www.freifunk-ag.org

`git`

Was ihr an der Uni noch nicht über die Versionsverwaltung gelernt habt. Ein praxisorientierter Workshop in zwei Teilen.

Martin Hübner, Freifunk-AG

Stand: 24. Februar 2023

www.freifunk-ag.org

1. Motivation
   - Stell' dir mal vor...
   - Wie kann uns `git` helfen?
2. Workshop-Time
   - Spielregeln
   - My first `git`-Experience
   - Wie speichert man eine Änderung?
   - Branches nutzen
   - Branches zusammenführen: `git merge`
   - Branches zusammenführen II: `git rebase`
3. Fortgeschrittene Workflows und Best-Practices
   - Meta-Gedanken zu `git`
   - Some advanced commands
   - Advanced Workflows

www.freifunk-ag.org

Teil Eins.

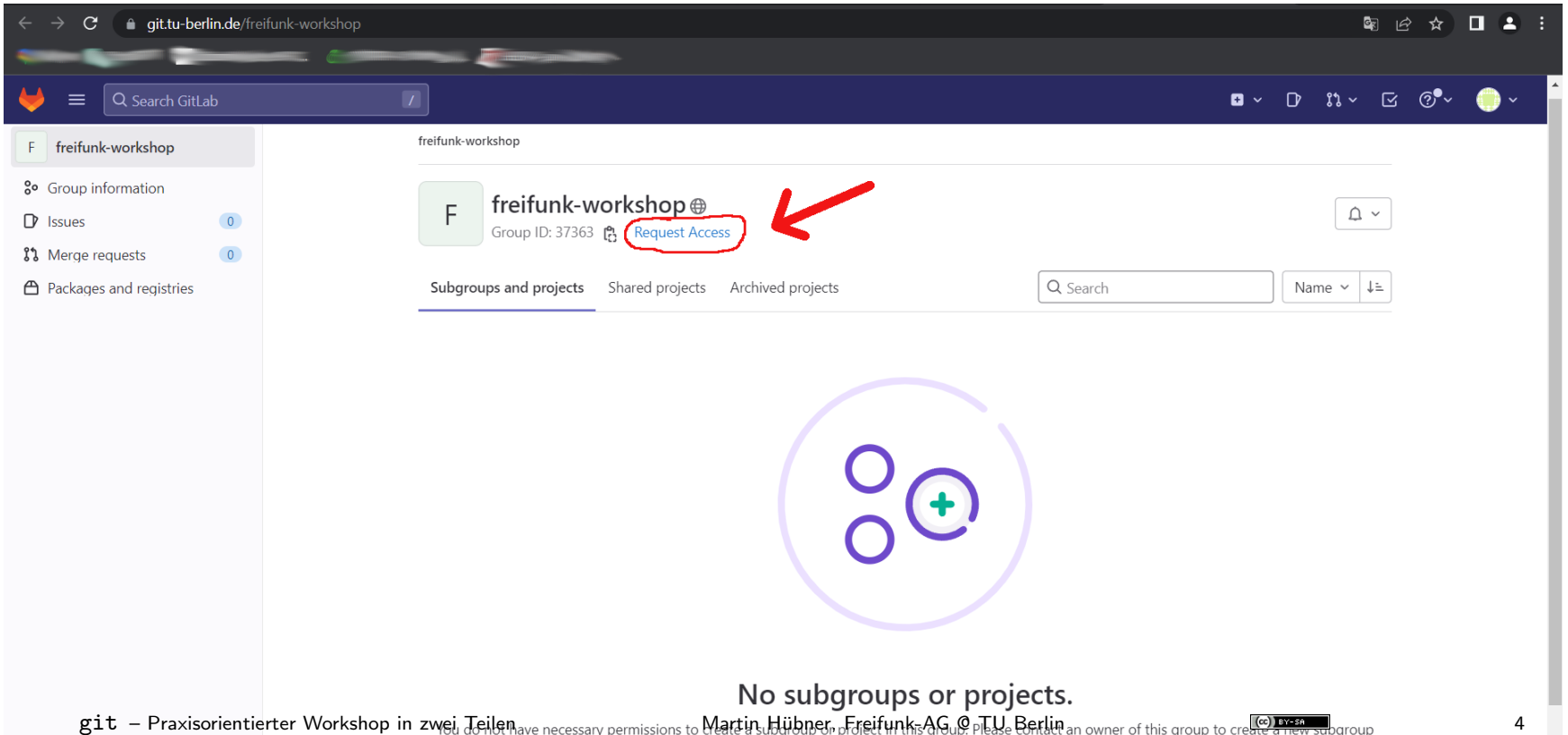
Die Grundlagen. . .

Workshop-Vorbereitung

1. Auf `https://git.tu-berlin.de/` nach Gruppe *freifunk-workshop* suchen

   (oder direkt auf `https://git.tu-berlin.de/freifunk-workshop` gehen)

2. Auf Request-Access drücken (siehe Screenshot)



www.freifunk-ag.org

Was ist Freifunk?

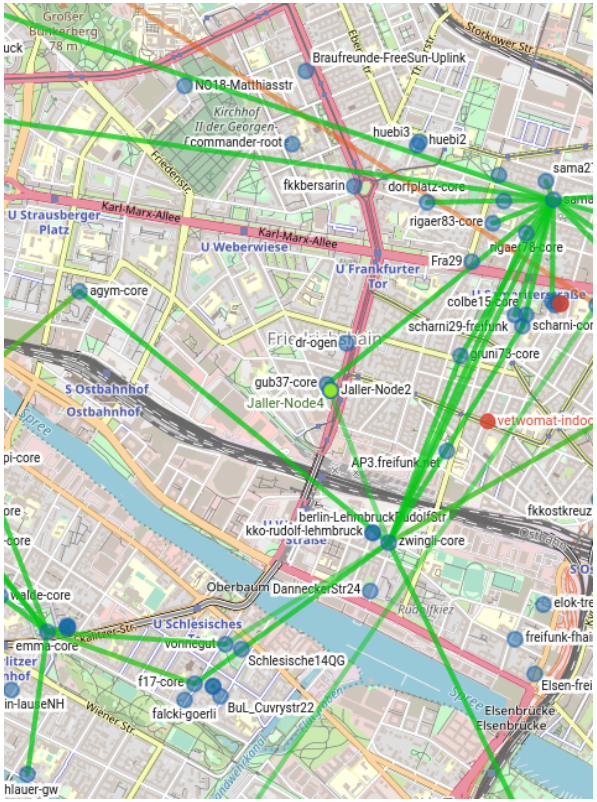
Eine Initiative um offene, vermashte Funknetze aufzubauen. . .

- . . . mit WLAN Routern
- . . . für frei verfügbaren und unzensierten Internetzugang
- . . . für anonyme Nutzungmöglichkeit
- . . . gegen Störerhaftung
- . . . und für den Fun. :)

Freifunk in Berlin

ist. . .

- seit ca. 2003 aktiv, ursprünglich um schnelleres Internet in abgehängte Bezirke zu bringen
- heute Mesh-Netzwerk mit ca 700 Knoten in Berlin und Umgebung
- mit Richtfunk-Standorten auf mehreren Kirchtürmen und hohen Gebäuden



www.freifunk-ag.org

Freifunk@TUB

- gegründet im Oktober 2021
- Freifunk in Studium, Lehre und Forschung bekannter machen
  - „Hardware hacken“ (→ Router mit OpenWRT flashen)
  - regelmäßige Treffen und Workshops zu Freifunk und (embedded-) Software-Entwicklung
  - Studierende an praktische Software-Entwicklung heranführen
- Freifunk-Standort auf dem Dach eines TU-Gebäudes (wieder-)aufbauen.

Kontakt und Treffen

Treffen derzeit Dienstag-Abend gegen 18 Uhr, Raum E131. Website unter `https://freifunk-ag.org/` oder aus dem FF-Netz unter `http://freifunk-ag.olsr`.

www.freifunk-ag.org

Ende des Werbeblocks. . .

www.freifunk-ag.org

Stell dir mal vor. . .

- Du schreibst einen Text. . .
- . . . du stellst an einem Punkt fest, dass du eigentlich alte Zustände behalten möchtest.

Lösung 1

Dateien umbenennen:

- `Arbeit.txt`
- `Arbeit_1.txt`
- `Arbeit_2.txt`
- `Arbeit_fertig.txt`
- `Arbeit_fertig2.txt`
- `Arbeit_fertig-jetzt-aber-wirklich.txt`

⇒ Funktioniert okay, wird aber sehr anstrengend, sobald man mit mehreren Leuten arbeitet und Dateien hin-und-her schickt!

Wir werden Informatikys: Natürlich schreiben wir ein Programm, um das ganze zu lösen! xD

Quelle: pixabay.com

www.freifunk-ag.org

Analyse: Was genau ist eigentlich unser Problem?

- Mehrere Versionen speichern
- Alte Versionen wiederherstellen können
- Übersichtlichkeit: Keine Dateien umbenennen!
- Es muss auch funktionieren, wenn 100 (oder mehr) Leute damit arbeiten

$\Rightarrow$ Es soll möglichst wenig nerven!

Eine Lösung: `git`

`git`: engl. für „Idiot“. `git` soll so einfach sein, dass es auch von „Idioten“ bedient werden kann

- *verteiltes* Versionsverwaltungssystem
- quasi der Branchenstandard
- Am wenigsten nervig...
  - kommt drauf an, wen man fragt...
  - ...und ob die Kollegies sich an bestimmte Regeln halten...

www.freifunk-ag.org

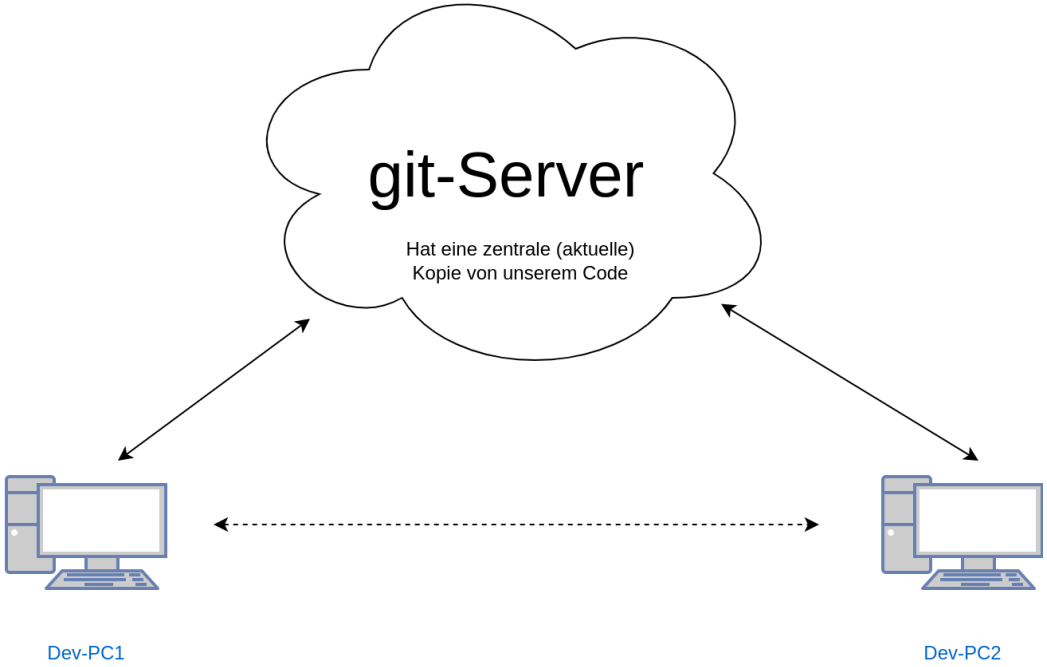
It's workshop time. . .

www.freifunk-ag.org

Vorab: How to play with `git` easily

1. `git` will dir helfen
   - viele ausführliche Fehlermeldungen mit Vorschlägen, was du als nächsten tun kannst (Lesen hilft oft schon weiter. . .)
   - `git status`
2. `git` kann fast alles . . .
   - . . . und deshalb ist es okay, wenn du am Anfang nicht alles verstehst.
3. `--force` kann Dinge kaputt machen!
   - eigentlich selbsterklärend: Nur dann nutzen, wenn man sich wirklich(!) sicher ist

My first `git`-Experience



Anmerkung

Wir benutzen heute ausschließlich Client-Server-Architektur. `git` kann aber auch peer-to-peer benutzt werden!

www.freifunk-ag.org

Let's open the terminal...

BACKUP: Repository clonen

```
~$ git clone https://git.tu-berlin.de/freifunk-ag/\
                                git-workshop-playground.git

Klone nach 'git-workshop-playground'...
remote: Enumerating objects: 48, done.
remote: Total 48 (delta 0), reused 0 (delta 0), pack-reused 48
Empfange Objekte: 100% (48/48), 8.25 KiB | 8.25 MiB/s, fertig.
Löse Unterschiede auf: 100% (25/25), fertig.

~$ cd git-workshop-playground/
~/git-workshop-playground$
```

BACKUP: lokales Repo mit `origin` synchronisieren

```
~/playground$ git pull
Bereits aktuell.

~/playground$
```

BACKUP: Nutzy-Infos im lokalen Repo setzen

```
~/playground$ git config --add user.name "Martin H"

~/playground$ git config --add user.email "martin@example.org"
```

Was man für das Speichern wissen sollte. . .

untracked

Dateien werden nicht auf Änderungen geprüft oder überwacht. Sie existieren nur im lokalen Dateisystem.

```
~/dev/tmp/git-workshop-playground  main  git status
Auf Branch main
Ihr Branch ist auf demselben Stand wie 'origin/main'.

Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
        example.txt

nichts zum Commit vorgemerkt, aber es gibt unversionierte Dateien
(benutzen Sie "git add" zum Versionieren)
```

git add

Staging Area

Datei ist mit Änderungen zum Speichern vorgemerkt. Wird im nächsten Changeset (-> Commit) gespeichert.

```
~/dev/tmp/git-workshop-playground  main  git add example.txt
~/dev/tmp/git-workshop-playground  main +  git status
Auf Branch main
Ihr Branch ist auf demselben Stand wie 'origin/main'.

Zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git restore --staged <Datei>..." zum Entfernen aus der Staging-Area)
        neue Datei:     example.txt
```

git commit

Commited

Datei ist mit ihrem aktuellen Zustand in git erfasst. Kann nun auch hochgeladen werden

```
~/dev/tmp/git-workshop-playground  main +  git commit
[main 6dd4d1a] add an example file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 example.txt
~/dev/tmp/git-workshop-playground  main
```

BACKUP: Code in `git` speichern

```
# Gibt es gerade ungetrackte Änderungen?
~$ git status  Gibt es gerade ungetrackte Änderungen?

# Änderungen im diff-Format anzeigen lassen.
~$ git diff

# Änderungen vormerken: Datei "FILE" zur Staging-Area hinzufügen
~$ git add $FILE
    # um alle Änderungen auf einmal vorzumerken:
    ~$ git add -u

# Änderungen in einem commit speichern
~$ git commit
```

www.freifunk-ag.org

Code in `git` speichern – Anmerkungen

Anmerkung

Bei `git commit` öffnet sich ein Fenster mit einem Editor. In diesem soll eine kurze Beschreibung der Änderungen eingefügt werden. So kann man Änderungen später besser nachvollziehen (→ `git log`).

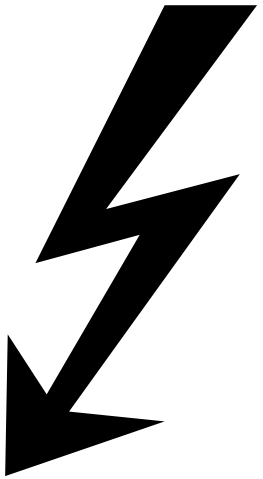
Wichtig!

Bitte `git add *` eher vermeiden! Das fügt auch `.idea/.vscode`-Ordner hinzu. Diese ändern sich sehr oft, enthalten teilweise temporäre Daten und haben deshalb nichts in der Versionsverwaltung zu suchen. Besser: `git add -u` oder `.gitignore` (später mehr).

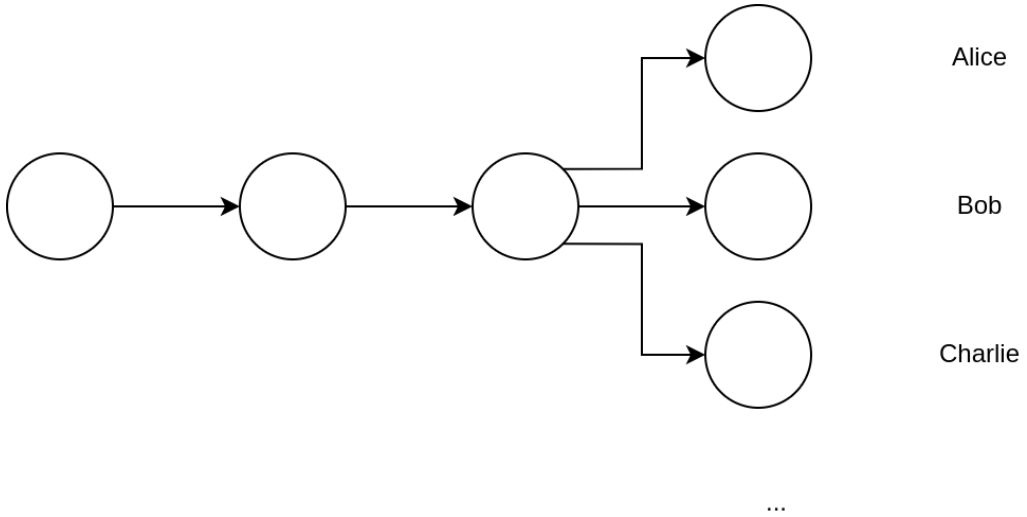
BACKUP: Änderungen hochladen

```
~$ git push
To https://git.tu-berlin.de/freifunk-ag/git-workshop-playground.git
 ! [rejected]        main -> main (non-fast-forward)
Fehler: Fehler beim Versenden einiger Referenzen nach
        'https://git.tu-berlin.de/freifunk-ag/git-workshop-playground.git'
Hinweis: Aktualisierungen wurden zurückgewiesen, weil die Spitze Ihres
Hinweis: aktuellen Branches hinter seinem externen Gegenstück zurückgefallen
Hinweis: ist. Führen Sie die externen Änderungen zusammen (z. B. 'git pull ...')
Hinweis: bevor Sie "push" erneut ausführen.
Hinweis: Siehe auch die Sektion 'Note about fast-forwards' in 'git push --help'
Hinweis: für weitere Details
```

Konflikte. . .



Was ist passiert?



Wir haben alle die gleiche Zeile bearbeitet. *Wer hat denn nun recht*?

$\Rightarrow$ keine Automatische Auflösung möglich. Entwicklys müssen eingreifen.

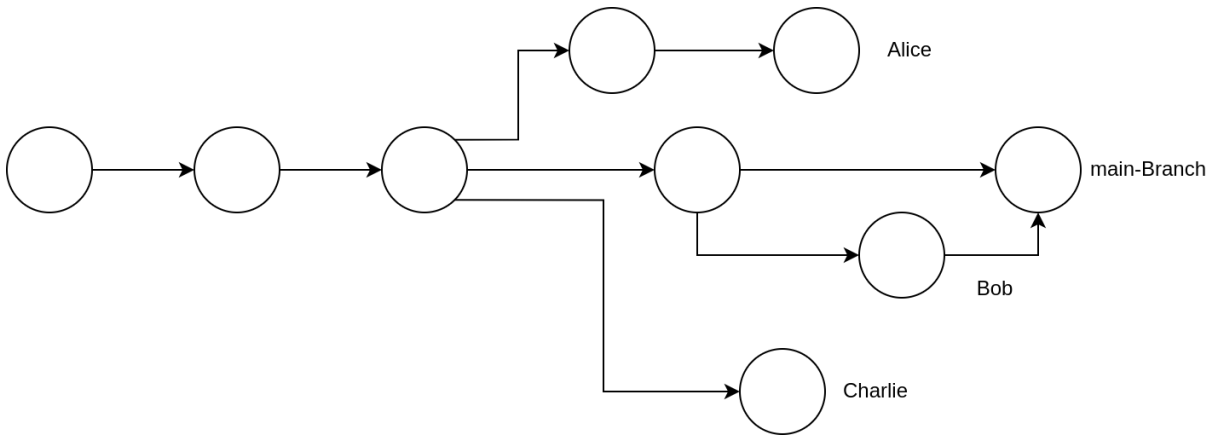
www.freifunk-ag.org

...und wie lösen wir das ganze?

Zwei Möglichkeiten:

1. vor jedem Commit einmal `git pull` ausführen.
   - etwas unpraktisch: Was ist, wenn jemand Anderes große Änderungen gemacht hat?
2. → `git`-**Branches**

Wofür sind Branches (Zweige) gut?



- jeder arbeitet erstmal „für sich“, ohne Konflikte mit den Anderen
- Änderungen können „der Reihe nach“ in main-branch übernommen werden
  - Es wird immernoch Konflikte geben. Aber wir haben die Chance, sie zu reparieren. ;-)

Wofür sind Branches (Zweige) gut?

Anmerkung

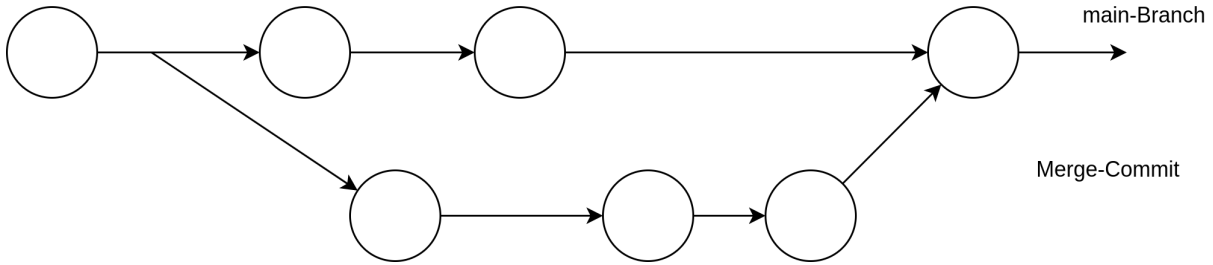
Erfahrene `git`-Nutzys kennen noch `git checkout`. Seit Version 2.23 (Q3.2019) wurde die Funktionalität in zwei separate Kommandos getrennt: `git switch` und `git reset`. Wir werden diese beiden statt `git checkout` verwenden.

Siehe `https://stackoverflow.com/a/57066202`

BACKUP: Das ganze nochmal mit Branches

```
$ git clone https://git.tu-berlin.de/freifunk-ag/\
                                git-workshop-playground.git

$ cd git-workshop-playground/
$ git switch -c MY_BRANCH
$ git status

# eigene Änderungen durchführen

$ git add -u # fügt alle Änderungen an getrackten Dateien hinzu
$ git commit # Bitte sinnvolle Commit-Message schreiben
$ git push # wird fehlschlagen
$ git push --set-upstream origin MY_BRANCH

# Weiter im Gitlab-Interface
```

Branches zusammenführen: `git merge`

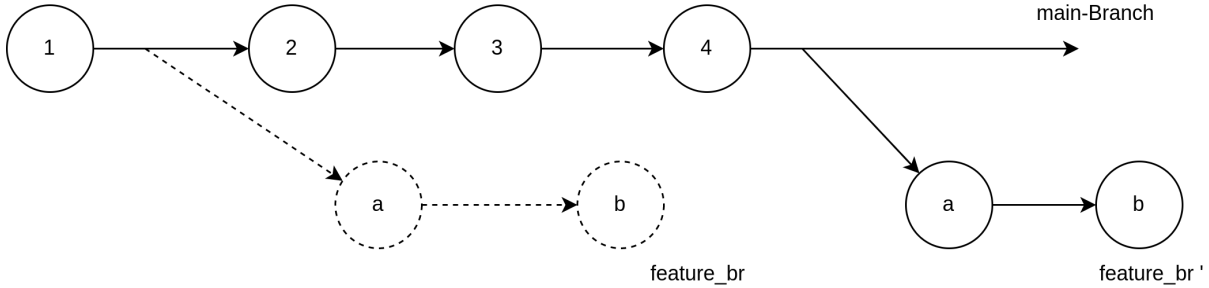


`git merge` führt Änderungen aus untersch. Branches zusammen.

- `git merge` erzeugt sog. Merge-Commit in der History
- Commits können in der history komisch sortiert angezeigt werden
  - mit `git log --graph` kann man anschauen, aus welchem Branch Commits kamen.

⇒ Sortierung kann verwirrend sein. Eventuell lieber `git rebase` nutzen.

Was ist ein rebase?



`git rebase` macht ähnliche Sachen wie `merge`, aber anders...

- + `git rebase` erzeugt keinen Merge-Commit
- + die entstehende History liest sich komplett linear
- – Commits sind nicht mehr zu branches zugeordnet
- – manche Personen finden `rebase` complex

Achtung!

Wenn man beim Auflösen der Konflikte nicht aufpasst, kann Code verloren gehen.

BACKUP: Howto rebase locally

```
$ git clone https://git.tu-berlin.de/freifunk-ag/\
                                git-workshop-playground.git

$ cd git-workshop-playground/
$ git switch -c MY_BRANCH
$ git status

# eigene Änderungen durchführen

$ git add -u # fügt alle Änderungen an getrackten Dateien hinzu
$ git commit # Bitte sinnvolle Commit-Message schreiben
$ git push # wird fehlschlagen
$ git push --set-upstream origin MY_BRANCH

# Weiter im Gitlab-Interface
```

Teil Zwei.

Advanced Stuff. . .

www.freifunk-ag.org

Best Practices: Commit-Messages

Sinvoller Nutzen von Commits:

- Enthält eine Änderung mit einem bestimmten Zweck
- Zwei Änderungen an unterschiedlichen Dateien haben gleichen Sinn? Eventuell in gleichen Commit packen

Gedanken zu Merge-Requests:

- viele Open-Source-Projekte nutzen Peer-Review
- Merge-Request/Branch ist zweckgebunden: Ein Feature implementieren oder Fehler fixen

$\Rightarrow$ Mit sinnvollen Commit-Messages hilft man dem Reviewer, Änderungen zu verstehen und so schneller zu mergen.

Commit-Messages: Beispiel

```
build_falter.sh: better error checking

build_falter will now check for more dependencies. In addition,
we activated errexit in the shell for every bit of the script,
except the imagebuilder. In the imagbuilder there's the possi-
bility that one single error might break the whole build (for
a whole target).

Fixes #116
```

Wann einen separaten Branch machen?

TLDR; Eigentlich immer.
Branches werden meistens fürs Implementieren eines Features oder einen Fehler-fix angelegt.

Branches

- ermöglichen, parallel an mehreren Dingen zu arbeiten
- sind themenbezogen. Eigentlich nicht auf Personen
- sind wie Hygieneartikel: Bitte nicht mehrfach verwenden. . . („Mein Branch“)

www.freifunk-ag.org

Best Practices: `.gitignore`

Manche Dateien in unserer lokalen Repo-Kopie sollen nicht Teil der Versionsverwaltung sein:

- Binärdateien (build-Artefakte, alles was sich aus Quellcode erzeugen lässt, etc)
- Einstellungen der IDE (`.vscode` oder `.idea`-Ordner)

Wir können `git` sagen, dass es diese Dateien immer ignorieren soll:

```
# Diese Ordner ignorieren
.vscode/
.idea/
build/
```

`git add`

Fügt Dateien der Staging Area hinzu (siehe erster Teil). Useful options:

- `-u` füge nur Änderungen aus bereits getrackten Dateien hinzu
- `-p FILE` gehe einzelne Stellen durch und frage separat, ob hinzufügen

Ich mag diese Kombination sehr: `git add -up`

git blame

Zeige für eine Datei an, welche Zeile von wem in welchem Commit bearbeitet wurde:

```
$ git blame build_falter

f6133a4d (Martin     2020-11-23 23:45:56 +0100 323)     else
58d12789 (Martin     2021-03-21 18:23:41 +0100 324)         rsync -a #[...]
be0f62f6 (pmelange   2020-11-23 13:53:59 +0100 325)     fi
^9689fe7 (Simon      2020-11-15 21:14:24 +0100 326)
^9689fe7 (Simon      2020-11-15 21:14:24 +0100 327)     cd ..
^9689fe7 (Simon      2020-11-15 21:14:24 +0100 328) }
^9689fe7 (Simon      2020-11-15 21:14:24 +0100 329)
```

`git bisect`

Finde einen Commit, der einen Bug eingeführt hat.

- Binäre Suche auf der Commithistory.
- in jedem Schritt manuell testen und Frage beantworten:
  - Tritt fehler in diesem Commit auf? ja/nein

Zitat von Jakob

Aber dafür müssen auch alle Commits in deinem Repo ausführbaren Code enthalten. :)

BACKUP: git bisect

```
$ git bisect start
Status: warte auf guten und schlechten Commit
$ git bisect bad
Status: warte auf gute(n) Commit(s), schlechter Commit bekannt
$ git bisect good 56bab5f53d8ad22936d857d0c2e510346a009e56
Binäre Suche: danach noch 12 Commits zum Testen übrig (ungefähr 4 Schritte)
[a7ca77698c77c0b5c73566ad0951de018f90a55c] luci-app-ffwizard-falter: swap German translation
$ git bisect good
[...]
$ git bisect bad
Binäre Suche: danach noch 0 Commits zum Testen übrig (ungefähr 1 Schritt)
[55bac9e5735331e8fe256f5e43a5a5473028b1dd] workflows: fix yaml-lint issues and enable yaml-lint for the repo
$ git bisect bad
Binäre Suche: danach noch 0 Commits zum Testen übrig (ungefähr 0 Schritte)
[b437c884df51ee6e7c317fec4f16b455b95984f7] workflows: add ShellCheck and ignore existing files with issues
$ git bisect bad
b437c884df51ee6e7c317fec4f16b455b95984f7 is the first bad commit
commit b437c884df51ee6e7c317fec4f16b455b95984f7
Author: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Date:   Tue Feb 7 17:25:26 2023 +0000

    workflows: add ShellCheck and ignore existing files with issues

 .github/workflows/shellcheck-lint.yml | 45 +++++++++++++++++++++++++++++++++++
 1 file changed, 45 insertions(+)
 create mode 100644 .github/workflows/shellcheck-lint.yml

 $ git bisect reset
```

`git rebase --interactive`: Geschichte (neu-)schreiben

Aus dem Alltag:

- „Hatte keine Lust, gute Commit-msg zu schreiben“
- „Musste committen, um an anderem Rechner weiterzuarbeiten“
- „Ach, da soll man was reinschreiben?“

Mit `git rebase --interactive HEAD~n` für $n \in \mathbb{N}$ kann man die Commit-History auf den letzten n Commits bearbeiten: Messages anpassen, Commits zusammenfassen, oder sogar Reihenfolge ändern.

`git rebase --interactive`: Geschichte (neu-)schreiben

```
pick 98a9914 falter-berlin-migration: add migration for 1.2.3-release
edit 97f04e2 falter-berlin-tunnelmanager: wait for mesh daemons to be started
pick b2a8bb5 OWM: update twice every hour
drop b385a06 falter-berlin-migration: Apply shfmt
pick 4f1d7f1 migration: fix bashism

# Rebase von 389074f..4f1d7f1 auf 389074f (5 Kommandos)
```

Achtung!

Nicht jede Änderung ist sinnvoll. `git` geht davon aus, dass ihr wisst, was ihr tut. Es macht genau das, was ihr verlangt. Gefahr von Datenverlust!

git revert

Erzeugt einen Commit in der history, der einen bestimmten Commit rückgängig macht.

```
$ git revert 83c88908ee3dfd53b8013731f9a44fc04d3562c9
[master 8eb65df] Revert "athwatch: rip"
 2 files changed, 50 insertions(+)
 create mode 100644 packages/athwatch/Makefile
 create mode 100644 packages/athwatch/files/athwatch.init
```

Tipp

`revert` kann eine gute Alternative zum dropen per `rebase --interactive` sein: Revert kann man in der commit-hiistory nachvollziehen. Dadurch, dass bestehende Commits erhalten bleiben, entstehen keine Merge-Konflikte.

git cherry-pick

Nimm den genannten Commit und wende ihn auf den aktuellen Branch an:

```
$ git cherry-pick 1d85c05897e9bc6ceb619338db4bce4ff71d0a1a
[main 93c4f90] test
 Date: Thu Feb 16 14:27:57 2023 +0100
 1 file changed, 2 insertions(+)
```

git stash

Der Chef kommt rein und möchte, dass du ganz dringend deine Arbeit unterbrichst und schnell etwas anderes fixt. Deine bisherige Arbeit ist aber noch nicht bereit zum Commiten.

```
$ git stash

# Anderen stuff machen
[...]

$ git stash apply
# Zustand wiederhergestellt und bereit zum weitermachen
```

Mit `git stash` können wir Zwischenstände in ein "Versteck" packen. `stash` funktioniert dabei wie ein Stack.

git reflog

„`reflog` ist git auf git“[1]

`reflog` zeigt die Updates von Branchspitzen an. Ohne auf Interna einzugehen:

Wir können unter andem

- branch switches,
- pulls,
- merges,
- rebases

etc, sehen. Sehr praktisch, um missglückte rebases zu reparieren.

[1] Carsten, Gesprächsnotiz, irgendwann im Februar 2023

BACKUP: git reflog

```
$ git reflog

[...]
f1e99a6 (haha) HEAD@{17}: checkout: moving from main to haha
f1e99a6 (haha) HEAD@{18}: commit (merge): Merge branch 'clemens'
22394dd HEAD@{19}: checkout: moving from clemens to main
ecb0698 (origin/clemens, clemens) HEAD@{20}: checkout: moving from main to clemen
22394dd HEAD@{21}: pull: Fast-forward
dec8e62 HEAD@{22}: checkout: moving from leo to main
a47c7f9 (origin/leo, leo) HEAD@{23}: rebase (continue) (finish): returning to ref
a47c7f9 (origin/leo, leo) HEAD@{24}: rebase (continue): Mehr Korrektur
dec8e62 HEAD@{25}: rebase (start): checkout main
8f18298 HEAD@{26}: checkout: moving from main to leo
dec8e62 HEAD@{27}: pull: Fast-forward
b4add6f HEAD@{28}: pull: Fast-forward
fc6c3b9 HEAD@{29}: checkout: moving from clemens to main
ecb0698 (origin/clemens, clemens) HEAD@{30}: pull: Fast-forward
9be006f HEAD@{31}: checkout: moving from lu to clemens
b137b8b (origin/lu, lu) HEAD@{32}: pull: Fast-forward
[...]
```

www.freifunk-ag.org

Änderungen anhängen

Ich habe etwas vergessen, aber es gehört sinngemäß noch zum aktuellen Commit dazu. Wie kann ich es anfügen?

```
$ git add FILE
$ git commit --amend

# Falls ursprünglicher COmmit schon gepusht:
$ git push --force
```

Beachte

Bitte nur auf persönlichen Branches force-pushen, die nicht von Anderen genutzt werden. Sonst unschöne Konflikte mit anderen Repo-Kopien.

www.freifunk-ag.org

Änderungen einfügen

Ich habe vergessen, etwas zu commiten. Aber der Commit, wo es eigentlich reingehört, liegt 4 Commits vor dem aktuellen.

```
$ git stash
$ git rebase --interactive HEAD~5

# Im Editorfenster den entsprechenden Commit von 'pick'
# auf 'edit' ändern

$ git stash apply
($ git add -u)
($ git status)
$ git commit --amend
$ git rebase --continue
```

Datei aus anderem Branch hierher kopieren

Ich möchten den Stand einer Datei aus einem anderen Branch hierher kopieren.

```
# Falls Branch noch nicht lokal ausgecheckt, einmal machen:
$ git switch OTHERBRANCH
$ git switch BRANCH

$ git reset OTHERBRANCH -- DATEINAME
```

www.freifunk-ag.org

Ende. . .