Git Project Workflow

Git einrichten

- installieren git version

Vorbereitung

Identität erstellen

check ob Namen und Email in Git config:

```
git config --list
```

Namen erstellen:

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

Email zu config hinzufügen

```
git config --global user.email "<your email address>"
```

Repository initialisieren

```
git init
```

wenn ihr nur diesen Befehl eingebt und nicht vorher Git gesagt habt, dass der Hauptbranch `main` heißen soll wird der Hauptbranch `master`sein.

um das zu ändern müsst ihr NACH des init Befehls folgenden Befehl ausführen:

```
git branch -m master main
```

um direkt bei der Initialisierung den Hauptbranch main zu nennen macht folgendes:

```
git init -b main
```

Um den Status unseres Neu-erstellten Repositories anzuschauen:

```
git status
```

Datei zu Staging hinzufügen

erstelle Datei test1.txt und füge sie der Staging-Area hinzu:

```
git add test1.txt
```

um die Datei wieder aus Staging herauszuholen

```
git rm --cached test1.txt
```

Füge test1.txt erneut der Staging-Area hinzu:

```
git add test1.txt
```

der erste Commit

wir commiten jetzt test1.txt

```
git commit -m "<your commit message>"
```

der zweite Commit

wir erstellen eine zweite test2.txt und commiten die mit den selben Schritten wie test1.txt

Commits anschauen

alle Commits bislang aufzeigen

```
git log
```

hier gibt es nun optionen, welche ihr dem log befehl hinzufügen könnt: - git log –oneline -> alles in einer Zeile - git log –pretty=oneline

Änderungen am Code

wir fügen zu test1.txt nun einen Text / Inhalt hinzu

```
git status
```

zeigt nun, dass test1.txt geändert/modified wurde. wir können nun ganz normal mit `git add text1.txt` und `git commit...` anschließend commiten.

um die Änderungen einer Datei mit dem letzten Stand der Datei vom letzten Commit abzugleichen

```
git diff
```

die Ausgabe kann folgendermaßen aussehen `@@ -0,0 +1 @@` = gelöschte Zeilen, geänderte Zeilen, hinzugefügte Zeilen

Branching

um neue Features zu implementieren oder zu testen generieren wir einen neuen ‘Branch’ der erst einmal unabhängig vom ‘main’ Branch ist

```
git branch <nameOfNewBranch>
```

um zu dem neuen Branch zu wechseln muss ich folgendes machen:

```
git checkout <nameOfBranch>
```

mit `git branch` zeigt mit der Output der Konsole alle existierenden Branches in meinem Repository und markiert den aktuellen Branch mit ’*’

Neuen Branch kreieren und direkt dahinwechseln

```
git checkout -b <nameOfBranch>
```

optionale Methode um den Branch zu welchseln (angebl. Idiotensicherer)

```
git switch <BranchName>
# wechseln und gleichzeitig einen neuen Branch generieren
git switch -c <NewBranchName>
```

Branch löschen

um einen Branch löschen wechsele erst in einen anderen Branch mit `git checkout` und gib folgenden Befehl ein

```
git branch -d <branchName>
```

Erster Merge

```
git merge <BranchYouWantToMergeIn>
```

Have a look at the Past

Du kannst mit Git dir den Zustand Deines Projets zu jedem x-beliebigen Commit-Zeitpunkt nochmals anschauen

```
git checkout <hashOfThePastCommit>
```

um wieder zurück in die Gegenwart zu kommen `git checkout main`

Projekt zurücksetzen

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

```
git reset --hard <hashOfPastCommit>
```

Create Pull Request (wenn Du im Team arbeitest)

Ich kreiere lokal ein neues Feature für das Projekt. - Dazu erstellen wir erste einen neuen Branch auf welchem wir das Feature entwickeln. - wir pushen dann das fertige Feature (also wenn commited) zu GitHub - unsere Kollegen können daraufhin dieses neue Feature testen - Wenn Feature okay wird es auf GitHub in den Main-Branch übernommen - Anschließen ziehe ich mir aus GitHub den akutellen Main-Branch auf meinen lokalen Main-Branch damit diese wieder den selben Stand haben

Lokales Repository

1. erstellen eines neuen Branches

   ```
   git checkout -b feature2
   ```

2. wir kreieren ‘main.py’ und schreiben eine kurze For-Schleife

3. Füge die neue Datei zu Staging

   ```
   git add main.py
   ```

4. committen main.py zum lokalen feature2 Branch repository

   ```
   git commit -m "main.py erstellt, schließlich machen wir
           Python"
   ```

5. pushe die Änderungen im ‘feature2’ Branch nun zu GitHub und erstelle auf GitHub automatisch ebenfalls einen ‘feature2’ Branch welcher gleichzeitig auf GitHub einen ‘Pull-Request’ auslöst.

   - Pull-Request ermöglicht es unseren Kollegen meine Änderungen zu überprüfen, testen und zu kommentieren.

   ```
   git push origin feature2
   ```

Auf GitHub

1. auf Github seht Ihr nun einen Button ‘Compare & pull request’
2. Ansicht um die Änderungen sich anzuschauen, wenn okay -> klicke auf ‘Create pull request’

3. Jetzt bekommst Du die Möglichkeit Deine Änderungen auf GitHub mit dem dortigen Main-Branch zusammenzuführen. (in real live, wenn Du in einer Organisation arbeitest, wird das wohl von Deinem Teamleiter gemacht und nicht von Dir)

Lokales Repositoy

1. das Projekt auf GitHub im Main-Branch ist jetzt aktueller als unser Projekt im Main-Branch. Wir bringen jetzt unseren Main-Branch auf den aktuellen GibHub Main-Branch Stand.
2. checken ob wir lokal uns aktuell auf dem Main-Branch befinden, wenn nicht auf Main-Branch wechseln
3. Im Main-Branch befindend holen wir uns nun die aktuelle Version aus GitHub `bash git pull origin main`
4. (optional) aufräumen: lösche unnötige branches `bash git branch -d feature2`

Neues Feature (best way, wenn Du nur für Dich alleine arbeitest - cc Sebastian)

Lokal

1. generiere einen neuen Branch in dem Du Deine Änderungen erstellen und testen möchtest. `git switch -c zorro`

2. wir fügen in der ‘main.py’ ein paar Code-Zeilen hinzu.

3. wir adden und commiten die geänderte ‘main.py’ im ‘zorro’ Branch

   ```
   git commit -am "Neue Funktion add zu main.py hinzugefügt"
   ```

4. ich teste meine neuen Funktionen jetzt auf Herz und Nieren.

5. wenn ich zufrieden bin und alles funktioniert, möchte ich diese neue Funktion nun in meinen ‘Main’-Branch übernehmen bzw. zusammenführen

   ```
   # dazu wechsele ich erst in den 'main' branch
   git checkout main

   # Nun merge ich die Änderungen aus dem 'zorro' Branch mit
           meinem 'main' Branch
   git merge zorro
   ```

6. ich möchte meine aktuelle Version meines Projektes nun auch zu meinem persönlichen GitHub Repo pushen

   ```
   git push origin main
   ```

Existierendes GitHub Repo klonen mit HTTPS

1. Gehe zum GitHup Repo welches Du klonen bzw. kopieren möchtest
2. Klicke auf Code und wähle ‘HTTPS’ und kopiere die dortige URL github-repo
3. auf Deinem lokalen Computer gehe zum Verzeichnis, in welches das neue Repo als Ordner gespeichert werden soll.

   ```
   cd /path/to/dirctory
   ```

4. füge die kopierte URL nun in folgenden Befehl ein, um das GitHub Repository auf Deinen Computer zu klonen

   ```
   git clone <URL/to/GitHub/Repository>
   ```

5. wechsle nun in das Verzeichnis des geklonten GitHub Repositories

   ```
   cd <Repository-Name>
   ```

neues Lokales Repo mit existierendem GitHub Repo verknüpfen

- wechsele in das Verzeichnis in welchem Du Dein neues Repo anlegen möchtest
- Repo initialisieren

  ```
  git init
  ```

Exkurs: aus modernen Gründen wird heute nicht mehr ‘master’ als Hauptbranch gerne verwendet sondern in zwischen ‘main’ Beim Anlegen eines neuen Repositories wird aber meist immer noch ‘master’ verwendet. Es gibt drei Möglichkeiten dies zu

1. global in der config:

```
git config --global init.defaultBranch main
```

1. während der initialisierung des Repositories

```
git init -b main
```

1. Master in main umbennen

```
git branch -m main
```

Exkurs ENDE

- lokales repo mit online Repo verbinden (Github)
  - auf GitHub im Repo die ‘HTTPS’ Adresse kopieren
  - lokal folgenden Befehl eingeben
  - `git remote add origin <URL>`
- Inhalte aus GitHub auf lokales Repo ziehen.
  - `git pull (origin main)`
  - evtl. werdet ihr nach Eurem GitHub-Usernamen und dem Token-Passwort gefragt
  - solltet ihr nach dem ‘pull’ keine Inhalte sehen müsst ihr nochmal den Branch wechseln

    ```
    git branch main
    ```

Git Merge Konflikte

1. Vorbereitung
   1. Erstelle neues Git-Repository oder wechsel in ein vorhandes
   2. Erselle eine neue Datei und füge Inhalte hinzu
   3. Stage und commite die Datei

      ```
      git add new_file.txt
      git commit -m "neue Datei erstellt"
      ```

   4. Erstelle neuen ‘feature’ Branch und wechsel in diesen

      ```
      git checkout -b feature
      ```

2. Änderung in beiden Branches vornehmen
   1. im ‘feature’-Branch ändere new_file.txt
   2. Stage und commite die Änderungen.

      ```
      # im feature Branch
      git add new_file.txt
      git commit -m "changes in feature branch"
      ```

   3. Wechsle nun zurück zum ‘main’-Branch

      ```
      git switch main
      ```

   4. Im ‘main’-Branch mache nun ebenfalls Änderungen (aber andere) in der Datei

5. Stage und commite die Änderungen dort ebenfalls

```
# im 'main'-Branch
git add new_file.txt
git commit -m "changes in main branch"
```

3. Merge Versuch
    1. **HINWEIS**: Es wird immer von dem Branch aus der Merge initialiert, in welchen ein anderer Branch integriert werden soll. Meistens will man in ‘main’ integrieren, also wird auch meistens von diesem Branch aus der Merge initialisiert
    2. Merge feature Branch in main Branch

```
# im main branch
git merge feature
```

    1. Git weist Dich nun darauf hin, dass es den Merge nicht ausführen kann, da es zu einem Konflikt in der ‘new_file.txt’ gekommen ist. Heißt. Git hat zwei Unterschiedliche Versionen von dieser Datei und weiß nicht wie es diese behandeln soll

```
Auto-merging new_file.txt
CONFLICT (content): Merge conflict in new_file.txt
```

4. Konflikt auflösen
    1. Git wird Dir nun beide Versionen der new_file.txt in der Datei anzeigen. Wenn Du die Datei in einem Editor nun öffnest wird sie folgendermaßen ausschauen

```
Zeile 1
<<<<<<<<<< HEAD
Änderung auf main-Branch
==========
Änderung auf feature-Branch
>>>>>>>>> feature
```

    1. Du musst die Datei manuell bearbeiten, um zu entscheiden, welche Änderungen beibehalten werden sollen. Du kannst entweder eine der beiden Versionen übernehmen oder die Änderungen zusammenführen.

```
Zeile 1
Änderung auf main-Branch
Änderung auf feature-Branch
```

5. Konflikt als gelöst markieren
    1. Wenn Du die Datei(en) manuell nun geändert hast musst Du diese als gelöst für Git markieren.

2. Dies machst Du in dem Du die Datei(en) nochmal Staged und commitest

```
# immer noch im main branch
git add new_file.txt
git commit -m "Conflict solved successfully and finished
        merge"
```

6. Schließlich kannst Du schauen ob der Merge erfolgreich war und den Verlauf Deiner Commits überprüfen

```
git log --oneline --graph
```