Versionsverwaltung & Git

Julian Huber & Matthias Panny

Versionsverwaltung & Git

🎯 Lernziele

- Studierende wissen was Versionsverwaltung ist & warum sie wichtig/hilfreich ist
- Studierende können mit Git über die Kommandozeile und über VS Code arbeiten
- Studierenden können Repositories auf github anlegen und verwalten

Git

- Git [gɪt] ist eine freie Software zur verteilten Versionsverwaltung von Dateien, die durch Linus Torvalds initiiert wurde
- Neben der Versionierung ist inzwischen ein Ökosystem zur Entwicklung und der Pflege von Software gewachsen

Linus Torvalds.
Initiator des Linux-Kernels und von Git.

Begriffe

- GitHub ist eine Website (von Microsoft) auf der man Code speichern und veröffentlichen kann. Dient als Backup, Visitenkarte uvm. Es gibt diverse andere Anbieter
- Git ist die dahinterliegende Technologie zur Verwaltung von Quellcode
- Ein Repository oder Repo ist das Projektverzeichnis, in dem meist mehrere Dateien liegen.

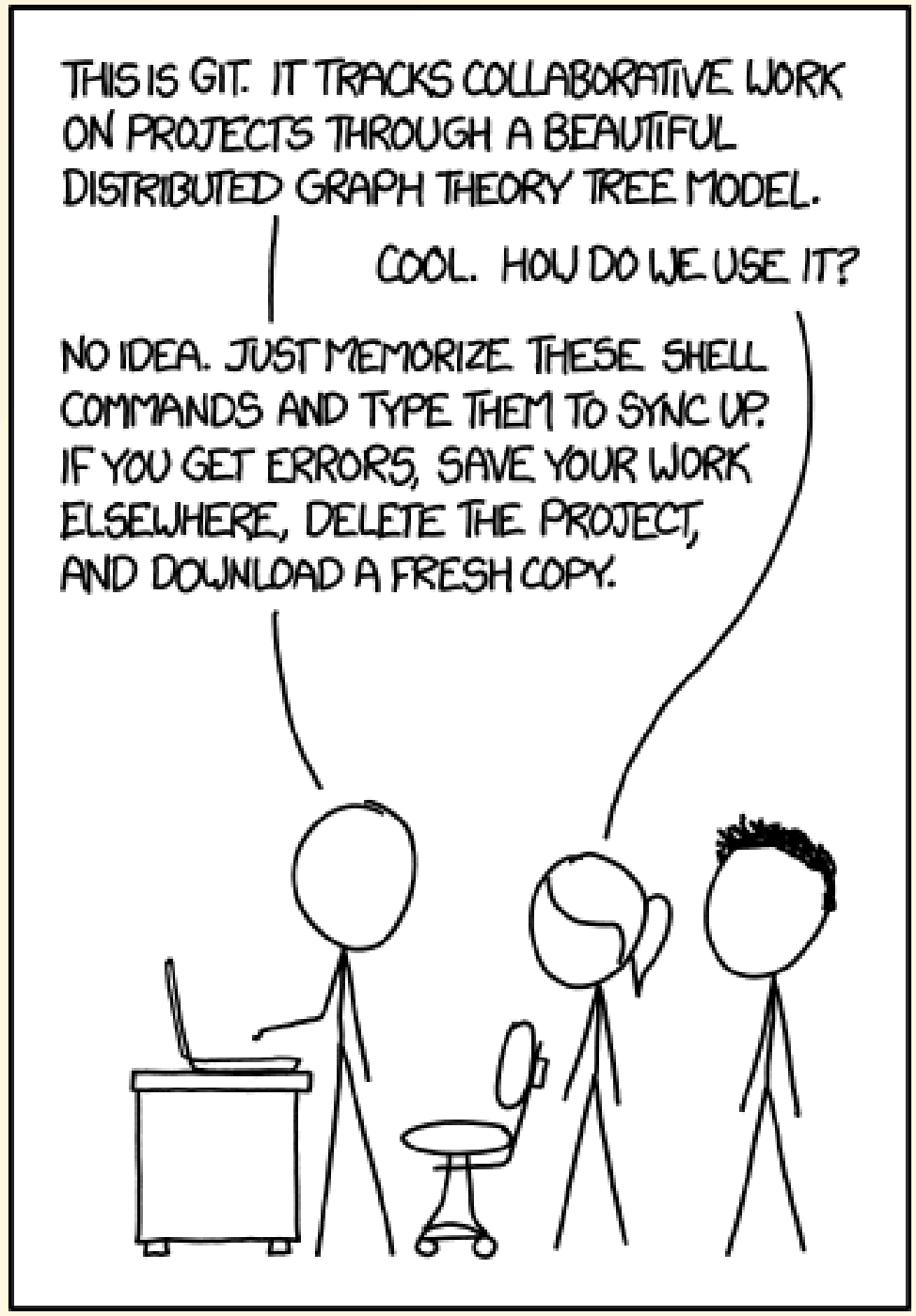


Bild: https://xkcd.com/1597/

Versionskontrolle beim alleinigen Arbeiten

- Mit Versionskontrollsoftware können Änderungen mit Notizen versehen werden → entkoppelt Dateinamen und Version
- Es kann jederzeit zu einer Vorversionen zurückgekehrt werden

Beispiel ohne Versionsverwaltung

- Version wird in Dateinamen festgehalten
  - `main.py`
  - `main2.py`
  - `main2_test.py`

Beispiel mit Versionsverwaltung

- Version wird in *commits* mit einer eindeutigen ID festgehalten
  - `main.py` (8e3ae40)
  - `main.py` (8c22176)
  - `main.py` (7e01f48)

Git - Commits & Commit-Messages

- Jede Version wird in einem commit festgehalten
- Zu jedem Commit gehört eine Nachricht, die die Änderung zur letzten Version beschreibt

Git - Projektstatus

- Durch Git bleibt der Projekt-Status konsistent → auch wenn mehrere Dateien von der Änderung betroffen sind

Beispiel

- Abschlussarbeit als $LaTeX$-Dokument
  - Version 1: `main.tex` und `box_plot.svg`
  - Version 2: `main.tex` und `scatter_plot.svg`

```
#Version 1
.
├── images
│   └── box_plot.svg
└── main.tex --> \includesvg{images/box_plot.svg}

#Version 2
.
├── images
│   └── scatter_plot.svg
└── main.tex --> \includesvg{images/scatter_plot.svg}
```

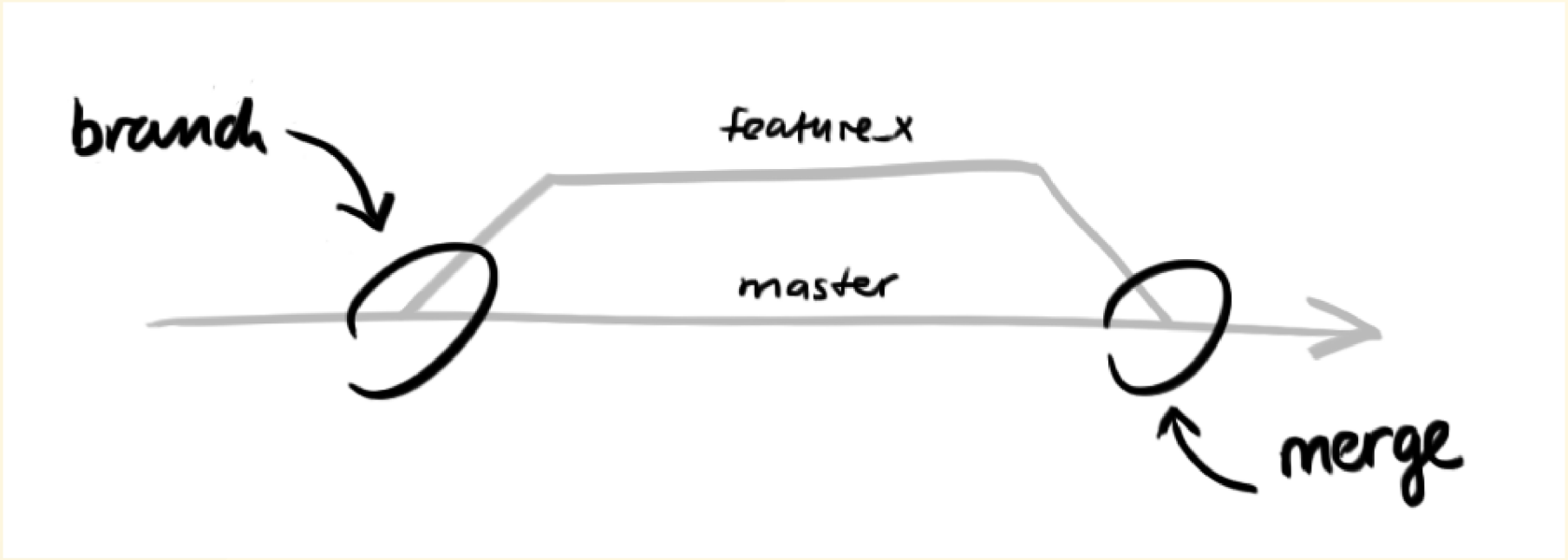
- Grafik & Inhalt von `main.tex` haben sich gemeinsam & konstisten geändert

Git - Projektstatus

- Mehrere Alternativen könne gleichzeitig verfolgt werden → wenn z.B. ein neues Feature getestet werden muss → Verzweigungen bzw. Branches im Projekt
- Ansatz: *"move fast and break things"* → funktioniert nur, weil man immer ein funktionsfähiges Backup hat

Git - Branches

- Verzweigungen eines Projekts
- An einem bestimmten Zeitpunkt zweigt das Projekt ab → hier mit dem Namen `feature_x`
- Ist das Feature fertig entwickelt und erfolgreich getestet, wird es wieder mit dem originalen Projektstamm `master` od. `main` gemerged



Git - Branches

- Es gibt immer eine stabile (funktionierende) Version → master oder `main`-Branch
- Neue Idee, Änderungen, Fehlerbehebungen werden in separaten Branches entwickelt & getestet (z.B.: `feature_x`)
- Sobald die Änderungen fertig sind, werden sie in den main-Branch gemerged - → hier gibt es eine Differenz zw.:
  - unveränderten Dateien im `main` Branch
  - veränderten Dateien im `feature_x` Branch

  → müssen durch eine bewusste Entscheidung beseitigt werden

Zu merken:

- Der Prozess:
  1. neuen Branch erstellen
  2. Änderungen durchführen & testen
  3. Änderungen in den `main`-Branch mergen

→ soll für jedes Feature durchgeführt werden!

github - Plattform für Git

- github.com ist eine zu Microsoft gehörende Online-Plattform zur Versionsverwaltung von Softwareprojekten mittels Git
- Stellt eine zentrale Plattform dar → Features für Projektmanagement, CI/CD, Wiki, Copilot, etc.

Anlegen eines github-Accounts

- Wir werden in weiterer Folge einen github-Account benötigen
- mit der "@mci4me.at"-Adresse können Sie sich ein kostenlosen GitHubPro Student-Account anlegen
- → muss evtl. noch durch Dokument (Studierendenausweis, etc.) bestätigt werden

Architektur von Git & Beispiel

Wir wollen uns die Funktionsweise von Git ansehen und diese anhand eines Beispiels mit einem lokalen Repository erproben

Git - Beispiel

- Wir wollen nun in einem Beispiel den gesamten üblichen Git-Workflow durchlaufen:
  - Anlegen eines neuen Projekts
  - Erstellen eines Git-Repository
  - Erstellen & hinzufügen einer Datei
  - Commit der Änderungen
  - Anderen Branch erstellen
- Dazu sehen wir uns Schritt für Schritt die Architektur von Git an

Begriffe

- *Repositories* sind im Allgemein Projektordner incl. `.git` Dateien
  → alles darin wird von Git versioniert
- *Commit* ist ein Snapshot/eine Version des Projekts zu einem bestimmten Zeitpunkt
- *Branch* ist eine Verzweigung des Projekts, um z.B. neue Features zu entwickeln

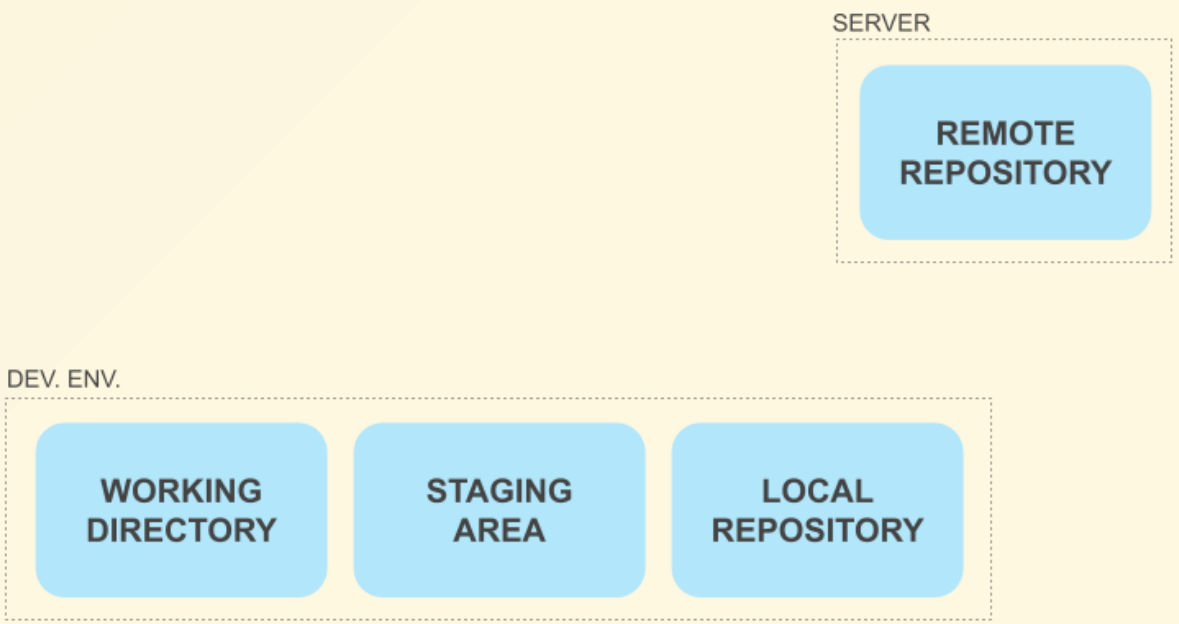
Architektur von Git

Entwicklungsumgebung → auf dem eigenen PC

- *Working directory*: auf dem PC an dem am Code gearbeitet wird
- *Staging area*: "Vorschau" für commit → `git add`
- *Local repository*: Repository in dem Git die Änderungen verwaltet → `git commit`

Server → üblicherweise github

- *Remote repository*: Teilt die Änderungen mit der Umwelt → `git push`



Quelle aller folgenden Bilder: Quelle

Beispiel in der Git Bash

- Anlegen eines neue Projekts

```
$ mkdir project_1
$ cd project_1/
```

- Erstellen eines neuen Git-Repository

```
$ git init
Initialized empty Git repository in /project_1/.git/
```

- Erstellen einer Datei, Hinzufügen zum Repository und Commit

```
$ nano main.py
$ git add .
$ git commit -m "created first file"
```

- Sollte beim ersten commit noch kein Name und Email-Adresse konfiguriert sein, wird dies in der Shell mit einer Fehlermeldung angezeigt
- Für die globale git-Installation geschieht dies mit

```
$ git config --global user.name "Max Mustermann"
$ git config --global user.email "max@mustermann.at"
```

→ verwenden Sie hier die gleichen Einstellungen wie für github

Beispiel in der Git Bash

- Das lokale Repository ist nun initialisiert → alle von Git erstellten Dateien sind im versteckten Verzeichnis `.git` abgelegt → wird dieses gelöscht ist das Repository nicht mehr verwendbar
- Die Datei `main.py` ist im Repository
- Die *Staging Area* ist leer und es gibt keine Änderungen

Beispiel

- Anzeigen des verstecken `.git` Verzeichnis

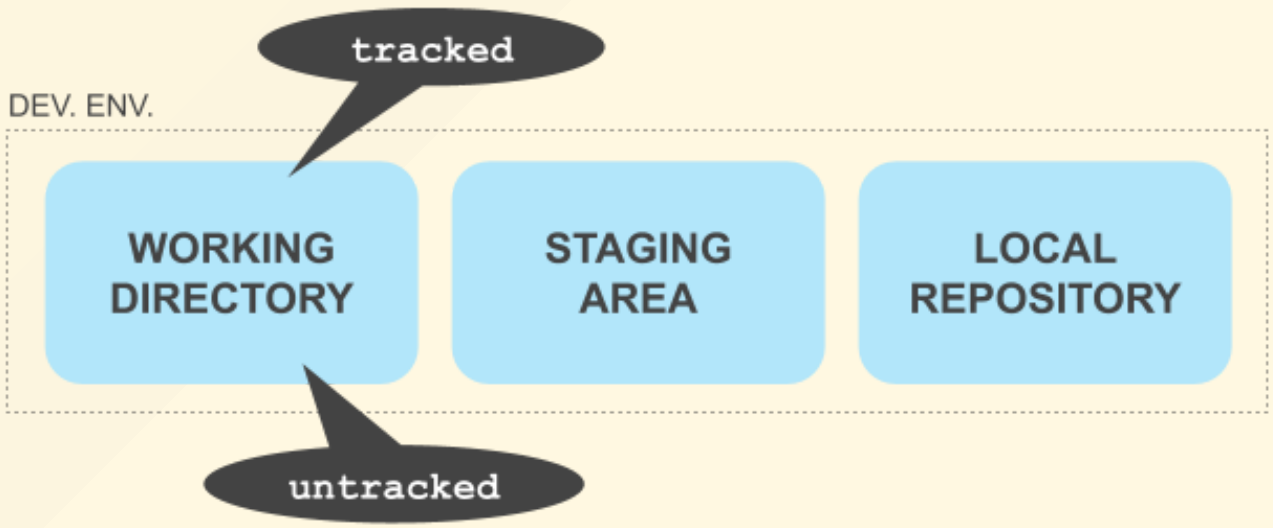
```
$ ls -la
total 9
drwxr-xr-x 1 jlhuber 1049089  0 Oct 10 09:15 ./
drwxr-xr-x 1 jlhuber 1049089  0 Oct 10 09:14 ../
drwxr-xr-x 1 jlhuber 1049089  0 Oct 10 09:16 .git/
-rw-r--r-- 1 jlhuber 1049089 15 Oct 10 09:15 main.py
```

- Überprüfen des Status auf Änderungen

```
$ git status
On branch master
nothing to commit, working tree clean
```

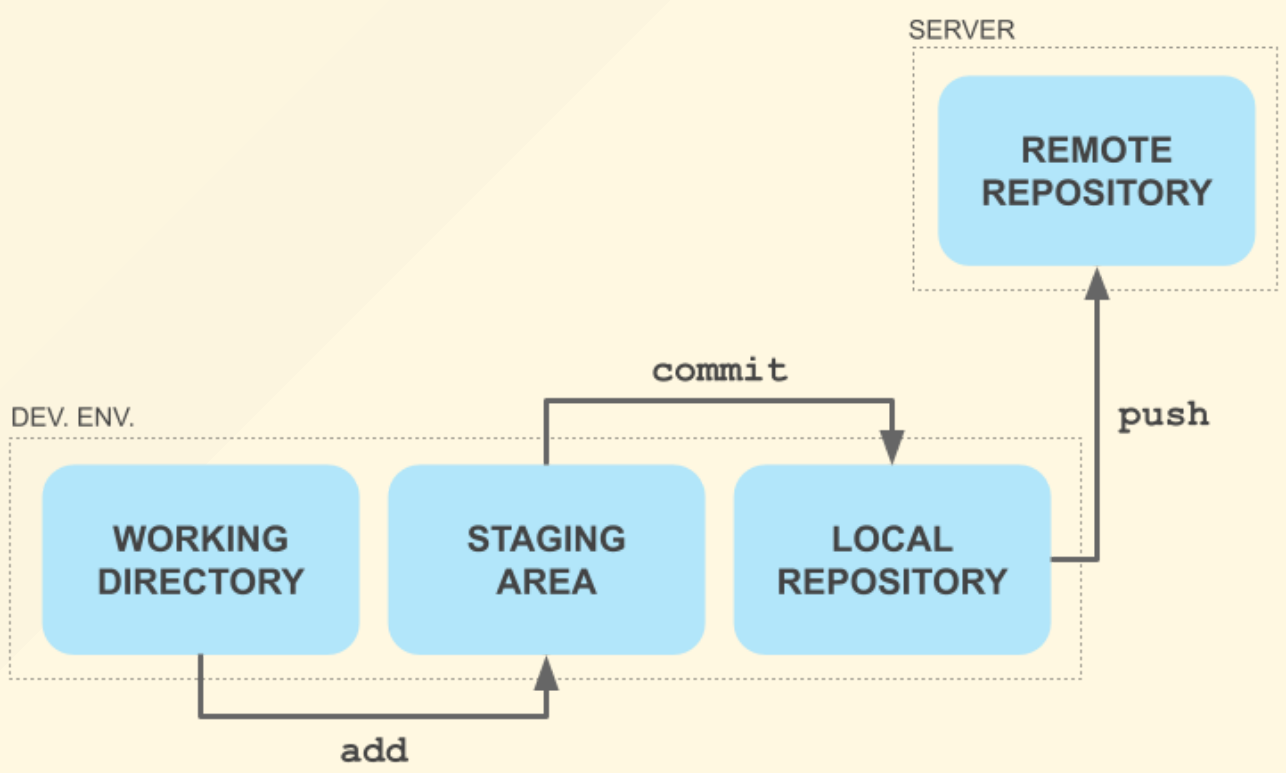
Architektur von Git - Änderungen im Working Directory

- Was passiert wenn sich Dateien im *Working Directory* ändern?
- → passiert durch normales speichern der Datei in z.B. VSCode oder nano
- Es gibt zwei Arten von Dateien im Working Directory
  - *Tracked*: Dateien, die git bereits kennt (nach `git add`)
  - *Untracked*: Dateien, die git (noch) nicht kennt



Architektur von Git - Updaten des Local Repository

- Wir wollen Änderungen von *Working Directory* in *Local Repository* übernehmen → egal ob ob Dateien *Tracked* oder *Untracked* sind
- Git erkennt automatisch, welche Dateien seit dem letzten Commit geändert wurden → werden der *Staging Area* mit `git add` hinzugefügt
- *Commit* fasst zusammen-gehörende Änderungen zusammen



Beispiel in der Git Bash

- Anpassen der Datei & Status überprüfen

```
$ nano main.py
$ git status
On branch master
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:   main.py
```

- Änderungen zu *Staging Area* hinzufügen

```
$ git add .
warning: LF will be replaced by CRLF in main.py.
The file will have its original line endings in your working directory

$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   main.py
```

- Änderungen committen

```
$ git commit -m "made it print hello world"
[master d16ac29] made it print hello world
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Beispiel in der Git Bash

- Wir können uns einen Überblick über alle vergangenen Commits verschaffen
- Anzeigen der Commit-History

```
$ git log
commit d16ac298c8bebfbee72520da0a64b0fa2690845f (HEAD -> master)
Author: jhumci <julian.huber@mci.edu>
Date:   Tue Oct 10 09:20:55 2023 +0200

    made it print hello world

commit 22cde1b198d304a8eb428f8905ee8d8304316fbd
Author: jhumci <julian.huber@mci.edu>
Date:   Tue Oct 10 09:16:18 2023 +0200

    created first file
```

- Es werden Commit-Hash (ID), Autor, Datum und Commit-Message angezeigt

Beispiel in der Git Bash

- Wir wollen einen neuen Branch erstellen und dorthin wechseln
- In git bash wird in runden Klammern der aktuelle Branch angezeigt
- Erstellen und Wechsel zu neuem Branch

```
jlhuber@4LAP104500JLH MINGW64 /h/project_1 (master)
$ git branch add_variable

$ git checkout add_variable
Switched to branch 'add_variable'
```

- Anpassen der Datei

```
jlhuber@4LAP104500JLH MINGW64 /h/project_1 (add_variable)
$ nano main.py

$ git add .
warning: LF will be replaced by CRLF in main.py.
The file will have its original line endings in your working directory

$ git commit -m "added a variable to the print statement"
[add_variable bc6fc8f] added a variable to the print statement
 1 file changed, 2 insertions(+), 1 deletion(-)
```

Beispiel in der Git Bash

- Commit-History ändert sich mit dem Branch-Wechsel → es wird nur die Historie des aktuellen Branches angezeigt
- Anzeigen der Commit-History

```
$ git log
commit bc6fc8f85adf2716bfc738c20a46c1ccc665e29e (HEAD -> add_variable)
Author: jhumci <julian.huber@mci.edu>
Date:   Tue Oct 10 09:24:27 2023 +0200

    added a varibale to the print statement

commit d16ac298c8bebfbee72520da0a64b0fa2690845f (master)
Author: jhumci <julian.huber@mci.edu>
Date:   Tue Oct 10 09:20:55 2023 +0200

    made it print hello world

commit 22cde1b198d304a8eb428f8905ee8d8304316fbd
Author: jhumci <julian.huber@mci.edu>
Date:   Tue Oct 10 09:16:18 2023 +0200

    created first file
```

Beispiel in der Git Bash

- Wir wollen nun in den `master`-Branch wechseln und die Änderungen von `add_variable` in `master` mergen
- Wechsel zu Master-Branch

```
jlhuber@4LAP104500JLH MINGW64 /h/project_1 (add_variable)
$ git checkout master
Switched to branch 'master'
```

- Merge von `add_variable` in `master`

```
jlhuber@4LAP104500JLH MINGW64 /h/project_1 (master)
$ git merge add_variable
Updating d16ac29..bc6fc8f
Fast-forward
 main.py | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
$ nano main.py
```

- VS-Code unterstützt uns in Zukunft dabei, dennoch ist es hilfreich die wichtigsten Befehle zu verstehen
- Öffnen Sie das Verzeichnis nun in VS Code, wählen Sie die Datei `main.py` aus und zeigen Sie die Zeitachse an

Branching in Git

- Branches sind sehr mächtig, können aber auch zu Probelemen führen wenn sie nicht richtig gehandhabt werden
- Wir haben nur die beiden folgenden Kommandos kennengelernt:
  - `git branch <branch-name>`: Erstellt einen neuen Branch
  - `git checkout <branch-name>`: Wechselt zu einem Branch
- Um alle weitern Befehle zu erproben kann das Online-Tool Learn Git Branching verwendet werden

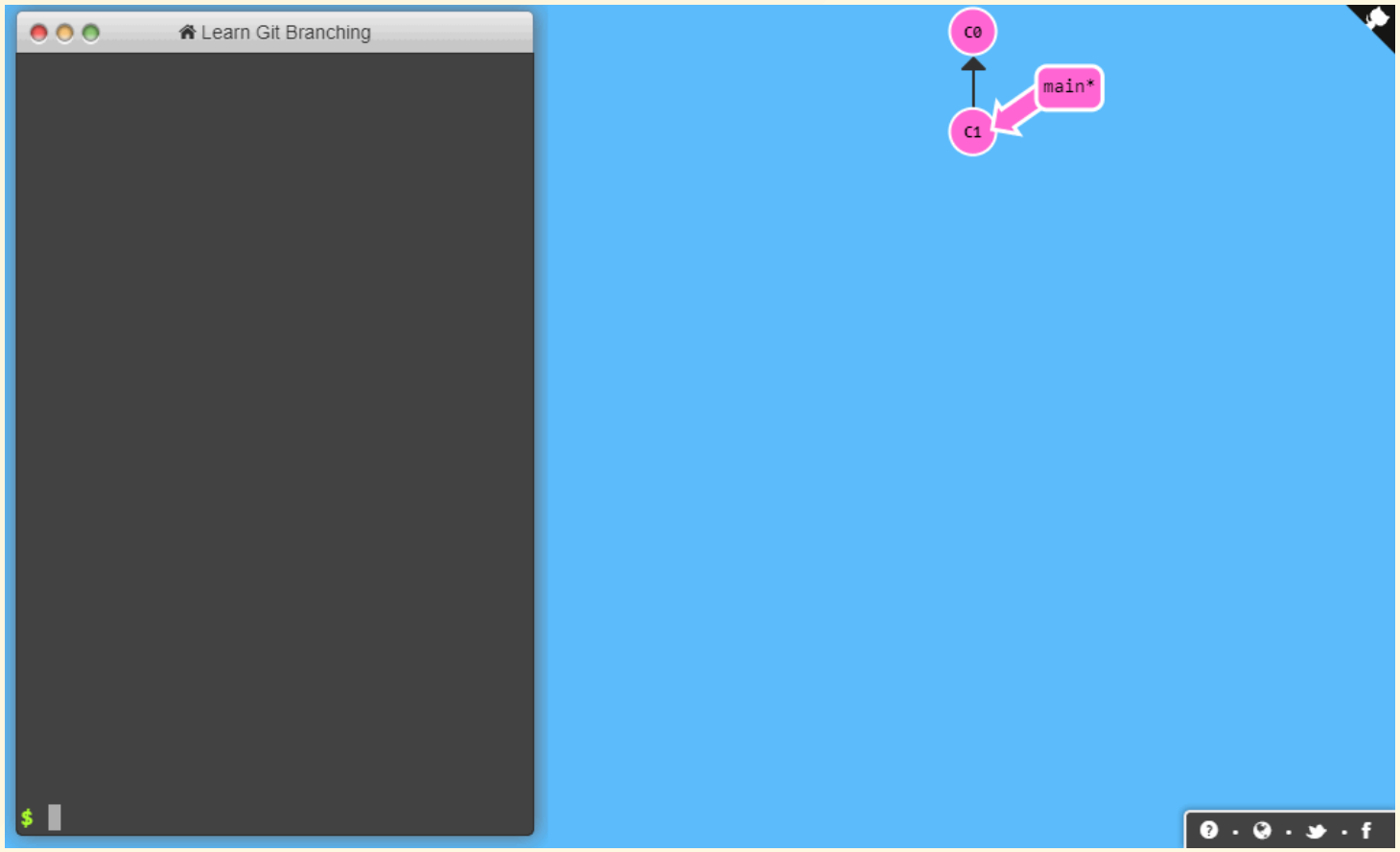


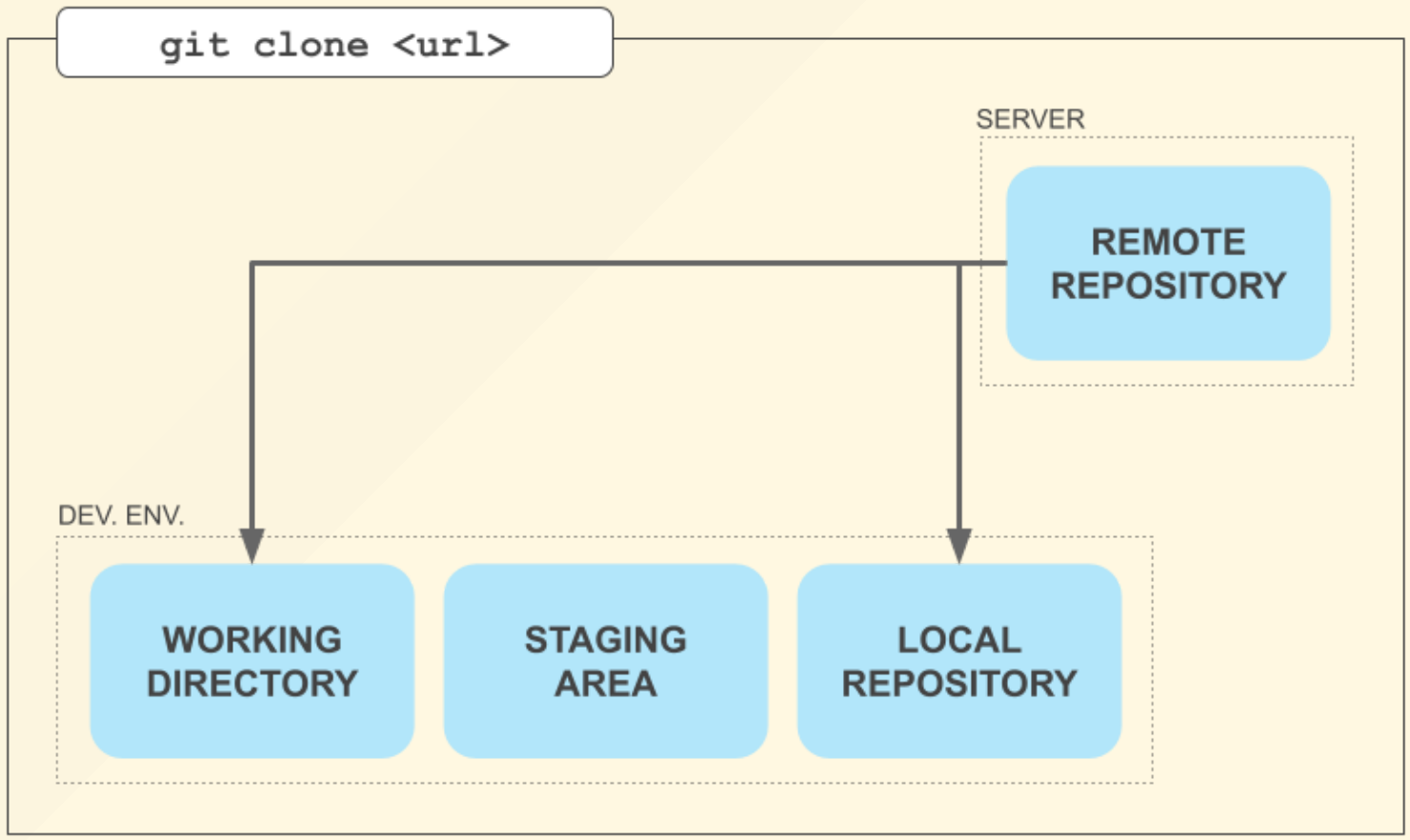
Bild: learngitbranching.js.org

Arbeiten mit remote respositories

Interaktion mit remote repository auf github

Architektur von Git - Klonen eines Repositories

- häufig ist Ausgangspunkt ein bestehendes Repository → soll kopiert werden um daran fortzusetzen → `git clone`
- das Remote Repository landet an zwei Orten
  - *Working directory*
  - *Local repository*



Beispiel in der Git Bash

- Wir wollen nun ein erstes Repository klonen:
  - Repository: `test_git_shell` → https://github.com/jhumci/test_git_shell
  - öffentliches Repository mit zwei Branches
- Klonen des Repositories

```
$ git clone https://github.com/jhumci/test_git_shell
Cloning into 'test_git_shell'...
remote: Enumerating objects: 68, done.
remote: Counting objects: 100% (68/68), done.
remote: Compressing objects: 100% (53/53), done.
remote: Total 68 (delta 30), reused 29 (delta 5), pack-reused 0 (from 0)
Receiving objects: 100% (68/68), 12.64 KiB | 1.40 MiB/s, done.
Resolving deltas: 100% (30/30), done.
```

- Inhalt des Repository überprüfen

```
$ cd test_git_shell/
$ ls -la
total 19
drwxr-xr-x 1 mtpanny 1049089    0 Nov  5 14:04 .
drwxr-xr-x 1 mtpanny 1049089    0 Nov  5 14:03 ..
drwxr-xr-x 1 mtpanny 1049089    0 Nov  5 14:04 .git
-rw-r--r-- 1 mtpanny 1049089   25 Nov  5 14:03 .gitignore
-rw-r--r-- 1 mtpanny 1049089 1088 Nov  5 14:04 LICENSE.md
drwxr-xr-x 1 mtpanny 1049089    0 Nov  5 14:03 data
-rw-r--r-- 1 mtpanny 1049089   67 Nov  5 14:03 hello_world.py
-rw-r--r-- 1 mtpanny 1049089  403 Nov  5 14:03 make_primes.py
-rw-r--r-- 1 mtpanny 1049089 2808 Nov  5 14:03 readme.md
```

Beispiel in der Git Bash

- Unabhängig von der Art des Softwareprojekts gibt es einige Dateien, die in jedem Projekt vorhanden sein sollten

Wichtige Dateien für Git

- `.gitignore`: Listet alle Dateien und Ordner auf, die nicht mittels git geteilt werden sollen → ⚠️ muss zu Beginn des Projekts erstellt werden!
- 🤓 `.gitattributes`: Konfigurationsdatei für git

Allgemeine wichtige Dateien

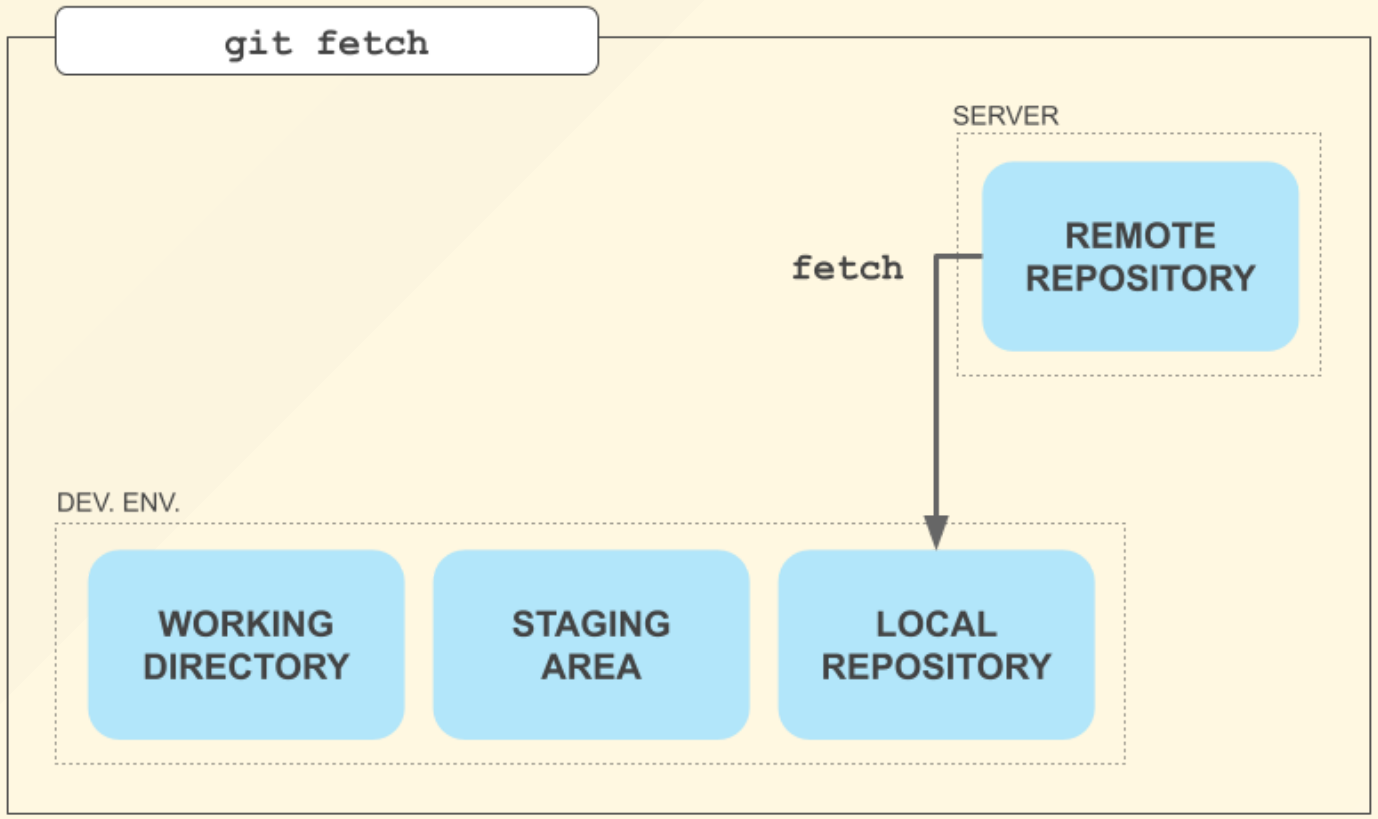
- `readme.md`: Textdatei mit Beschreibung & Erklärung des Projekts
- `LICENSE` oder `LICENSE.md`: Lizenzdatei für das Projekt

Architektur von Git - gitingore

- `.gitignore`-Datei ist eine Textdatei, die Dateien und Ordner enthält, die von Git ignoriert werden sollen
- Üblicherweise wollen wir nur die Dateien und Ordner im Repository haben, die für das Projekt relevant sind und NICHT automatisch generiert werden
- Beispiele:
  - `.DS_Store`-Dateien auf Mac
  - `.vscode`-Ordner
  - `__pycache__`-Ordner
  - `.venv`-Ordner
  - `*.pyc`-Dateien
  - etc.
- Für Projekte in den meisten Programmiersprachen gibt es bereits vorgefertigte `.gitignore`-Dateien → github/gitignore → jeweilige Datei herunterladen, in das Projekt kopieren und in `.gitignore` umbenennen

Architektur von Git - Updaten des lokalen Repository

- Zum Zeitpunkt des Klonens ist das lokale Repository auf dem neusten Stand → wir werden aber nicht automatisch über Änderungen im *Remote Repository* informiert
- Bevor lokal weiter gearbeitet wird immer mit `git fetch` die aktuellen Änderungen abholen:
  - das *Local Repository* wird auf den neusten Stand gebracht
  - Änderungen im eigenen *Working Directory* bleiben unangetastet



Beispiel in der Git Bash

- Lokalen Stand des Repository überprüfen

```
$ git log
commit a0d6f07bd29595e9bbe346acf78322489851e423 (HEAD -> master)
Author: Julian Huber <94922106+jhumci@users.noreply.github.com>
Date:   Wed Feb 7 08:54:08 2024 +0100
    Update readme.md
commit c303672b3a9215a69b07b48a49833e53b872c017
Author: jhumci <julian.huber@mci.edu>
Date:   Wed Jul 12 10:12:20 2023 +0200
    new tasks
```

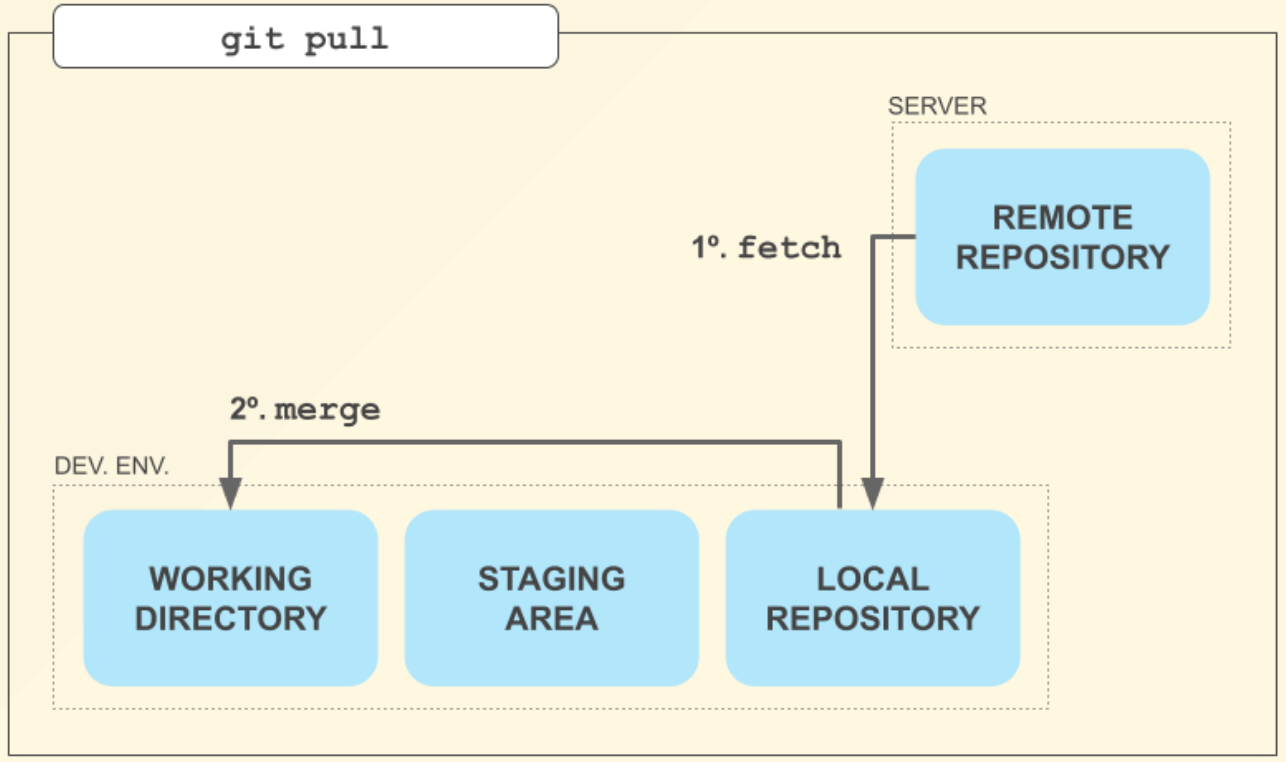
- Änderungen von Remote Repository holen → `git fetch`
- Änderungen zu aktuellem lokalen Stand überprüfen

```
$ git log HEAD..origin/master
commit 982a416434965dcf9a4956824c22a105782662b4 (origin/master, origin/HEAD)
Author: Julian Huber <94922106+jhumci@users.noreply.github.com>
Date:   Mon Apr 8 10:01:23 2024 +0200
    Update readme.md
commit 9602dce8ff2ba72b9900289525f336b57b85d497
Author: Julian Huber <94922106+jhumci@users.noreply.github.com>
Date:   Mon Apr 8 08:13:51 2024 +0200
    Update readme.md
commit c009fda83f2d19c6040bc70a5b527dbc1efa9963
Author: Julian Huber <94922106+jhumci@users.noreply.github.com>
Date:   Wed Feb 7 08:55:48 2024 +0100
    Update .gitignore
commit 29893946a2cbdfa7e80ac771af428805d2be91fc
Author: Julian Huber <94922106+jhumci@users.noreply.github.com>
Date:   Wed Feb 7 08:55:21 2024 +0100
    Update readme.md
```

Architektur von Git - Pulling

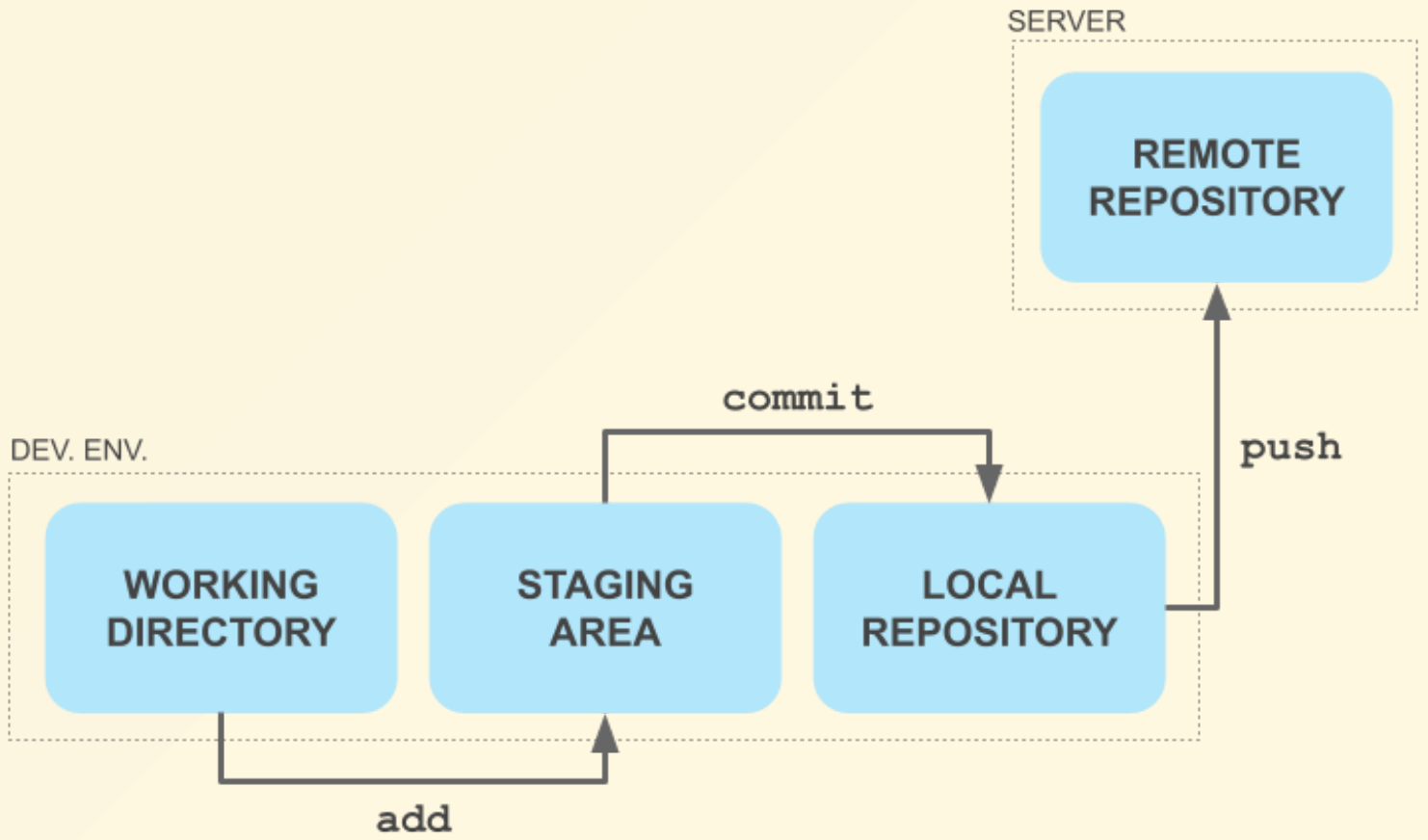
- Nachdem die Änderungen abgeholt wurden, können sie in das eigene *Working Directory* eingebaut werden → `git merge`
- Üblicherweise wollen wir beide Schritte kombinieren
  - `git fetch`: Holen der aktuellen Änderungen
  - `git merge`: Einbauen der Änderungen

  → `git fetch` + `git merge` → `git pull`



Architektur von Git - Updaten des Remote Repository

- 🔁 Um Änderungen aus dem *Working Directory* ins *Local Repository* zu übertragen → `git add` + `git commit`
- Änderungen im *Local Repository* können auch ins *Remote Repository* übertragen werden → `git push`



Git - Umgang mit Unterschieden

- Änderungen des *Working Directory* gegenüber dem letzten Commit können mit `git diff` angezeigt werden
- `+` In der Datei ist Code hinzugekommen
- `-` Code wurde aus der Datei entfernt

```
Techie Delight@ThinkPad MINGW64 ~/Desktop/api (master)
$ git diff HEAD~5..HEAD --stat
 CHANGELOG.md                                | 95 ++++++++++++++++++++++++++++++++++
 RELEASE_NOTES_TEMPLATE.md                   | 57 ++++++++++++++++++++
 app/controllers/submissions_controller.rb | 54 +++++++++++++++++++-
 app/helpers/config.rb                       |  4 +-
 config/routes.rb                            |  3 +-
 judge0-api.conf.default                     |  6 +-
 6 files changed, 214 insertions(+), 5 deletions(-)

Techie Delight@ThinkPad MINGW64 ~/Desktop/api (master)
$ git diff HEAD~5..HEAD --compact-summary
 CHANGELOG.md                                | 95 ++++++++++++++++++++++++++++++++++
 RELEASE_NOTES_TEMPLATE.md (new)             | 57 ++++++++++++++++++++
 app/controllers/submissions_controller.rb | 54 +++++++++++++++++++-
 app/helpers/config.rb                       |  4 +-
 config/routes.rb                            |  3 +-
 judge0-api.conf.default                     |  6 +-
 6 files changed, 214 insertions(+), 5 deletions(-)

Techie Delight@ThinkPad MINGW64 ~/Desktop/api (master)
$
```

✍️ Aufgabe

- Wir wollen nun mit dem Repository `test_git_shell`, welches wir bereits geklont haben, arbeiten
- Stellen Sie sich vor, Sie wollen Ihren Code auf einem Supercomputer laufen lassen, um alle Primzahlen zu finden. Der Supercomputer hat ein Linux-Betriebssystem und kann über Secure Shell (SSH) aus der Ferne angesprochen werden.
- Wir simulieren die SSH-Verbindung, indem wir stattdessen git bash verwenden
- Die genaue Anleitung wie das Programm zum finden der Primzahlen funktioniert, und was Sie damit tun sollen, ist in der `readme.md` des Repositorys zu finden

🏆 Hausaufgabe

Benchmarking Sortier-Algorithmen auf 2 PCs

Benchmarking Sortier-Algorithmen auf 2 PCs

- Erstellen Sie zu zweit ein gemeinsames öffentliches Git-Repository für das Projekt z.B. direkt auf github.com
- Ihnen wird ein Paket aus drei zip-Files zur Verfügung gestellt, die Sie zunächst in drei verschiedenen Branches einfügen:
- `inital.zip`: nach dem Erstellen des Respositories befinden Sie sich im `main`-Branch. Auf einem der Computer fügen Sie die Dateien aus `inital.zip` ein, comitten die Änderungen und pushen Sie diese auf github
- `sort_1.zip`: auf dem ersten Computer erstellen Sie einen neuen Branch `sort_1` und fügen die Datei `sort_1.py` ein, comitten die Änderungen und pushen Sie diese auf github
- `sort_2.zip`: auf dem zweiten Computer erstellen Sie einen neuen Branch `sort_2` und fügen die Datei `sort_2.py` ein, comitten die Änderungen und pushen Sie diese auf github

Benchmarking Sortier-Algorithmen auf 2 PCs

- mergen Sie die Änderungen von beiden Branches nacheinander in den `main`-Branch (Indem Sie auf der github-Website `Compare & pull request` auswählen) oder in VS Code mittels `"Branch/ Branch zusammenführen"`
- Pullen Sie die Änderungen nachdem Sie alles im `main`-Branch gesammelt haben auf die beiden Rechner und stellen Sie sicher, dass sie in den `main`-Branch wechseln
- Führen Sie das Skript `test.py` auf beiden Rechnern aus
- Stellen Sie die neu erstellen Dateien im Repository im `main`-Branch bereit (`commit` und `push`) und `mergen` Sie falls nötig die Änderungen in den `main`-Branch

🏆 Benchmarking Sortier-Algorithmen auf 2 PCs

- Geben Sie die Links zur `results.txt` und die History in einem öffentlichen github Repository ab. Die Dateien könnten z.B. wie folgt aussehen:
  - results.txt
  - Branch-Graph

```
Computer Name: rechner_1
Date of Test: 2023-10-10 17:20:03
Bubble Sort Time: 13.187571 seconds
Quick Sort Time: 0.019813 seconds

Computer Name: rechner_2
Date of Test: 2023-10-10 17:20:03
Bubble Sort Time: 11.245562 seconds
Quick Sort Time: 0.022345 seconds
```