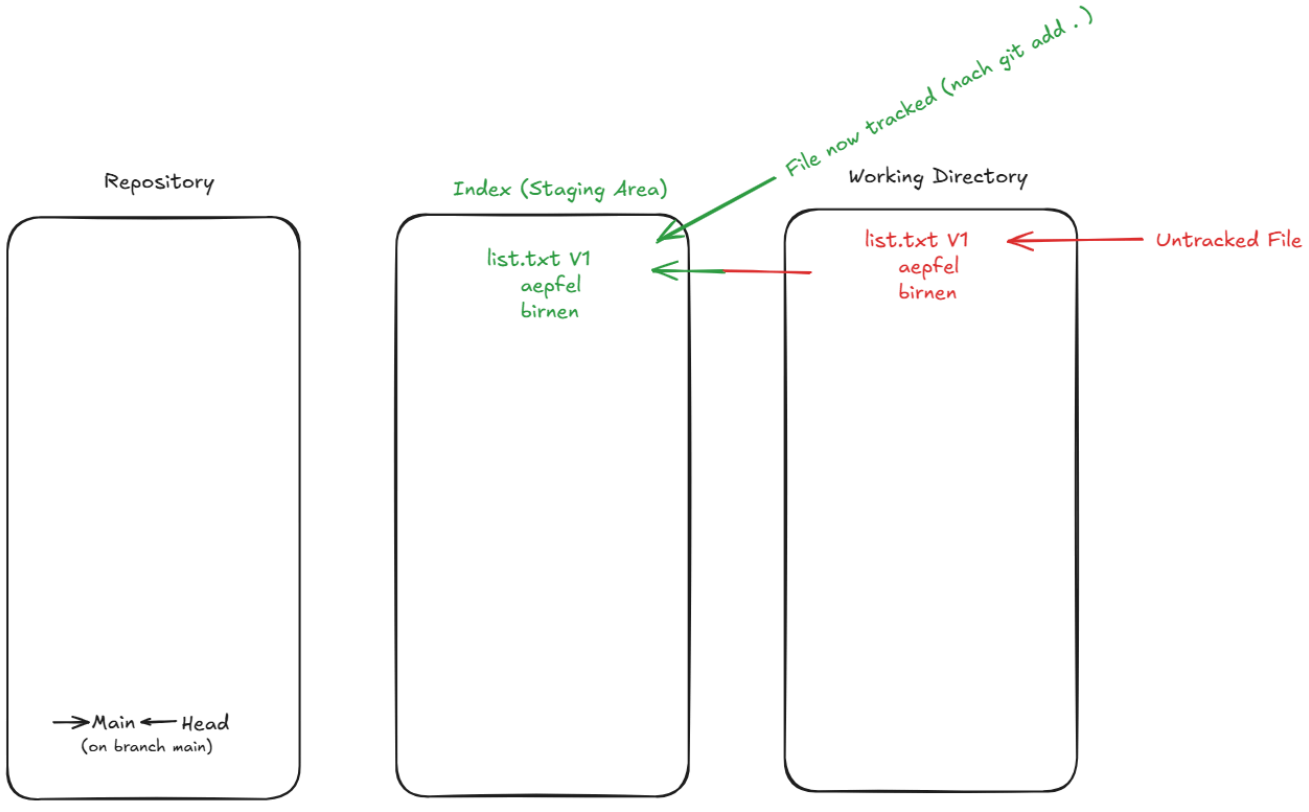
- git config --local user.email (add email here)
- git config --local user.name (add name here)

Wie entsteht eine neue Version?



Nachdem das File list.txt erstellt wurde

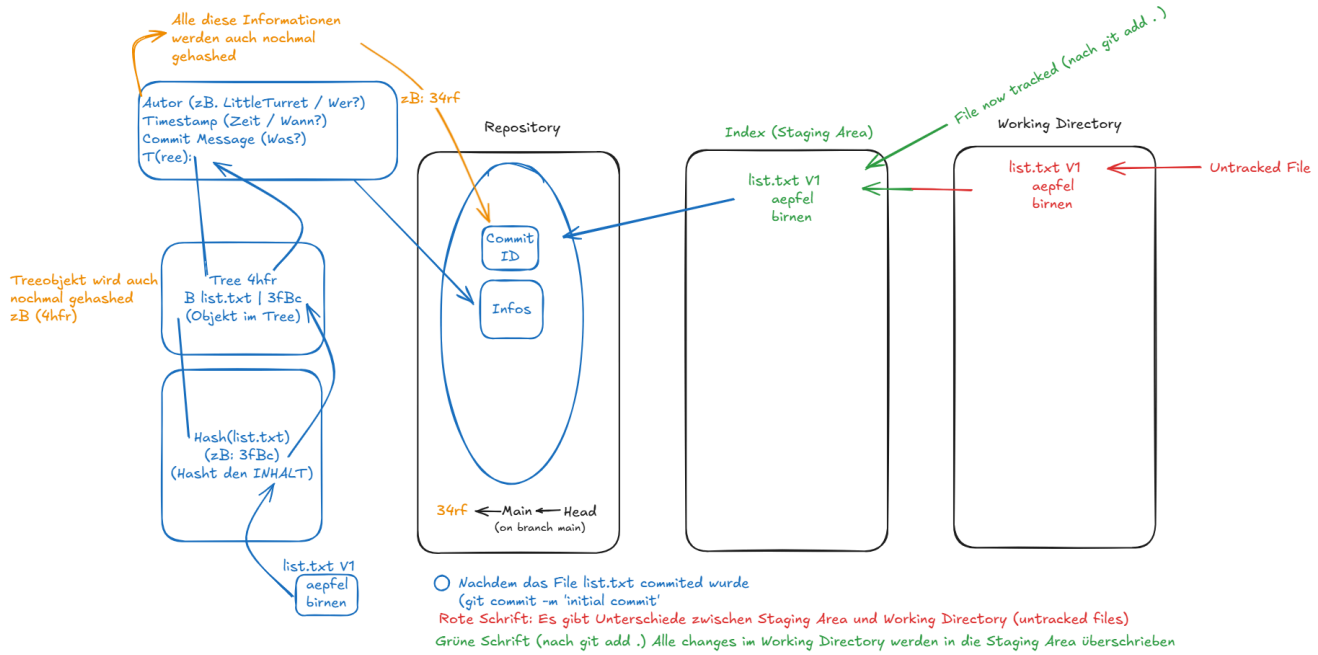
Die rote Schrift in git status gibt an, dass es Änderungen im working directory gibt, welche nicht im Index vorhanden sind. Diese würden beim Erstellen einer neuen Version nicht übernommen werden. Um diese Änderungen in den Index zu übernehmen, muss git add verwendet werden (git add . übernimmt jegliche Änderungen.



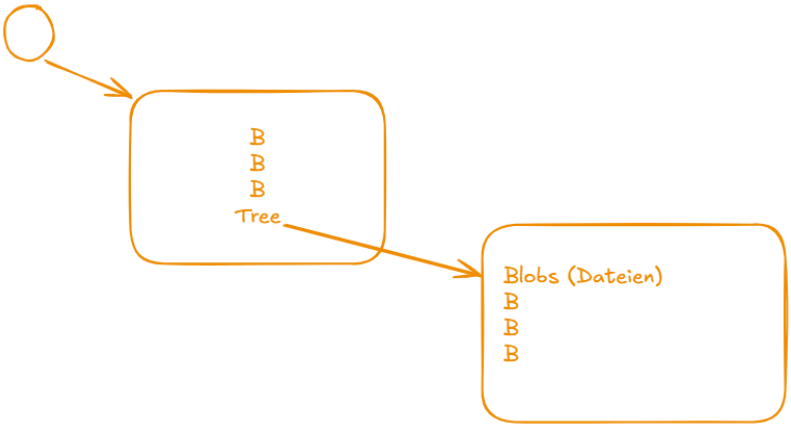
Nachdem das File list.txt gestaged wurde

Rote Schrift: Es gibt Unterschiede zwischen Staging Area und Working Directory (untracked files)
Grüne Schrift (nach git add .) Alle changes im Working Directory werden in die Staging Area überschrieben

Nachdem die Daten gestaged wurden, kann eine neue Version erstellt werden. Dies wird mit dem git commit Befehl gemacht, welcher eine commit message braucht.



Für jeden commit nach dem initial commit wird im commit auch der "Parent" übernommen (Der Parent ist die Commit ID des vorherigen Commits)



Die Dateien werden in einer Tree Struktur abgelegt, die Dateien selbst werden als Blobs bezeichnet und abgelegt.

GIT LOG zeigt alle erreichbaren Versionen des derzeitigen Branches an.

```
$ git log --graph --oneline --all --decorate
* af225c5 (HEAD -> main) initial commit
```

git log kann ein bissl special shit machen
weil das zum tippen nervig ist erstellen wir einen alias dafür

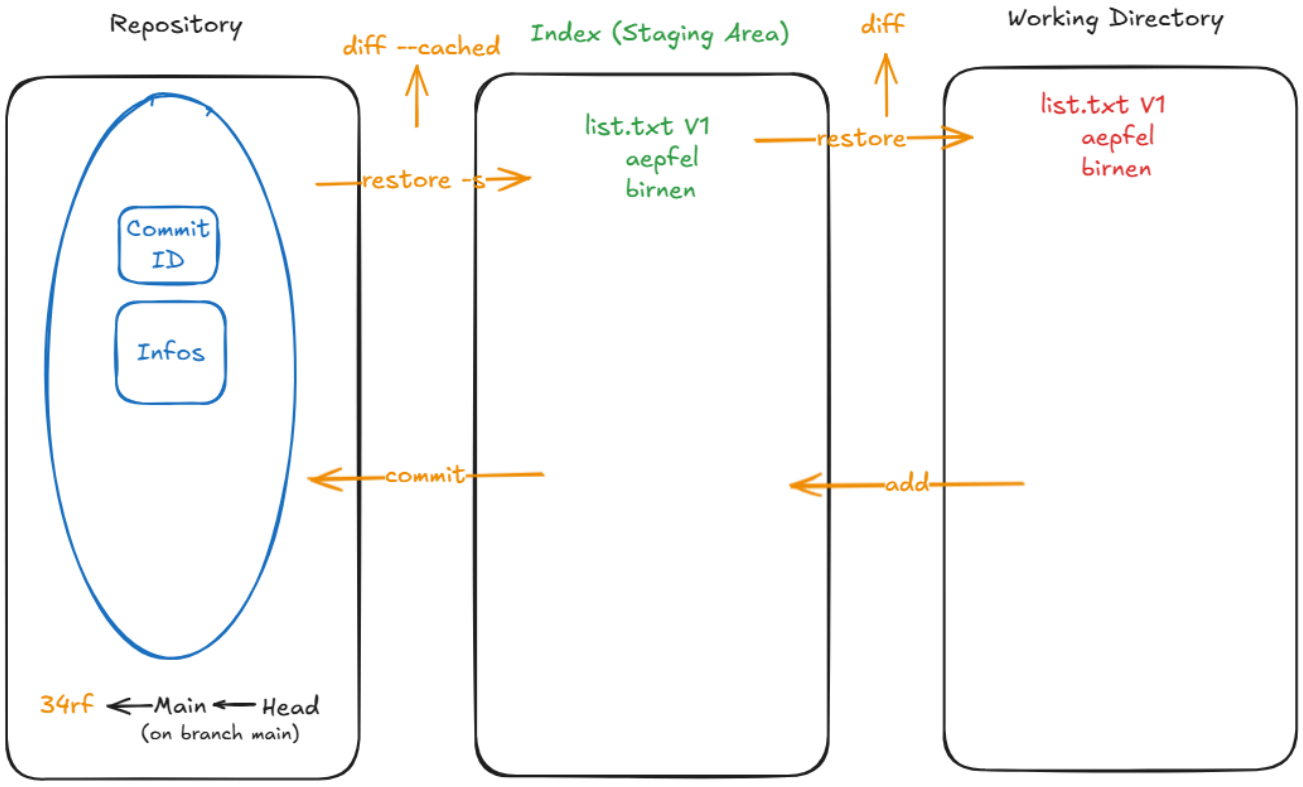
Ein alias ist ein keyword das beim ausführen einfach einen anderen vorher spezifizierten Befehl ausführen
der alias ist hier das keyword graph

mit **git restore (-s) filename** kann man die neueste Version in den Index schreiben (standardmäßig ist git restore -w, heißt der Index wird automatisch auch die work directory überschreiben)

mit **git checkout -- filename** kann das selbe erreicht werden
mit **git checkout -- branchname** kann man aber auch branch wechseln
git checkout ist n bissl nen weirdes teil

git kann keine leeren Verzeichnisse versionieren, es braucht immer einen dateiinhalt
wenn man das doch möchte, kann man mit **.gitkeep** eine unsichtbare Datei erstellen damit git mit dem directory arbeiten kann.

mit **git mv** und **git rm** kann man löschen und umbennen gleich auf der Indexebene durchführen.

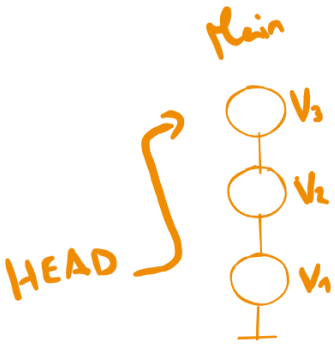


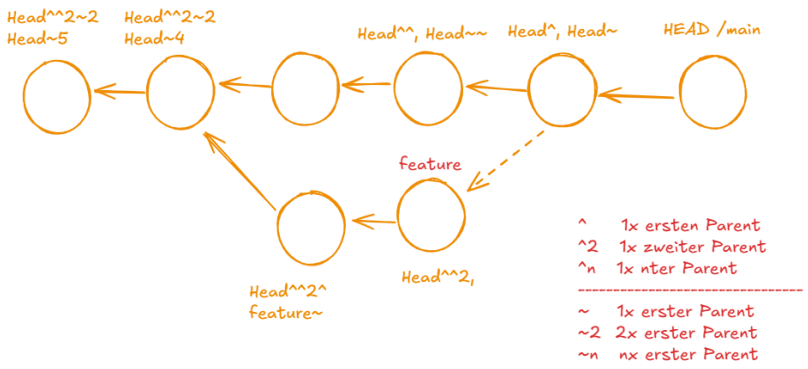
4.11.2025

Der HEAD ist ein Zeiger auf den aktuellen Arbeitsbranch (std. Main)

Um in Branches zu navigieren muss man den Commit den man haben möchte über den HEAD Pointer referenzieren
(HEAD/main -> aktueller Commit)
Mit den Zeichen '^' oder '~' kann man ältere Versionen aufrufen

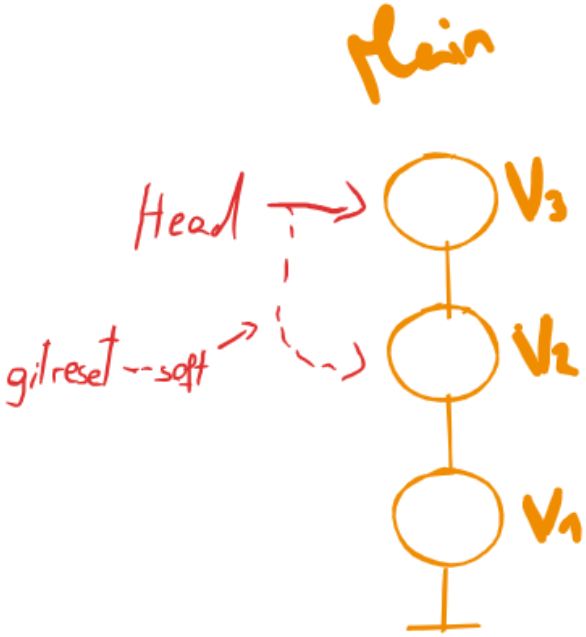
Bewegen innerhalb der Commit Timeline:





Mithilfe von **git reset** kann ich auf einen älteren Commit rückstellen

- --soft (Soft-Reset) Ein Soft-Reset schiebt nur den HEAD Zeiger auf den gewünschten Commit zurück
- --hard(Hard-Reset)



Hier wäre der korrekte Reset Befehlt `git reset --soft HEAD^ V3 exisitiert noch, aber es gibt keine direkte Referenz mehr Sollte man doch wieder auf den neuen commit wollen braucht man die ID davon. Diese kann man zB mit` git graph --reflog`` herausfinden. (--reflog zeigt alle commits)
(Macht basically die neueren git commit Befehle ungeschehen)

Hat man die ID, kann man sie wieder im **git reset** angeben um den Zeiger zurückzuschieben
`$ git reset --soft CommitID`

Als zweite Option für git reset gibt es
`$ git reset --mixed HEAD^`
Der mixed reset schieben wir den Zeiger auf den gewünschten commit. **Zusätzlich werden alle Dateien die in dem Commit vorhanden waren in den Index verschoben (er wird**

auch zurückgesetzt)
(Macht basically die neueren **git commit & git add** Befehle ungeschehen)

Konsole nach dem GIT Mixed:

```
$ git reset --mixed HEAD^
Unstaged changes after reset:
M       list.txt

User@DESKTOP-HaaM2005 MINGW64 /c/Users/Public/wse-ws25/shopping-list (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   list.txt

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

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

```
User@DESKTOP-HaaM2005 MINGW64 /c/Users/Public/wse-ws25/shopping-list (main)
$ ls -a
./  ../  .git/  .gitignore  lib/  list.txt  tmp.txt
```

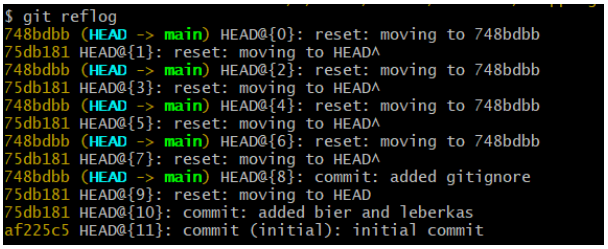
Im working repository ist es noch vorhanden, allerdings nicht mehr im Index

Als letzte Option für git reset gibt es
`$ git reset --hard HEAD^`
Der hard reset schieben wir den Zeiger auf den gewünschten commit. **Zusätzlich werden alle Änderungen in Index und Working Directory überschrieben!)**
(Macht basically die neueren git commit & git add__ Befehle ungeschehen und alle lokalen Änderungen gehen verloren!!!)

Ablauf der verschiedenen reset Funktionen

Der Befehl
`$ git reflog`
zeig alle Bewegungen innerhalb der commit timeline an.

Beispiel aus der Konsole:



Arbeiten mit Branches

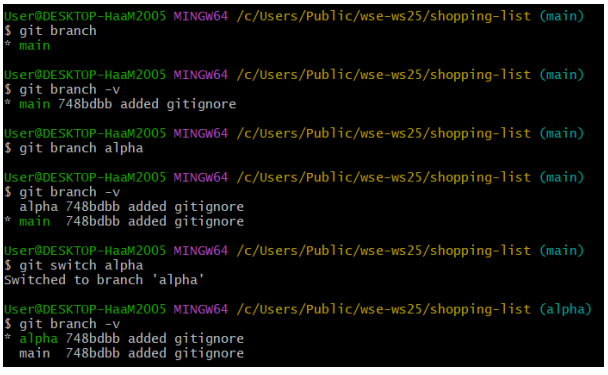
'Der Headzeiger markiert den aktuellen Arbeitsbranch'

Mithilfe des Befehles **git branch** kann ich eine Liste aller Branches anzeigen lassen, und auch neue Branches erstellen

```
$git branch
//Zeigt Liste der Branches
$git branch -v
//Zeigt Branchliste mit aktuellen commits
$ git branch BranchName
Erstellt einen neuen Branch BranchName
```

Um auf einen neuen Branch zugreifen zu können gibt es den **git switch befehl**
`git switch BranchName`
Diese Aktion ändert den HEAD Zeiger. Dieser zeigt jetzt nicht mehr auf main, sondern auf BranchName

Beispiel aus der Konsole:



Der Befehl
`git show`
ermöglicht es, bestimmte Informationen und Dateien anzuzeigen. Damit kann man auch Dateien aus anderen Branches einsehen.
zB
`git show BranchName: list.txt`
zeigt die list.txt datei im Branch BranchName an.

Mit dem Befehl
`git stash`
können lokale Änderungen vorübergehend gespeichert werden. Das ist vor allem nützlich, wenn man einen Branch wechseln möchte, aber dadurch lokale Änderungen verloren gehen würden.
Prinzipiell werden die Stashed in einem Stack abgelegt, man kann die stashes aber auch benennen und über
`git stash apply`
gewünschte Stashversionen aufrufen.

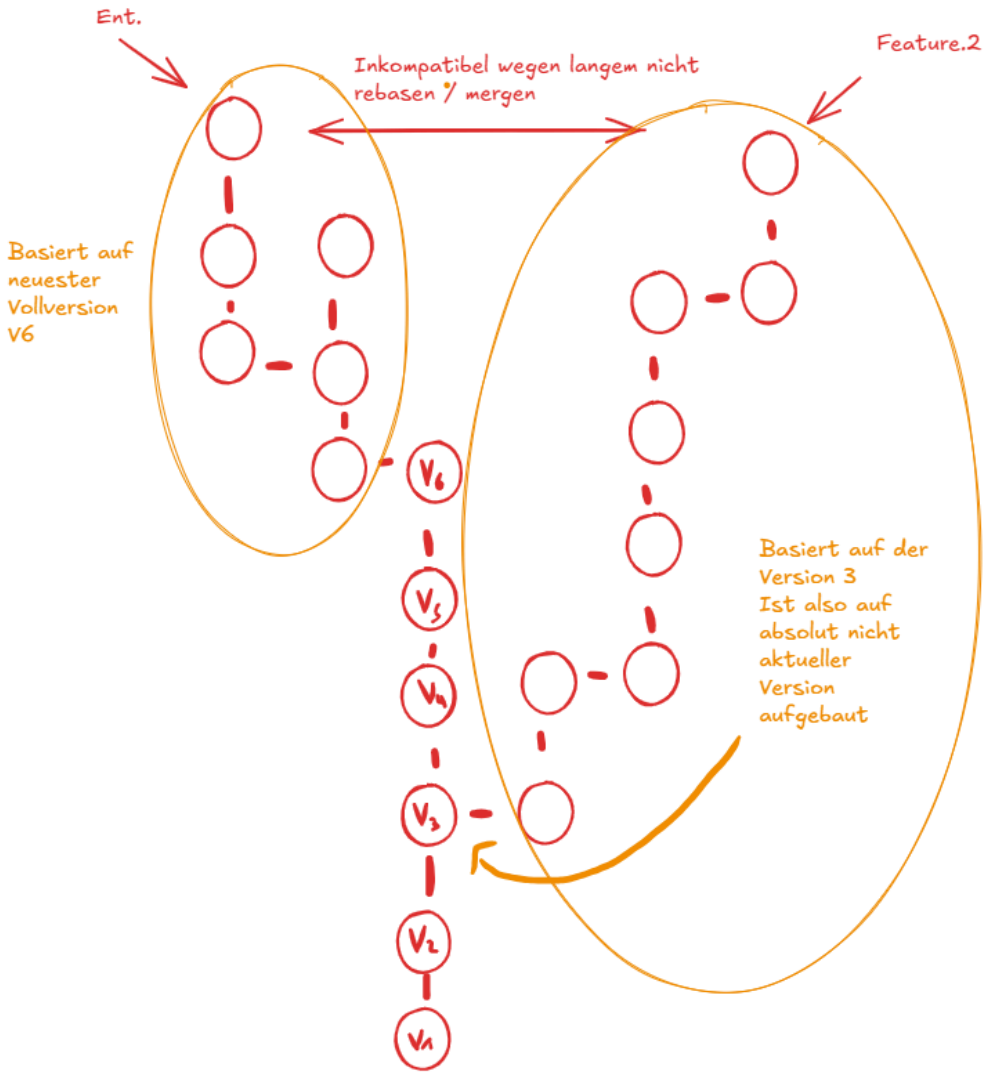
Will man die Änderungen aus dem Stash wieder ins working directory übertragen, benutzt man den Befehl
`git stash pop`

Der Befehl
`$ git merge`
verschmilzt zwei Branches zu einem neuen commit. Der aktuelle Arbeitsbranch wird automatisch als Zielbranch betrachtet, den Quellbranch muss man im Befehl angeben
`$ git merge alpha`
Der neue commit ist nun in der Timeline des **Zielbranches** angesiedelt. (Hier im Beispiel also im Main)

FF-Merge
Treeway Merge
Transform Threeway to FF mit Rebase
Merge Conflicts

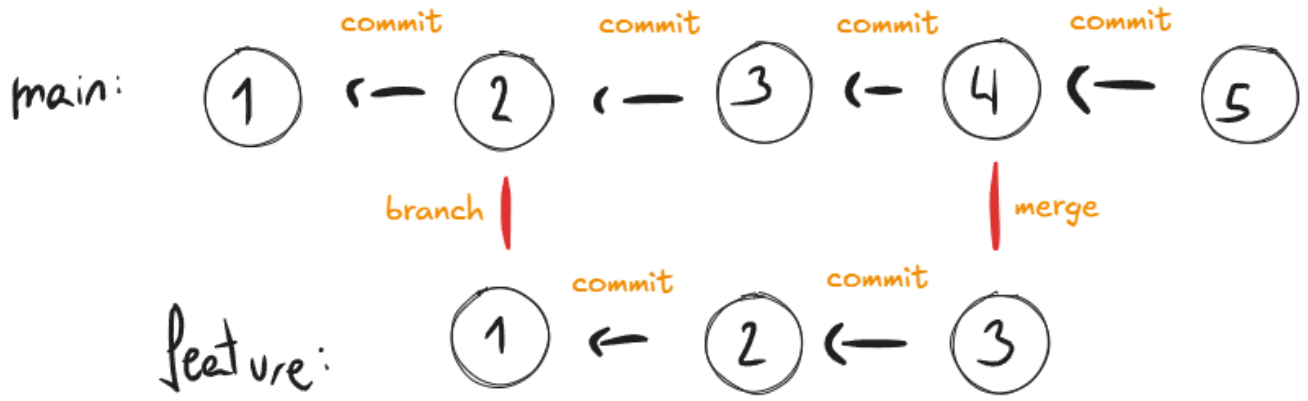
14.11.2025

Integration Hell



Um zu verhindern, dass verschiedene Branches zu weit auseinandergehen, müssen die Änderungen häufig in den Main Branch übernommen werden. Dies kann man entweder mit Branch merging, oder mit einem `git rebase` machen.

Gerade wenn man einen Branch hat, der aus einer älteren Vollversion der Main-Branches stammt, sollte man diesen in regelmäßigen Abständen mit einem Rebase auf die neueste Vollversion ziehen oder synchronisieren.
Wichtig ist es, dass man das nebenbei immer wieder tut, um **Integration Hell**, also das übermäßige manuelle Zusammenfügen und Mergekonflikte, zu vermeiden.
Auch die Zusammenhänge an sich, wie das Verhalten von bestimmten Funktionen/Prozeduren, können zwischen Branches divergieren, was nicht auffällt, wenn man nach langer Zeit mergt. Das führt natürlich, wenn man weitere features auf den veralteten Funktionen aufbauen, zu sehr zeitaufwendigen Reparaturen.



Arbeiten mit GitHub

GitHub ist ein Netz aus Servern, auf welchen man Repositories ablegen, diese absichern und teilen kann. Die repositories auf GitHub sind **bare repositories**, also repos ohne eine working copy

Nach dem Erstellen eines GitHub Repos ist ein leeres Repo erstellt worden. Um auf dieses zugreifen zu können, muss man einen **remote origin** mit dem Repositorylink angeben. Damit weiß das Repository, von wo es später Daten ziehen bzw. wohin es Daten schieben kann.

Git Push

Um Daten vom lokalen Repositories auf GitHub zu laden, wird `git push origin main` verwendet. (Es werden alle nicht synchronisierten commits hochgeladen und wie bei einem merge auf Konflikte geprüft.)

`git push` alleine kann nicht verwendet werden, weil git nicht weiß, auf welchen remote branch es die Daten schieben kann. Der remote ist ohne die explizite Angabe **nicht mit dem local branch verbunden.**

Local / Remote (Tracking) Branches

Es existieren 4 verschiedene Arten von Branches in Git / GitHub

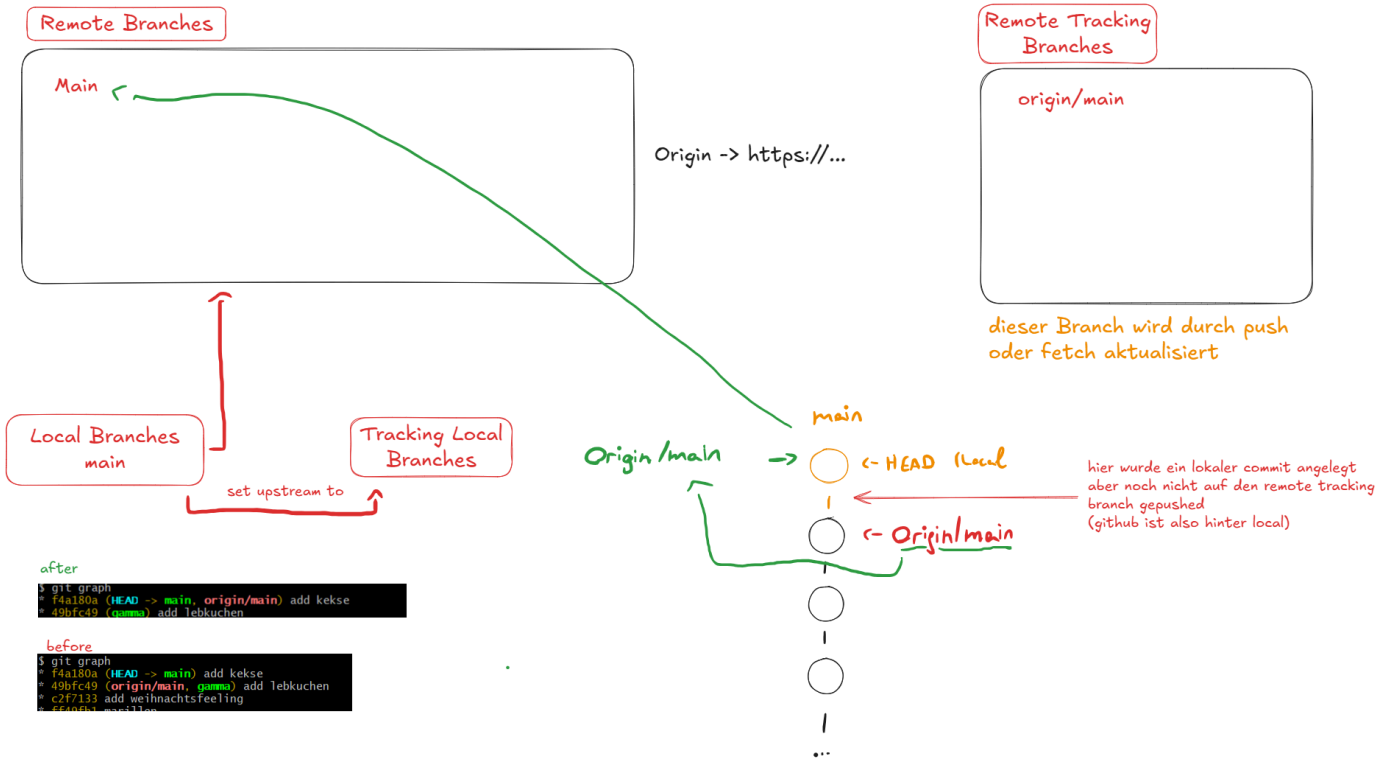
- Local Branches
  - Lokale, nicht getrackte Branches
- Remote Branches
  - Auf dem Server liegende Branches
- Local Tracking Branches
  - Lokale Branches, welche mit einem Remote Tracking Branch verknüpft sind
- Remote Tracking Branches
  - Tracker, welcher bei fetch und push die Änderungen von lokalen & lokalen tracking branches auf die remote branches synchronisiert.

Um den lokalen Branch mit dem remote Branch zu verbinden, muss ich ihn zu einem **tracking branch** umformatieren. Dies geht mit:

`git branch --set-upstream-to=origin/main`
(Entfernen kann man diese referenz mit:)
`git branch --unset-upstream`
Durch diese Änderung wird main zu einem **tracking local branch**. Dies bedeutet, dass man nicht mehr angeben muss, wohin man pushen will, da der remote bereits mit dem tracking branch verbunden ist.
Dies hat auch den Vorteil, dass `git status` anzeigen kann, ob der lokale Branch vor oder hinter dem remote Branch liegt.

main trackt origin/main

Remote tracking branch und der local tracking branch werden nur bei fetch und push Befehlen synchronisiert, nicht dauerhaft live!



(Hier fehlt die Umwandlung von main in einen Tracking branch.)

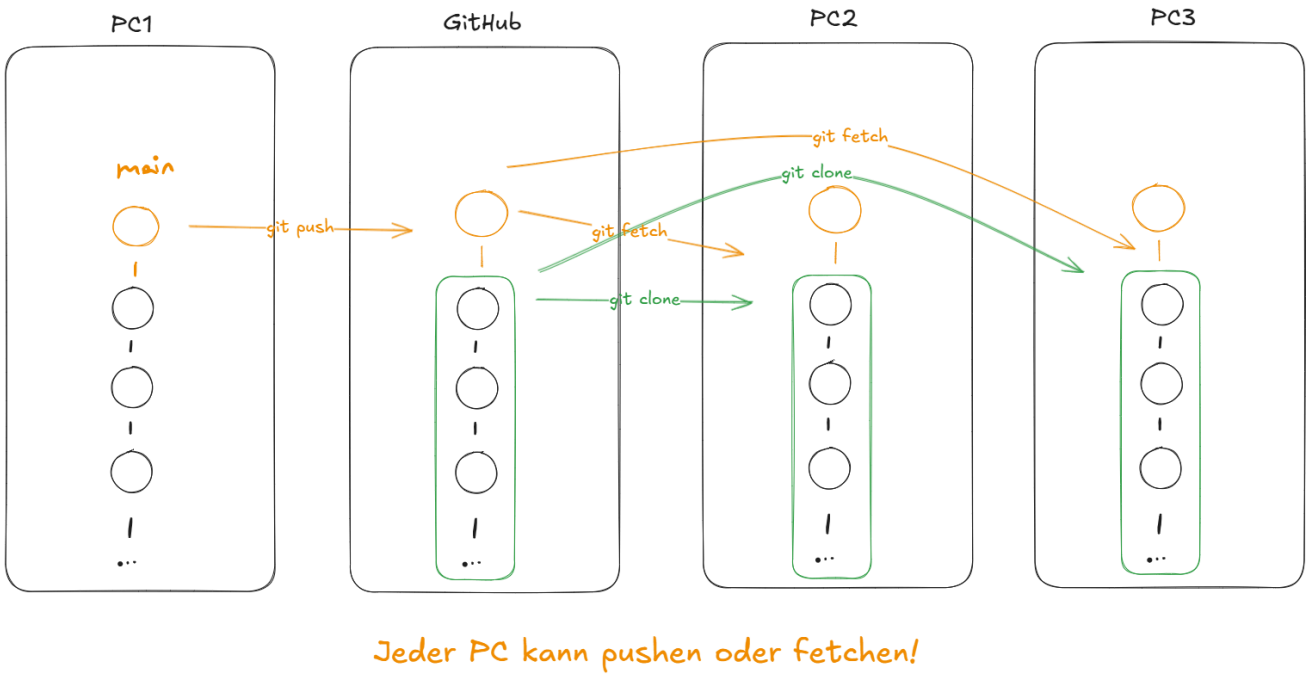
Im remote repo können auch neue branches erstellt werden (dies erstellt auch einen neuen remote tracking branch, analog zu origin/main)
`git push origin main:NewBranchName`
(Und diese können auch wieder gelöscht werden)
`git push -d origin NewBranchName`

Git Clone

Mit `git clone https://...` kann ich ein Repository von GitHub in ein neues Verzeichnis (muss kein git repo sein) kopieren. Hierbei wird ein Working Directory und ein Working

System angelegt, um mit dem neuen Repo arbeiten zu können.

Von den neuen Repositories aus kann ich auch neue Änderungen nach GitHub pushen, oder neue Versionen von GitHub auf das geklonte Repo herunterladen.



Git Fetch / Git Pull

`git fetch` zieht Änderungen vom remote auf das lokale Repository. Bestehendes wird aktualisiert, neue Branches werden als remote tracking branch gedownloaded (und erst bei einem switch in diese als lokaler tracking branch angelegt.) Bei Fetch von einem remote wird der remote tracking branch verschoben nicht aber den aktiven Branch (weil das macht ja merge).

Git Fetch ist ein Teil des des `git pull` Befehls.
`git pull = git fetch + git merge`

!Git Fetch ändert die Positionen der remote tracking branches
Git pull ändert die position der remote tracking branches UND ändert den Head auf den neuesten Commit

Git Revert

`git revert` ist in der Lage, einen neuen Commit zu erstellen, welcher zuvor gemachte Änderungen wieder rückggäning machen kann. Das ist relevant, weil es mit der Alternative `git push -f` zu Problemen mit der Timeline kommen kann.