## Kurzbeschreibung:

Git ist ein Versionierungstool bzw. ein DVCS (Distributed Version Control System) und dient dazu, bei längeren Projekten alle Versionen zu verschiedenen Zeitpunkten sauber zu dokumentieren.

Um Teamarbeit zu ermöglichen oder verschiedene Ideen parallel zu entwickeln, kann man bei Git – ähnlich wie bei einem Baum – am Stamm (dem *main*- oder *master*-Branch) arbeiten und sich von dort aus verzweigen. Das bedeutet, man kann an unterschiedlichen Ästen bzw. Branches arbeiten (z. B. jedes Teammitglied an einem eigenen) und die Ergebnisse, die man behalten möchte, später wieder zusammenführen (mergen).

Das typische Vorgehen sieht folgendermaßen aus:  
Jedes Teammitglied arbeitet in einem eigenen Branch (Zweig). Änderungen und neue Dateien werden dabei zunächst lokal über Git gespeichert.  
Regelmäßig sollte der eigene Branch zur Einsicht und Zusammenarbeit in ein zentrales, gemeinsam genutztes Verzeichnis (Remote-Repository) hochgeladen werden. So können andere Teammitglieder Verbesserungsvorschläge einbringen oder notwendige Änderungen anregen.

Wenn der Branch ausreichend geprüft und für gut befunden wurde, kann er mit dem Hauptzweig (*main* oder *master*) zusammengeführt (gemergt) werden.  
Damit alle Teammitglieder auf dem aktuellen Stand bleiben, sollten sie regelmäßig den neuesten Stand des Haupt-Branches abrufen (pull).

Auf diese Weise entsteht im Laufe der Zeit ein gut strukturiertes Projekt. Es bleibt jederzeit möglich, auf ältere Versionen zuzugreifen oder sie wiederherzustellen.

## Arbeitsweise:

Git speichert bei jedem *Commit* (Speichervorgang) eine Art Snapshot des gesamten Dateiordners bzw. des zu überwachenden Projekts. Diese Snapshots werden chronologisch geordnet im Repository abgelegt.

Um sinnvoll mit Git zu versionieren, ist es essenziell, jeden Commit mit einer kurzen, aussagekräftigen Beschreibung zu versehen. Diese sollte idealerweise alle Änderungen seit dem letzten Stand sowie ggf. die dahinterstehende Intention enthalten. So wird es möglich, später auf eine bestimmte Version zurückzublicken und nachzuvollziehen, was sich verändert hat und warum.

Wie bereits erwähnt, können in Git mehrere Personen – jeweils in ihrem eigenen Branch – unabhängig voneinander arbeiten. Auch in diesen Branches wird bei jedem Commit ein Snapshot erstellt und im Repository „abgelegt“. Im gemeinsamen Remote-Repository befinden sich somit die Snapshots aller Branches sowie des Haupt-Branches (*main* oder *master*) nebeneinander.

Beim Merge-Vorgang wird ein bestimmter Branch mit dem *main*-Branch zusammengeführt.

Jede\*r Mitarbeitende verfügt – nach einem *Pull* vom Remote-Repository – über eine vollständige, lokale Kopie des gesamten Projekts. Dadurch kann das Repository jederzeit auf den aktuellen Stand gebracht oder auf einen früheren Zustand zurückgesetzt werden.

# GIT einrichten mit GITHUB etc:

1. Melde dich bei GITHUB an. Das ist eine Internetseite, die es ermöglicht öffentlich zugängliche Repositorys anzulegen und zu verwalten. Benötigt wird dafür ein Username und eine Emailadresse.
2. Installiere GIT und öffne GITBASH (einfach unten in Suchleiste eingeben)
3. Nun Username und email (zwingend die gleichen wie in GITHUB angeben)
   1. git config --global user.name "Dein Name"
   2. git config --global user.email [deine@email.de](mailto:deine@email.de)
4. Prüfen ob alles passt
   1. git config –list
5. Nun verknüpfe dein lokales GIT mit GITHUB und ssh
   1. ssh-keygen -t ed25519 -C [deine@email.de](mailto:deine@email.de)
   2. enter drücken
   3. enter drücken
   4. cat ~/.ssh/id\_ed25519.pub
   5. alles inklusive mail kopieren
6. In GITHUB auf Profil > Einstellungen > SSH.. > add new SSH > Text einfügen und speichern
7. In GITBASH prüfen
   1. ssh -T [git@github.com](mailto:git@github.com)
   2. „yes“ eingeben

## Erstes Repo bauen und in GITHUB anzeigen lassen

Im allgemeinen möchte man mit GIT nur bestimmte Dateien tracken. Deshalb benötigen wir dafür einen Ordner in dem alle Dateien liegen. Darin werden wir nun unser Repo also einen (.git) Ordner legen. Das heißt auch, dass nur Dateien die in dem Ordner liegen getrackt werden können.

1. Optionen:
   1. Ziehe den Ordner per Drag and Drop in GITBASH
   2. cd dateipfad
2. git init
3. Im Ordner oben bei Ansicht ausgeblendete Elemente anzeigen und prüfen ob ein .git Ordner erscheint
4. git add . // „.“ Fügt alles aus dem Ordner zu GIT hinzu
5. git commit -m "Notiz" // erster Commit fügt den Ordner nun lokal zur Datenbank
6. Prüfe ob alles hinzugefügt wurde
   1. cd dateipfad
   2. ls -a //list files all zeigt also alle Dateien aus dem Repo

Jetzt erstellen wir das Repo in GITHUB

1. In GITHUB oben rechts auf „+“ dann „New repository“
2. Namen des Projektes angeben. Evtl selben wie den Ordnernamen
3. Privat oder public
4. Das README-Zeug nicht ändern
5. Create Repository
6. In GITBASH:
   1. git remote add origin [git@github.com:deinBenutzername/dein-repo-name.git](mailto:git@github.com:deinBenutzername/dein-repo-name.git)

Erster PUSH

1. git branch -M main
2. git push -u origin main // das -u verlinkt nun den lokalen Branch mit GITHUB

// In Zukunft reicht es git push zu schreiben

## Wichtige Begriffe:

**Modified:**

Dateien die geändert, aber noch nicht in lokaler Datenbank eingecheckt wurden.

**Repository**  
Ein Projektordner, der von Git verwaltet wird. Enthält alle Dateien und die gesamte Versionshistorie.

**Commit**  
Ein gespeicherter Versionsstand mit einer Nachricht, die beschreibt, was geändert wurde. Wird vorerst lokal gespeichert.

**Branch**  
Ein Entwicklungszweig, in dem unabhängig gearbeitet werden kann. Ermöglicht parallele Entwicklung an verschiedenen Funktionen oder Fehlerbehebungen.

**Merge**  
Das Zusammenführen von Änderungen aus einem Branch in einen anderen (z. B. in den Hauptbranch).

**Main (früher Master)**  
Der Hauptentwicklungszweig eines Projekts. Standardmäßig heißt er heute oft „main“.

**Clone**  
Herunterladen einer Kopie eines Remote-Repositories auf den lokalen Rechner.

**Push**  
Übertragen der lokalen Commits in das Remote-Repository, damit andere darauf zugreifen können.

**Pull**  
Aktualisieren des lokalen Repositories mit den neuesten Änderungen aus dem Remote-Repository.

**Remote Repository/ Origin**  
Ein zentrales Repository auf einem Server (z. B. GitHub, Bitbucket), das von mehreren Personen genutzt wird. Falls von dort geklont wird, lässt sich darauf mit origin referenzieren.

**Staging Area**  
Ein Zwischenspeicherbereich, in dem Änderungen gesammelt werden, bevor sie committet werden.

**Conflict (Merge-Konflikt)**  
Ein Versionskonflikt, der entsteht, wenn zwei Änderungen dieselbe Stelle im Code betreffen und nicht automatisch zusammengeführt werden können.

**Checkout**  
Wechseln zu einem anderen Branch oder zu einem bestimmten Commit.

**Fork**  
Eine Kopie eines Repositories, meist um unabhängig daran arbeiten zu können (z. B. in Open-Source-Projekten).

**Pull Request (PR)**  
Ein Antrag zur Zusammenführung von Änderungen aus einem Branch in einen anderen, meist mit der Möglichkeit zur Überprüfung und Diskussion.

## GIT-Bash/CMD-Befehle im Überblick:

|  |  |
| --- | --- |
| Befehl | Beschreibung |
| git init | Erstellt ein neues Git-Repository im aktuellen Verzeichnis |
| git clone <url> | Klont ein entferntes Repository (z. B. von GitHub) lokal |
| git status | Zeigt den aktuellen Status der Arbeitskopie (z. B. Änderungen, Staging-Bereich) |
| git add <datei> | Fügt eine Datei dem Staging-Bereich hinzu |
| git add . | Fügt alle Änderungen im Verzeichnis dem Staging-Bereich hinzu |
| git commit -m "Nachricht" | Speichert die Änderungen mit einer Commit-Nachricht |
| git log | Zeigt eine Historie aller Commits |
| git diff | Zeigt die Unterschiede zwischen Dateien/Commits |
| git branch | Zeigt alle lokalen Branches |
| git branch <name> | Erstellt einen neuen Branch |
| git checkout <name> | Wechselt zu einem anderen Branch |
| git checkout -b <name> | Erstellt und wechselt zu einem neuen Branch |
| git merge <branch> | Führt den angegebenen Branch in den aktuellen zusammen |
| git pull | Holt neue Änderungen vom Remote-Repository und merged sie |
| git push | Schickt lokale Commits ins Remote-Repository |
| git remote -v | Zeigt die verbundenen Remote-Repositories |
| git reset --hard <commit> | Setzt das Projekt hart auf einen früheren Commit zurück |
| git stash | Speichert ungesicherte Änderungen zwischen |
| git stash pop | Stellt gespeicherte Änderungen wieder her |
| git rm <datei> | Entfernt eine Datei aus dem Arbeitsverzeichnis und dem Staging-Bereich |
| git config --global user.name "Name" | Setzt den globalen Git-Benutzernamen |
| git config --global user.email "E-Mail" | Setzt die globale Git-E-Mail-Adresse |

## Commit-Änderungen

Es kann durchaus vorkommen, dass man ausversehen zu früh einen stand commitet und z.B. die Notiz fehlerhaft ist oder unvollständig. Oder es wurde vergessen ein File hinzuzufügen.

Um die Commit-history nicht zu überladen gibt es Möglichkeiten die manche Commits rückgängig zu machen bzw. zu überschreiben. Dadurch wird der ursprüngliche Commit nicht mehr in der Commit-history sein und vollständig ersetzt.

Das funktioniert jedoch nur bei lokalen Commits und sollte bei einem Push unterlassen werden. (Teammitglieder könnten so zu unterschiedlichen Ständen kommen, was zu Problemen führt)

Relevante Fälle und Anwendung:

1. Fehlerhafte/ mangelhafte Notiz oder fehlende Files:
   1. Nach dem fehlerhaften Commit die Änderungen an Files vornehmen und stagen
   2. git commit --ammend -m „neue Notiz“
2. File aus der staging area entfernen
   1. git reset HEAD filename //löscht die Datei mit Namen filename (z.B. text.txt) aus der staging area
3. Datei auf Stand vom letzten Commit zurücksetzen
   1. git restore -- filename // setze Datei filename auf Stand vom letztem Commit zurück
4. Datei auf Stand vom letzten Stage-Stand zurücksetzen
   1. git checkout --staged filename // setze Datei filename auf den letzten Staged Zustand zurück

# Commit-Notizen

Eine übersichtliche Struktur hilft den Überblick zu behalten. Folgende Konventionen helfen:

Struktur:

<typ>(optional: scope): <kurze erklärung>

Typ: Beschreibung:

feat Neues Feature

fix Bugfix

docs Nur Dokumentation geändert

style Code-Formatierung (kein Codeverhalten)

refactor Code geändert ohne Verhalten zu ändern

test Tests hinzugefügt oder geändert

chore Wartung oder Aufgaben ohne Logikänderung

perf Performance-Verbesserung

ci Änderungen an CI/CD-Konfiguration

Bsp:

feat(auth): füge Login-Funktionalität hinzu

fix(api): behebe Timeout-Bug bei GET-Anfragen

docs(readme): aktualisiere Projektbeschreibung

# Branching

Ein Branch ist eine parallele Version eines Projekts zu einem bestimmten Zeitpunkt, der sich weiterentickeln lässt.

Ein Bild, das Screenshot, Text, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Hier sieht man also, wie sich das Projekt vom Startzustand C0 immer weiterentwickelt. In C2 wurde das letzte mal auf den master gepusht. Von da entwickelt sich der Branch „iss53“ und der Branch „hotfix“.

Um zwischen Branches zu wechseln ist es immer sinnvoll zu committen.

1. Aktuellen Branch committen
2. git checkout -b hotfix //erzeugt den neuen Branch hotfix und wechselt dorthin
3. Änderungen vornehmen und committen

Man kann zwar auch ohne vorher zu committen den Branch wechseln, aber nur wenn die geänderten Dateien nicht im Konflikt mit dem Branch stehen in den man wechseln möchte.

# Merging und Mergekonflikte:

Nun möchten wir zwei Branches zussammenführen, z.B. hotfix mit master.

1. git checkout master //Wechsle auf den master-Branch um hotfix dann einzufügen
2. git merge hotfix //führt hotfix in master über
3. git branch -d hotfix //löscht hotfix, da er nicht mehr benötigt wird

Beim mergen können auch Konflikte auftreten. Das passiert wenn in beiden Branches eine Datei unterschiedlich an der gleichen Stelle geändert wurde. Dann kann git nicht entscheiden welcher Branch nun der richtige ist.

Deshalb ist folgendes wichtig:

1. Unterschiedliche Branches beschäftigen sich möglichst mit unterschiedlichen Aufgaben/Problemen um Mergekonflikte zu vermeiden.
2. Regelmäßig sollte ein Branch auf den aktuellen Stand gebracht werden oder nur kurz bestehen.

# Arbeitsablauf

Mit dem Wissen von vorhin lässt sich also ein allgemeiner Arbeitsablauf für Git beschreiben.

Grundvorraussetzungen sind hier ein Remote-Repo.

Folgende Regeln sollten beachtet werden um sinnvoll zu arbeiten

1. Jeder Aufgabe wird in einem eigenen Branch bearbeitet, oder mehrere kleine. Sinnvoll ist auch, wenn jede Person einen eigenen Branch hat.
2. Jede kleine abgeschlossene Aufgabe sollte commitet werden. (Bzw. allgemein regelmäßig commiten um später Änderungen schnell zu finden). Jeder
3. Commit sollte sauber dokumentieren was sich geändert hat.
4. Ist eine Aufgabe für einen Branch abgeschlossen, wird dieser mit dem Main oder anderen gemergt.
5. In regelmäßigen Abständen sollte jedes Teammitglied sich vom Remote den aktuellen Stand ziehen.

In Git Bash:

1. Ändere eine Datei (im Verzeichnisbaum)
2. Stelle selektiv eine Änderung in Staging Area um sie später zu comitten
3. Commit: Snapshot von Datei in Staging Area wird in GIT-Verzeichnis aufgenommen

Das heißt: wenn eine Datei verändert wurde, ist sie erst modified.

1. git pull //Nur nötig um Dateien vom Remote zu ziehen
2. git add . //bzw ein anderer add Befehl
3. git status //zeigt: neue/geaänderte Dateien, offene Commits
4. git commit -m „sinnvolle Notiz“ //staged
5. git push //pushed die Dateien auf das GITHUB-Repo
6. git log //zeigt commit-history

Sollte Git in einer Umgebung verwendet werden, die GIT bereits integriert, findet man die einzelnen Befehle idR als Symbole recht schnell über das git-Symbol.

## Arbeiten mit VS-Code

Visual Studio Code ist ein von Microsoft bereitgestellter Editor, welcher das arbeiten in verschiedensten Programmiersprachen ermöglicht. Anwendung findet es insbesondere um einen kurzen Überblick über Code zu bekommen oder schnelle Änderungen vorzunehmen.

Über GIT ist es dann noch möglich an einem Projekt über Pycharm oder Intellij etc. und VSCode zu arbeiten.

1. Falls man sich bei VSCode anmelden muss, am besten mit den gleichen Daten wie GITHUB.
2. In VSCode einfach links auf das GIT-Symbol klicken und der Anleitung folgen.

## Jupyter -Notebooks und R in VS-Code

Wie gesagt bietet VS-Code verschiedenste Möglichkeiten. So kann man auch Jupyter-Notebooks oder auch in R coden, wenn man die Plugins dafür installiert.

Um jetzt ein Jupyter-Notebook in VS-Code zu öffnen, brauchen wir zunächst einmal R oder R-Studio. (<https://cran.r-project.org/>)

Außerdem Python, am besten die neueste Version. (<https://www.python.org/downloads/windows/>)

In VS-Code:

1. R installieren
2. Python installieren

In R:

1. install.packages("IRkernel")
2. IRkernel::installspec()

In VS-Code:

1. pip install jupyter
2. pip install notebook

# CML mit GIT

CML verfügt bereits über integrierte Git-Funktionalitäten, erkennbar an der linken Seitenleiste (Sidebar).  
Die einfachste Möglichkeit, den eigenen Fortschritt in CML zu versionieren, ist das Anlegen eines Remote-Repositories, z. B. bei **Bitbucket** oder **GitHub**.

**Vorgehen in Bitbucket:**

1. Ein neues Repository in Bitbucket anlegen
2. Die Repository-URL kopieren
3. Optional: Über das Terminal prüfen, ob Name und E-Mail korrekt hinterlegt sind:
   * git config --user.name
   * git config --user.email

**In CML:** siehe auch [Commit a change to GitHub form JupyterLab GUI](https://www.youtube.com/watch?v=-V9GdXNXalQ) (1.21 min)

1. In das Verzeichnis wechseln, das versioniert werden soll
2. In der linken Sidebar auf das Git-Symbol klicken und **„Clone Repository“** auswählen
3. Die zuvor kopierte URL einfügen, Benutzername und Passwort (bzw. Token) eingeben

Daraufhin wird auf der gewählten Ebene ein neuer Ordner mit dem Namen des Repositories erstellt. Dieser Ordner wird nun von Git überwacht (getrackt).

Innerhalb dieses Ordners kann man über das Git-Symbol in der Sidebar alle Änderungen einsehen. Dort erscheinen alle Dateien als **staged**, **changed** oder **untracked**. Eine **History-Ansicht** aller bisherigen Commits ist ebenfalls verfügbar. Nicht commitete Dateien werden durch Warnhinweise hervorgehoben.

**Commit:**  
Im Feld **„Summary“** sollte eine kurze Beschreibung der vorgenommenen Änderungen eingetragen werden (Pflicht für den Commit, da Git sonst einen Fehler ausgibt).  
Optional kann im Feld **„Description“** genauer erläutert werden, warum bestimmte Änderungen vorgenommen wurden – besonders hilfreich bei komplexeren Projekten oder für eine saubere Dokumentation.

**Push & Pull:**  
In der oberen Menüleiste von CML finden sich unter dem Menüpunkt **Git** die Optionen **Push** und **Pull** sowie weitere Git-Funktionen.

~~Branch & Merge~~

Bitbucket erweiterung

~~Video von Christoph zu CML-Git~~

Screenshots

~~Gitdoku link~~

Arbeitsablauf im Team mit Git (Was wann wie machen)

In CML code lesen verstehen und Python klasen ansehen

Projekt test > test\_christoph>test>config>install>A-SASPwd.ipynb und ASASPwd.py

# Quellen und Hilfestellungen:

Git-Doku: <https://git-scm.com/book/en/v2>

Git-CML-Kurzanleitung: [Commit a change to GitHub form JupyterLab GUI](https://www.youtube.com/watch?v=-V9GdXNXalQ) (1.21 min)

Git-VSCode Kurübersicht: <https://www.youtube.com/watch?v=i_23KUAEtUM>