Git Tutorial

Git installieren

Link zur Git Homepage

- installieren von git für Windows
- macOS und Linux sollten bereits mit vorinstalliertem Git kommen

Git einrichten

- git config-Datei konfigurieren
- überprüfen ob config eingerichtet ist

```
git config --list
```

- mindestens E-mail und User-Name muss eingerichtet werden

```
  # User Name erstellen
  git config --global user.name "<Your Name>"

  # E-Mail erstellen
  git config --global user.email "<your email address>"
```

- optional direkt immer main als Haupt-Branch festlegen

```
git config --global init.defaultBranch <name>
```

Git Repository initialisieren

- dieser Befehl initialisiert das Git Repo mit dem Haupt-Branch 'master'.

```
git init
```

- um den Namen nachträglich zu 'main' zu ändern

```
git branch -m main
```

- um direkt bei der Initialisierung den Hauptbranch main zu nennen:

```
git init -b main
```

- wenn Git Repo initialisiert kann der status abgefragt werden

```
git status
```

Dateien tracken bzw. zu Staging hinzufügen

- Datei im Arbeitsverzeichnis anlegen und mittels Git command zu Staging hinzufügen

```
git add <datei.endung_der_datei>

# oder alle Dateien auf einmal
git add .
```

- Datei ggf. wieder unstagen

```
git rm --cached <file_name>
```

Der erste Commit

- mit einem Commit werden alle Dateien, welche sich gerade im Staging befinden in das lokale Repository (Storage), mit dem aktuellen Staging Zustand, aufgenommen (Snapshot des aktuellen Zustands)

```
git commit -m "<Deine Commit Message>"
```

Der zweite Commit

- lege eine neue Datei Deiner Wahl an und füge diese zum Staging hinzu mit git add und commite anschließend diesen neuen Stand Deines Arbeitsverzeichnisses

git log

- git log gibt eine Übersicht über alle commits mit den dazugehörigen Daten
- optional nimmt git log die `--oneline` Flag um die Übersicht kürzer anzuzeigen

```
git log --oneline
```

Reise in die Vergangenheit

- um einen vergangen Snapshot bzw. den Zustand des Git-Projektes anzuschauen:

```
git checkout <hash/versionsnummer>
```

- dort kann herumexperimentiert werden und verschiedene Dinge getestet werden ohne das Hauptprojekt zu verändern
- um wieder in das Hauptprojekt bzw. die Gegenwart zurückzukehren

```
git checkout main
# oder
git checkout -
```

ACHTUNG

- manchmal möchtest Du Änderungen verwerfen und zu einem früheren Zeitpunkt Deines Projetes zurückkehren, nicht nur um zu schauen wie es zu dem Zeitpunkt aussah, sondern um von dem Zeitpunkt aus neu zu starten

```
git reset --hard <commit-hash>
```

- wenn die Änderungen, die danach gemacht wurden nicht gelöscht werden sollen, dann wird stattdessen aus dem vergangen commit ein neuer Branch abgeleitet und nicht resetet

Branching

durch Branching wird eine Kopie des aktuellen Zustands des Haupt-Branch (hier: main) erstellt, z.B. feature-branch. In diese Feature-Branch wird nun eine neue Funktionalität des Projektes entwickelt

- neuen branch erstellen:

```
git branch <branch_name>
```

- in den neuen branch wechseln

```
git checkout <branch_name>
```

- besser und neuer

```
git switch <branch_name>
```

- Branch anlegen und direkt in den neuen Branch wechseln

```
git switch –c <new_branch_name>

# oder mit checkout
git checkout –b <new_branch_name>
```

Merging

- Änderungen sowie die Commit-Historie eines Branches mit einem anderen Branch zusammenführen.
- z.B.: wenn ein neues Feature für die App programmiert wurde und nun in den Hauppfad / Haupt-Branch integriert werden soll
- ein merge wird immer von dem Branch ausgeführt, in welchen die Änderungen überführt werden sollen
- d.h ich möchte Änderungen von feature_1 in main überführen, also wird der Merge auf dem Main-Branch ausgeführt

```
git merge <Feature–Branch>
```

- nun ist im Haupt-Branch auch die komplette Commit-Histore aus dem Neben-Branch zu sehen.
- optional kann nun der Feature-Branch gelöscht werden

```
git branch –d <Feature–Branch>
```

Merge-Konflikte und Auflösung

Entstehung eines Konfliktes

Ein Merge-Konflikt tritt auf, wenn Git nicht automatisch entscheiden kann, wie zwei unterschiedliche Änderungen an der gleichen Datei zusammengeführt werden sollen. Das passiert typischerweise in folgenden Situationen:

1. Gleiche Datei wurde in zwei verschiedenen Branches geändert

- Beispiel: Du bearbeitest eine Zeile in feature_branch, während ein Kollege dieselbe Zeile in main geändert hat.
- Git kann nicht automatisch entscheiden, welche Version übernommen werden soll.

2. Gleiche Datei wurde gelöscht & geändert

- Ein Branch löscht eine Datei, während der andere Änderungen daran macht.

3. Unterschiedliche Änderungen an denselben Codeblöcken

- Zwei Entwickler haben dasselbe Code-Segment auf unterschiedliche Weise bearbeitet.

Auflösung eines Konfliktes

1. Merge durchführen & Konflikt erkennen

```
git merge <branch-name>
```

- Falls Konflikt

```
CONFLICT (content): Merge conflict in <filename>
Automatic merge failed; fix conflicts and then commit the result.
```

2. Konfliktstellen in Dateien bearbeiten

- Git markiert den Konfliktbereich in den betroffenen Dateien

```
<<<<<<< HEAD
Code aus aktuellem Branch
=======
Code aus dem zu mergeenden Branch
>>>>>>> feature_branch
```

- Manuelle Entscheidung der Änderungen

3. Konflikt als gelöst markieren

```
git add <filename>
```

- Git wird mitgeteilt, dass die Datei wieder in Ordnung ist

4. Merge abschließen

```
git commit -m "Merge resolved"
```

- damit ist der Konflikt offiziell aufgelöst und der Mege abgechlossen

.gitignore – Was ist das und wofür wird sie genutzt?

📌 Was ist eine `.gitignore`?

- Eine **Spezialdatei** in einem Git-Repository.
- Enthält eine Liste von **Dateien und Verzeichnissen**, die **nicht** in Git getrackt werden sollen.

🛠️ Was macht die `.gitignore`?

- **Verhindert, dass unerwünschte Dateien ins Repository gelangen**.
- Spart Speicherplatz und hält das Repository sauber.
- Reduziert das Risiko, sensible oder überflüssige Dateien versehentlich zu committen.

📄 Typische Inhalte einer `.gitignore`

- **Build-Dateien und Binärdateien** (`*.exe`, `*.o`, `dist/`)
- **System- und Editor-Dateien** (`.DS_Store`, `Thumbs.db`, `.vscode/`)

- **Umgebungsdateien und Secrets** (`.env`, `config.json`)
- **Dependency-Ordner** (`node_modules/`, `venv/`, `__pycache__/`)

🚀 Beispiel `.gitignore` für ein Python-Projekt

```
__pycache__
*.log
venv/
```

Falls eine Datei bereits getrackt wurde und ignoriert werden soll

```
git rm -r --cached <filename>
```

Verknüpfen mit einem remote Repository (Github)

Github Webseite

1. Auf Github ein leeres Repository anlegen
2. Den 'HTTPS' Link zu dem neuen Remote Repo kopieren
3. im lokalen Repository in der commandline folgende Befehle ausführen:
    1. `git remote add origin <Link zu Remote Repository>`
    2. `git branch -M main`
4. das lokale Repository auf das remote Repository pushen
    1. `git push origin main`

- `origin` ist der Name unserer Remote Verbindung

Remote Repository lokal clonen

Ein bestehendes Online Repository auf den lokalen Computer kopieren und dann dort Änderungen machen

1. Auf dem Github Repository den Repository Link kopieren
2. Auf dem lokalen Computer einen Ordner anlegen in welchen das neue Repository kopiert werden soll.
3. zum 'klonen' des online Repos folgenden Befehl ausführen: `git clone <Link zum Repo>`
4. damit wird lokal eine Kopie des Github Repos gedownloaded.
5. Dieses Repo ist auch schon direkt mit dem Github Repo verknüpft und enthält alle bereits getätigten Commits.
6. Mit `git push origin main` können dann lokale Änderungen wieder auf das Github Repo hochgeladen werden, SOFERN Ihr die Erlaubnis hierfür habt.

Änderungen eines Remote Repos auf Lokales Repo übertragen

```
git pull origin main
```