



**JAVACREAM**

*Training  
Consulting  
Projectmanagement*

# GIT

- Name
- Rolle im Unternehmen
- Themenbezogene Vorkenntnisse
- Aktuelle Problemstellung
- Konkrete individuelle Zielsetzung



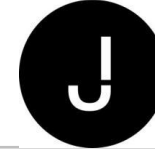
## Ausgangssituation

- Ein Satz von Dateien repräsentiert einen Stand eines Projekts
  - angereichert mit Meta-Informationen
    - “Wer hat wann warum welche Änderungen gemacht”
- Parallele Fortentwicklung von Ständen
- Konsistente Zusammenführen (“Mergen”) von parallel entwickelten Ständen
- Zentrale Ablage aller Informationen
  - Authentifizierung und Autorisierung
- Verfahren und Methoden zur Zusammenarbeit im Team
- Unterstützung durch etablierte Verfahren zum effizienten Arbeiten

- Ein Satz von Dateien repräsentiert einen Stand eines Projekts
  - angereichert mit Meta-Informationen
    - “Wer hat wann warum welche Änderungen gemacht”
- Parallele Fortentwicklung von Ständen
- Konsistente Zusammenführen (“Mergen”) von parallel entwickelten Ständen
- Zentrale Ablage aller Informationen
  - Authentifizierung und Autorisierung
- Verfahren und Methoden zur Zusammenarbeit im Team
- Unterstützung durch etablierte Verfahren zum effizienten Arbeiten

- Ein Satz von Dateien repräsentiert einen Stand eines Projekts
  - angereichert mit Meta-Informationen
    - “Wer hat wann warum welche Änderungen gemacht”
- Parallele Fortentwicklung von Ständen
- Konsistente Zusammenführen (“Mergen”) von parallel entwickelten Ständen
- Zentrale Ablage aller Informationen
  - Authentifizierung und Autorisierung
- Verfahren und Methoden zur Zusammenarbeit im Team
- Unterstützung durch etablierte Verfahren zum effizienten Arbeiten

- Git Server sind Produkte von Herstellern außerhalb der Git-Community
  - Proprietär
  - Aufgrund der klaren Problemstellung sind aber die Produkte sehr ähnlich
- Übersicht
  - **GitHub Enterprise** (Microsoft)
  - GitLab (gitlab.com)
  - BitBucket (Atlassian)

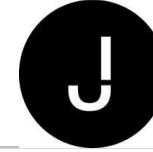


- Ein Satz von Dateien repräsentiert einen Stand eines Projekts
  - angereichert mit Meta-Informationen
    - “Wer hat wann warum welche Änderungen gemacht”
- Parallele Fortentwicklung von Ständen
- Konsistente Zusammenführen (“Mergen”) von parallel entwickelten Ständen
- Zentrale Ablage aller Informationen
  - Authentifizierung und Autorisierung
- Verfahren und Methoden zur Zusammenarbeit im Team
- Unterstützung durch etablierte Verfahren zum effizienten Arbeiten





- Workflows zum effizienten Arbeiten mit Git
- Beispiele
  - Git Flow (Atlassian)
  - GitHub Flow (GitHub-Community)
- Hinweise
  - Sowohl Atlassian Git Flow als auch GitHub Flow sind Produkt-unabhängig
  - Git Flows sind immer als Templates zu verstehen, die im konkreten Einsatz an ein Projekt / ein Unternehmen angepasst werden



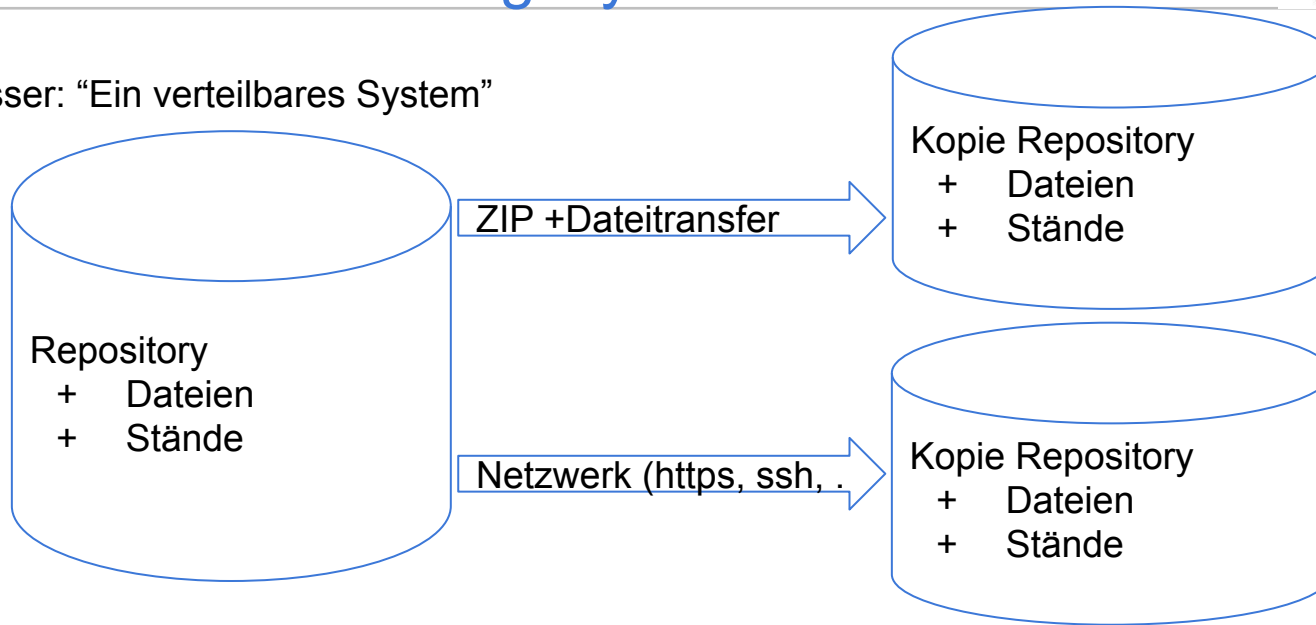
- Ein Satz von Dateien repräsentiert einen Stand eines Projekts
  - angereichert mit Meta-Informationen
    - “Wer hat wann warum welche Änderungen gemacht”
- Parallele Fortentwicklung von Ständen
- Konsistente Zusammenführen (“Mergen”) von parallel entwickelten Ständen
- Zentrale Ablage aller Informationen
  - Authentifizierung und Autorisierung
- Verfahren und Methoden zur Zusammenarbeit im Team
- Unterstützung durch etablierte Verfahren zum effizienten Arbeiten

Tag 1 + Tag 2 erste  
Session

# Git ist ein “verteiltes Versionsverwaltungssystem”



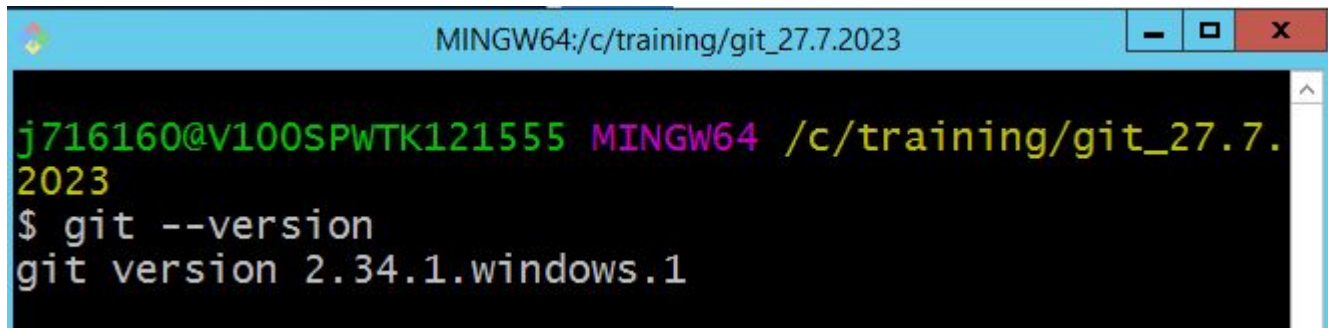
Besser: “Ein verteilbares System”



Unbedingt notwendig ist eine Fälschungs-sichere Konsistenz der verteilten Informationen  
Lösung wird durch Merkle-Trees erreicht (Informationen bestimmen einen Hashwert, Der Hash der Vorgänger-Information wird dabei mitberücksichtigt) -> Blockchain  
GIT benutzt / arbeitet mit Blockchain-Technologie

## First Contact

- Kommandos über ein Terminal

A screenshot of a Windows terminal window. The title bar is light blue and contains the text 'MINGW64:/c/training/git\_27.7.2023' and standard window controls. The terminal background is black with green and white text. The prompt 'j716160@V100SPWTK121555' is followed by 'MINGW64 /c/training/git\_27.7.2023'. The command '\$ git --version' has been entered, and the output 'git version 2.34.1.windows.1' is displayed.

```
MINGW64:/c/training/git_27.7.2023

j716160@V100SPWTK121555 MINGW64 /c/training/git_27.7.2023
$ git --version
git version 2.34.1.windows.1
```

- Installation native oder “Portable Git”
  - Git ist kein Hintergrund-Dienst
  - Die Git-Kommandos stellen das komplette Versionsverwaltungssystem bereit

- Konfiguration von Benutzer-Name und eMail-Adresse
  - `git config --global user.name`
  - `git config --global user.email`
- Hinweise
  - Bei Ihnen bereits gesetzt
    - Check: `git config --get user.name`
  - `--global`
    - Global heißt: Für den Benutzer, `.gitconfig` im UserHome-Directory
    - `--local` (pro Repo) oder `--system` (Rechner-übergreifend)
  - `-gitconfig` ist eine strukturierte Text-Datei

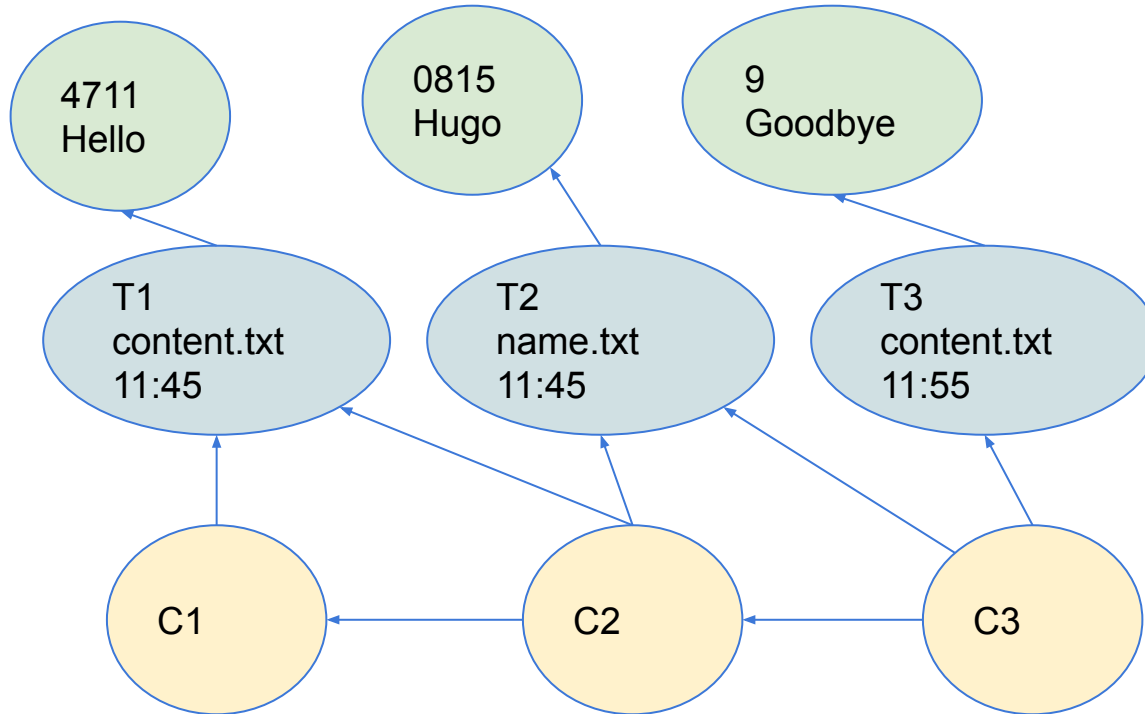
- **Untypisch: Initialisieren eines lokalen, leeren Repositories**
  - Im Seminar erst einmal die einzige Möglichkeit
  - In der Praxis würden Sie ein Repo auf GitHub anlegen und clonen -> später
- **mkdir training, cd training**
  - training ist ein ganz normales Verzeichnis
- **git init**
  - Erzeugt in training das Unterverzeichnis .git
  - **Wording**
    - -git -> Git Repository
    - training ist immer noch “normal”, aber ich nenne es ein Git-Projektverzeichnis
      - Alle Dateien außerhalb von -git sind Bestandteil des Workspaces
  - **Check**
    - Im Prompt steht “(master)”
    - git status erzeugt nur unauffällige Ausgaben
- Hinweis: Ein Löschen des Verzeichnisses ,git zerstört das Repository unwiderruflich

- Erstellen einer Datei im Workspace
  - `echo Hello > content.txt`
- `git status` zeigt eine Inkonsistenz an: Es existiert eine Datei im Workspace, die dem Repository unbekannt ist
  - Datei ist “rot”
    - Kein Fehler!
- `git add content.txt`
  - Parameter: Liste von Dateien mit Platzhaltern
- `git status` zeigt eine Inkonsistenz an: Das Repository enthält in einer “Staging Area” eine Datei, die noch keinem Stand zugeordnet ist
  - Datei ist “grün”
    - Es ist nicht alles “OK”
-



- Definition eines Standes erfordert das Erfassen der Meta-Informationen
  - Wer (user.name) hat wann (timestamp) warum (?) welche (Informationen in der Staging Area) gemacht
  - `git commit -m "Beschreibung: warum? - > Commit Message"`
- `git status` ist unauffällig
- `git log`
  - Ausgabe der letzten Meta-Informationen eines Standes

- Normales Arbeiten im Workspace
- Git
  - `git add .`
  - `git commit -m "neuer Stand"`



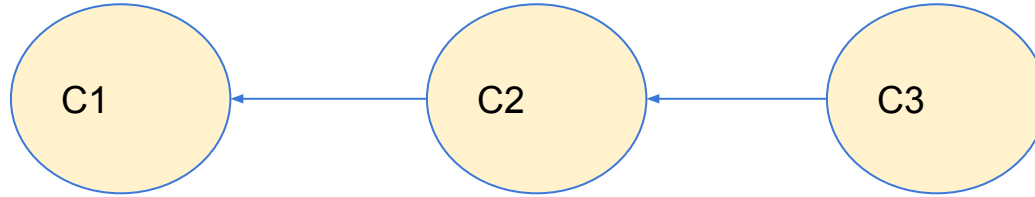
Content oder BLOB  
Objekt

Tree Objekt

Commit Objekt



- `git add`
  - erzeugt ausschließlich Content-Objekte
- `git commit`
  - erzeugt die Tree-Objekte
  - Erzeugt ein neues Commit-Objekt mit Verweis auf den Vorgänger
    - Blockchain



Commit Objekt

git commit

- + Erzeugt ein neues Commit-Objekt mit Verweis auf den Vorgänger

- `git checkout <hash>`
  - Damit wird der Workspace mit einem Stand synchronisiert = überschrieben
- Empfehlung Sawitzki
  - “Checkout nur bei unauffälligen Status”
  - Falls auffällig
    - `git add .`
      - `git commit -m “...”`
      - `git stash`
        - Im Wesentlichen ein lokaler Backup der Staging-Area
          - Bitte aber nur im “Notfall”
          - Details -> Dokumentation / Online

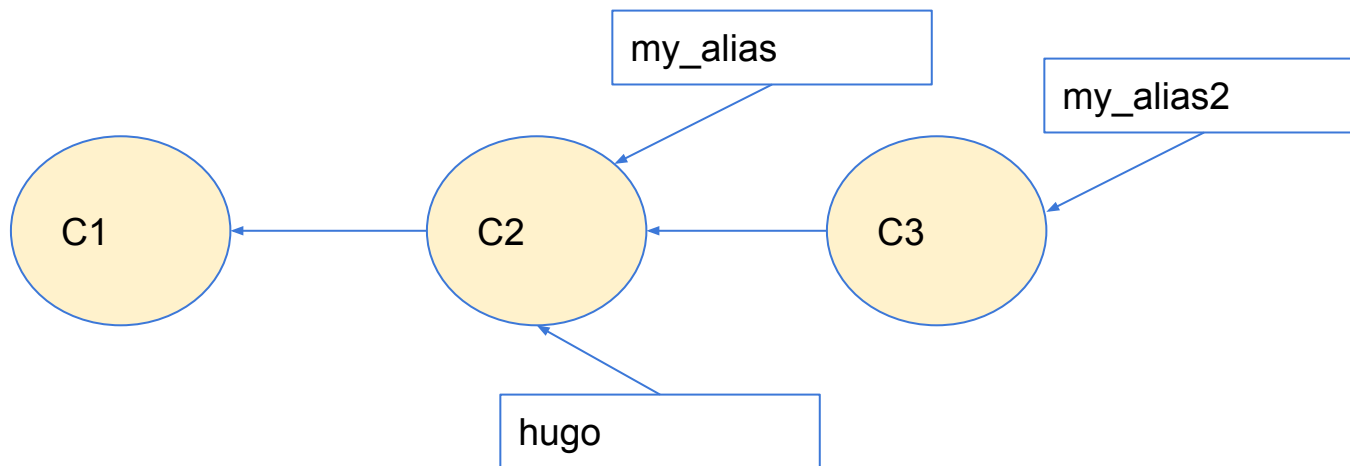
- Nachvollziehen
- Ist ein Commit auch auf “alten” Ständen möglich?
- Wie würden Sie das im Graphen der Commit-Objekte visualisieren?
- Vorsicht:
  - Behalten Sie die Hash-Werte in der Historie, ein git log zeigt nicht immer alle Hashes an
-



- Eigentlich genügt das bisher vermittelte Wissen...
- Das Arbeiten mit den Hash-Werten ist intern in Git, aber aus Sicht eines Anwenders ist das der “Nerd-Modus”



- Ein Alias-Name macht einen Commit auch über einen sprechenden Namen identifizierbar





- Alias-Name auf einen festen Stand
  - Release mit einer Versionsnummer
    - v1.1
  - Milestones, Build-Number
  - Merker
    - “heute morgen”, test, savepoint, ...
- Alias-Name auf eine gerade durchgeführte Aktion
  - Auftrag
    - implement\_feature1
  - Jira-Issue
    - jira\_12345
  - Merker
    - experiment\_42, working,

- Fixer Stand
  - `git tag v1.0`
    - Erzeugt ein Tag auf dem aktuellen Commit
    - `git tag v1.0 23456787654323456`
  - `git tag --list`
    - Auflistung aller Tags
      - Auch interessant für eine Projekt-Übersicht, “Was haben wir denn schon alles erreicht?”
  - `git tag -d v1.0`
- Hinweis für FI
  - Tags in der FI können nur mit einer Signatur auf GitHub hochgespielt werden

- Laufende Aktion
  - `git branch implement_new_feature`
    - Erzeugt einen Branch auf dem aktuellen Commit
    - `git branch my_branch 23456787654323456`
  - `git branch --list`
    - Auflistung aller Branches
      - Auch interessant für eine Projekt-Übersicht, “Was machen wir denn gerade?”
  - `git branch -d my_branch`
- Hinweise
  - Erzeugen eines Branches wird häufig mit einem checkout kombiniert
    - `git checkout -b new_branch <hash>`

- Tags und Branches sind und bleiben Alias-Namen auf vorhandene Commit-Objekte
- Tags und Branches sind Trivial-Operationen
  - Anlegen
    - -> CPU = 0%
    - -> Repository-Größe += ein paar Byte
- Löschen eines Tags oder Branches ändert im Geflecht der Commit-Objekte NICHTS
- Alias-Namen müssen im Repository eindeutig benannt sein

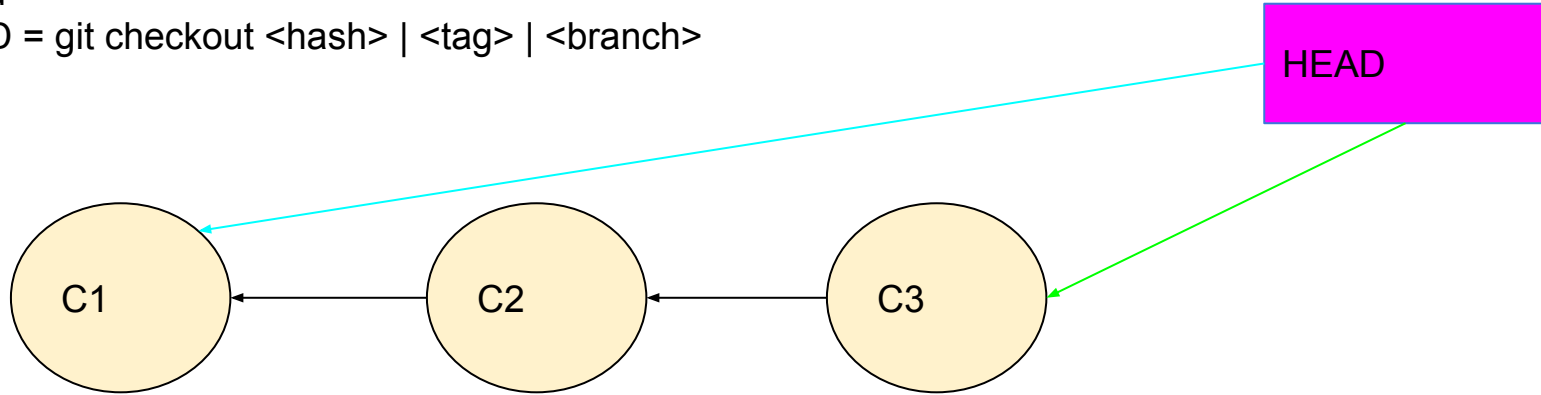


- `git log --oneline --graph --decorate --all`
  - Erzeugt eine übersichtliche Kompakt-Ausgabe des Commit-Geflechts
    - Damit wird die Historie des Projekts simpel visualisiert
- `git config --global alias.dl "log --oneline --graph --decorate --all"`
- `git dl`

```
* 2f8ad55 (feature/planet_part2) add Jupiter
* d8bc94a add Uranus
| * 45e577a (feature/planet_part1) add Mercury
|/
* ff6e19f (feature/planet) add venus
| * b08ec94 (feature/star) change to Sol
| * 4e601b1 add vega
|/
* 29df802 (HEAD -> main, tag: v1.0) setup project
```

Alias-Name "HEAD" zeigt auf den gerade aktiv ausgecheckte Stand

HEAD = git checkout <hash> | <tag> | <branch>



```
git checkout C1  
git checkout C3
```



- `git checkout <hash> | <tag>`

```
$ git d1
* 2f8ad55 (feature/planet_part2) add Jupiter
* d8bc94a add Uranus
| * 45e577a (feature/planet_part1) add Mercury
|/
* ff6e19f (feature/planet) add venus
| * b08ec94 (feature/star) change to Sol
| * 4e601b1 add vega
|/
* 29df802 (HEAD, tag: v1.0, main) setup project
```

Ergebnis: Detached HEAD

- git checkout main

```
$ git d1
* 2f8ad55 (feature/planet_part2) add Jupiter
* d8bc94a add Uranus
| * 45e577a (feature/planet_part1) add Mercury
|/
* ff6e19f (feature/planet) add venus
| * b08ec94 (feature/star) change to Sol
| * 4e601b1 add vega
|/
* 29df802 (HEAD -> main, tag: v1.0) setup project
```

Ergebnis: HEAD attached auf den Branch

# Unterschied attached / detached HEAD

Manifestiert sich bei einem Commit

Beim Commit mit attached HEAD wird der Branch bewegt, der HEAD deutet weiter auf den Branch

```
$ git dl
* 954b2fd (HEAD -> main) add name
* 2f8ad55 (feature/planet_part2) add Jupiter
* d8bc94a add Uranus
| * 45e577a (feature/planet_part1) add Mercury
|/
* ff6e19f (feature/planet) add venus
/
* b08ec94 (feature/star) change to Sol
* 4e601b1 add vega
/
* 29df802 (tag: v1.0) setup project
```

```
git checkout main
echo...
git add .
git commit -m "..."
```

# Unterschied attached / detached HEAD

Manifestiert sich bei einem Commit

Beim Commit auf detached HEAD deutet der HEAD auf das neu erzeugt Commit-Objekt -> NERD-Modus

```
$ git d1
* fa812af (HEAD) add name
| * 954b2fd (main) add name
|/
| * 2f8ad55 (feature/planet_part2) add Jupiter
| * d8bc94a add Uranus
| | * 45e577a (feature/planet_part1) add Mercury
| | /
| * ff6e19f (feature/planet) add venus
|/
| * b08ec94 (feature/star) change to Sol
| * 4e601b1 add vega
|/
* 29df802 (tag: v1.0) setup project
```

```
git checkout 29df802
echo...
git add .
git commit -m "..."
```

Falls irrtümlich im detached HEAD committed -> git checkout -b new\_branch

## Zusammenfassen von Ständen



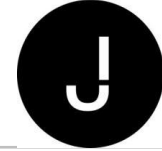
- Der main-Branch (= aktuelle Projekt-Stand) soll beide neuen Features enthalten
- Die Commit-Historie soll alle relevanten durchgeführten Aktionen detailliert enthalten

- feature/planet soll die beiden Parts inkludieren
  - Dabei können Konflikte auftreten, da potenziell konkurrierende Änderungen durchgeführt wurden
    - planet.txt
- feature/planet soll auch feature/star inkludieren
  - Dabei können Konflikte auftreten, da potenziell konkurrierende Änderungen durchgeführt wurden
    - es kommt die Datei star.txt mit dazu
- Zum Abschluss soll der main auf den Stand von feature/planet (neu) gezogen werden
  - Hier darf kein Konflikt auftreten

```
* f315c73 (feature/planet_part2) add Jupiter
* 9a0c7d4 add Uranus
* 5183b3f (feature/planet_part1) add Mercury
* a7e5c8e (feature/planet) add venus
* bae9486 (feature/star) change to Sol
* cdc5f89 add vega
* e717155 (HEAD -> main, tag: v1.0) setup project
```



# git checkout feature/planet

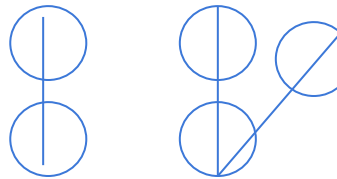


```
$ git dl
* f315c73 (feature/planet_part2) add Jupiter
* 9a0c7d4 add Uranus
| * 5183b3f (feature/planet_part1) add Mercury
|/
* a7e5c8e (HEAD -> feature/planet) add venus
| * bae9486 (feature/star) change to Sol
| * cdc5f89 add vega
|/
* e717155 (tag: v1.0, main) setup project
```

git merge <source\_commit> <target\_commit>  
git merge <target\_commit> -> source = aktueller Stand

Arbeitsweise:

- + Git prüft, ob zwischen Source und Target eine gerade Verbindungslinie besteht
- + Falls Ja (wie in diesem Fall), wird ein Fast Forward Merge ausgeführt



```
$ git merge feature/planet_part1
Updating a7e5c8e..5183b3f
Fast-forward
 planet.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

j716160@V100SPWTK121555 MINGW64 /c/training/git_27.7.
2023/training_branches (feature/planet)
$ git dl
* f315c73 (feature/planet_part2) add Jupiter
* 9a0c7d4 add Uranus
* 5183b3f (HEAD -> feature/planet, feature/planet_p
art1) add Mercury
/
* a7e5c8e add venus
* bae9486 (feature/star) change to Sol
* cdc5f89 add vega
/
* e717155 (tag: v1.0, main) setup project
```

## Arbeitsweise:

- + Git prüft, ob zwischen Source und Target eine gerade Verbindungslinie besteht
- + Falls Nein (wie in diesem Fall), wird ein Recursive Merge ausgeführt
  - + Konflikte sind möglich und können von Git erkannt und in Ausnahmefällen automatisch gelöst werden
  - + VORSICHT: Das Ergebnis einer Auto-Conflict-Resolution muss IMMER geprüft werden

Konflikte auflösen = planet.txt editieren,  
speichern

git add planet.txt

git commit

```
* 95bd7b0 (HEAD -> feature/planet) Merge branch 'feature/planet_part2'
into feature/planet, resolve conflicts in planet.txt
\
* f315c73 (feature/planet_part2) add Jupiter
* 9a0c7d4 add Uranus
* | 5183b3f (feature/planet_part1) add Mercury
|/
* a7e5c8e add venus
|
* bae9486 (feature/star) change to Sol
|
* cdc5f89 add vega
|/
* e717155 (tag: v1.0, main) setup project
```

Erwartete Konflikt betrifft zwei disjunkte Dateien: planet.txt, star.txt  
Git: Das kann ich beheben, ich werfe die beiden Dateien zusammen

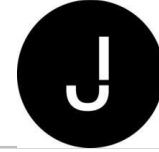
```
* c240d3f (HEAD -> feature/planet) Merge branch 'feature/star' into feature/planet
|
| * bae9486 (feature/star) change to Sol
| * cdc5f89 add vega
| * | 95bd7b0 Merge branch 'feature/planet_part2' into feature/planet,
| resolve conflicts in planet.txt
| |
| | * | f315c73 (feature/planet_part2) add Jupiter
| | * | 9a0c7d4 add Uranus
| | * | 5183b3f (feature/planet_part1) add Mercury
| |
| | * / a7e5c8e add venus
| |
| * e717155 (tag: v1.0, main) setup project
```

# git checkout main

## git merge feature/planet

Das muss ein Fast Forward Merge sein!

```
* c240d3f (HEAD -> main, feature/planet) Merge branch 'feature/star'
into feature/planet
\
* bae9486 (feature/star) change to Sol
* cdc5f89 add vega
* | 95bd7b0 Merge branch 'feature/planet_part2' into feature/planet,
resolve conflicts in planet.txt
| \
| * | f315c73 (feature/planet_part2) add Jupiter
| * | 9a0c7d4 add Uranus
* | | 5183b3f (feature/planet_part1) add Mercury
| \
* / a7e5c8e add venus
| \
* e717155 (tag: v1.0) setup project
```



- Verhindert einen potenziell möglichen Fast Forward
- Statt dessen: Neues Commit-Objekt mit Committer, Timestamp, Message, der den merge dokumentiert

- Stars und Planets mit no-ff mergen
- Lösung

```
* 5ca9b68 (HEAD -> main) Merge branch 'feature/planet'
* 2474852 (feature/planet) Merge branch 'feature/star' into feature/planet
* c72e50d (feature/star) change to sol
* 5645a17 add vega
* 1884021 Merge branch 'feature/planet_part2' into feature/planet, resolve conflict
* 790a1dd (feature/planet_part2) add Jupiter
* e9380cb add Uranus
* 93385ab Merge branch 'feature/planet_part1' into feature/planet
* 90a72da (feature/planet_part1) add Mercury
* fd59223 add venus
* 26277fd (tag: v1.0) setup project
```



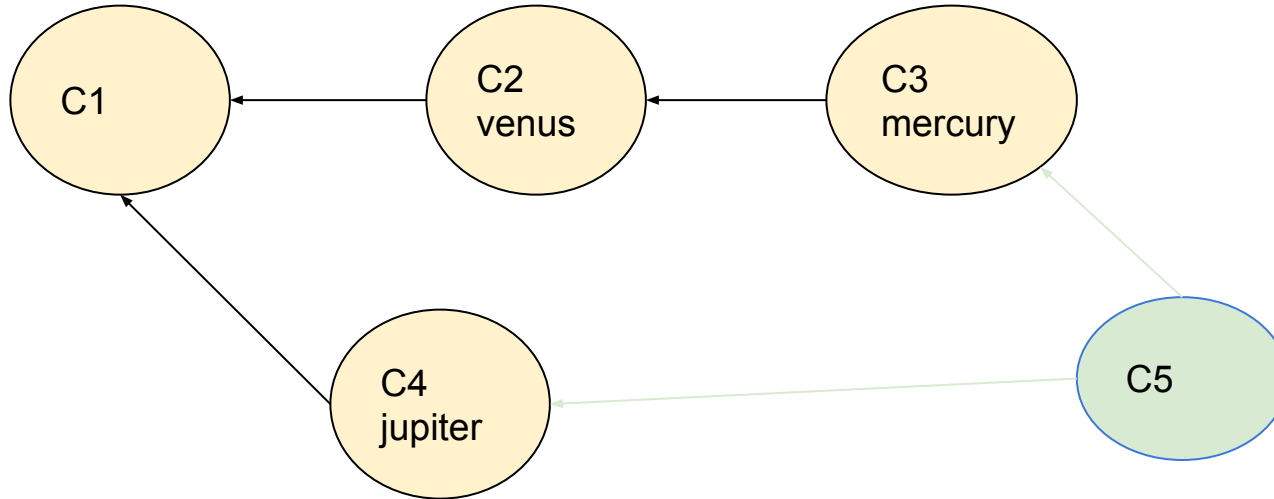
```
* 5ca9b68 (HEAD -> main, tag: v1.1-Milestone1) Merge branch 'feature/planet'
* 2474852 Merge branch 'feature/star' into feature/planet
* c72e50d change to sol
* 5645a17 add vega
* 1884021 Merge branch 'feature/planet_part2' into feature/planet, resolve conflict
* 790a1dd add Jupiter
* e9380cb add Uranus
* 93385ab Merge branch 'feature/planet_part1' into feature/planet
* 90a72da add Mercury
* fd59223 add venus
* 26277fd (tag: v1.0) setup project
```



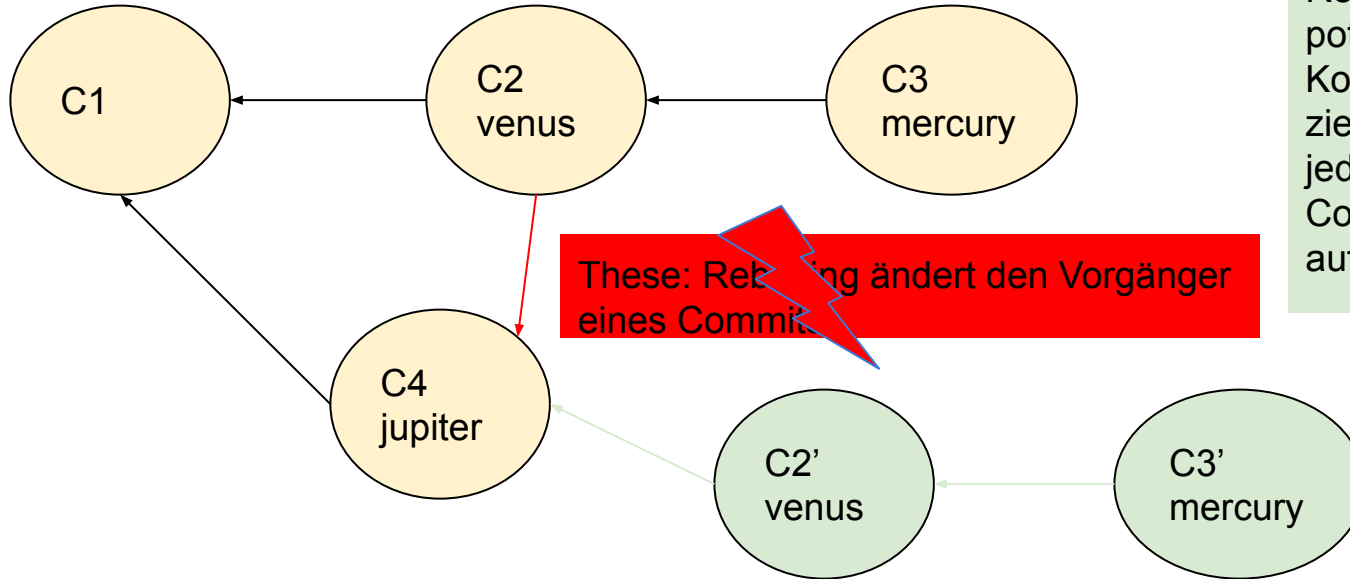
- Auftreten “unlösbarer” Konflikte
  - `git merge --no-ff branch_X`
  - `git merge --abort`
- “Irrtümliches” merge
  - Beispiel: star und planet waren doch gar nicht fertig...
  - Neuerzeugung der feature-Branches
  - Löschen und Erzeugen des main-Branches auf der ursprünglichen Position
    - `git checkout v1.0`
    - `git branch -D main`
      - -D: force delete, -d führt zu Fehler, da git korrekterweise davon ausgeht, dass unreachable commits entstehen
    - `git checkout -b main`
  - Hinweis:
    - Damit verschwinden Commit-Objekte aus dem git log
    - `git fsck --unreachable --no-reflog`

- Der main-Branch (= aktuelle Projekt-Stand) soll beide neuen Features enthalten
- Die Commit-Historie soll alle relevanten durchgeführten Aktionen übersichtlich, sequentiell enthalten

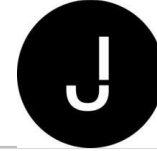
# Konsolidieren von Ständen mit Merge



Merge mit  
potenziellen  
Konflikten, die beim  
Mergen alle im  
Workspace auftreten



Rebase mit potenziellen Konflikten, die potenziell mehrfach (bei jedem neuen Commit-Objekt) auftreten können

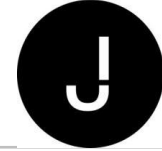


```
* f2b7731 (feature/planet_part2) add Jupiter
* 5005211 add Uranus
| * 3ce9b49 (feature/planet_part1) add Mercury
|/
* 3036bc3 (feature/planet) add venus
| * fd4db0c (feature/star) change to Sol
| * 75f4a49 add vega
|/
* fe02091 (HEAD -> main, tag: v1.0) setup project
```



- “Ich tue so, als hätte ich die beiden Parts von planet nacheinander gemacht”
  - rebase von part1 auf planet und part2 auf planet
- “Ich tue so, als hätte ich das feature/star nach planet gemacht”
  - rebase von star auf planet
- Vorziehen des Main durch ff auf planet

# git checkout feature/planet\_part2 git git rebase feature/planet\_part1



**JAVACREAM**

Training  
Consulting  
Projectmanagement

1. Konflikt:
  - a. Uranus auf Mercury
  - b. vi planet.txt, git add ., git commit, git rebase --continue
2. Konflikt
  - a. Jupiter auf Mercury Uranus
  - b. vi planet.txt, git add ., git commit, git rebase --continue

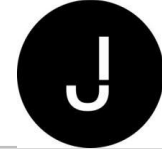


- `git checkout main`
- `git merge feature/star`
  - das ist ein beabsichtigter Fast Forward
-



- wie bei merge: Entfernen der abgeschlossenen Feature-Banches, Tag des erreichten Milestones

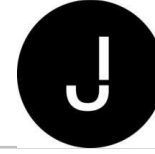
```
* a2503d1 (HEAD -> main, tag: v1.1-Milestone1) change to sol
* 40c98d1 add vega
* b35605f add Jupiter
* a009b65 add Uranus
* 3ce9b49 add Mercury
* 3036bc3 add venus
* fe02091 (tag: v1.0) setup project
```



- Git motiviert zu häufigen Commits
  - Zeitskala eher wenige Stunden
- Commit Messages
  - Verpflichtend
  - sind Bestandteil der Dokumentation des Projektfortschritts
  - Vorgaben für gute Commit-Messages sind vorhanden
    - Zeile 1: Kurzbeschreibung, “was wird passieren”
    - Abstract
    - Langbeschreibung
    - Optional: Liste der Dateien

- Aus einer Menge von eher “minderwertigen” Commits sollen einige (wenige) hochwertige Commits zusammengefasst werden
  - Bei und
    - implement feature planet introducing planet.txt
    - implement feature star introducing star.txt
- Ablauf eines interactive rebase
  - `git rebase --interactive ...`
    - Ergebnis ist ein Editor-Fenster mit einem rebase-Script
    - Nach Schließen des Scripts
      - Commit Messages für die verbliebenen Commits
  - Konkret in unserem Beispiel
    - `git checkout main`
    - `git rebase -i v1.0`

```
* 6641a4c (HEAD -> main, tag: v1.1-Milestone1) implement feature star, add star.txt
* 4704146 implement feature planet, add planet.txt
* a2503d1 (tag: finish/star_planet) change to Sol
* 40c98d1 add vega
* b35605f add Jupiter
* a009b65 add Uranus
* 3ce9b49 add Mercury
* 3036bc3 add venus
* fe02091 (tag: v1.0) setup project
```



## GitHub Enterprise

- Git Server
  - Verwaltet eine Menge von Git-Repositories mit Benutzerverwaltung
- Web Frontend
  - Dateiexplorer
  - Übersicht der Branches und Tags
- Tool zur Aufgaben-Verwaltung -> Jira
- Erstellung einer WIKI-basierten Dokumentation -> Confluence
- Continuous Integration / Deployment|Delivery ("CI/CD") -> Jenkins
- Keine Entwicklungsumgebung

# Erstellen eines Repositories in GitHub

- Verweis
  - FI-Richtlinien und Vorgaben zu finden auf den GitHub Confluence Seiten
- Neues Repository anlegen
  - Web Frontend...
- Zugriff aus der Git Bash erfolgt im Seminar über ein sogenannten Personal Access Token

Signed in as j716160

Set status

Your profile

Your repositories

Your enterprise

Your projects

Your stars

Your gists

Help

Settings

Sign out

Security log

Developer settings

Personal access tokens

Generate new token

Note

training\_27.7.2023

What's this token for?

Expiration \*

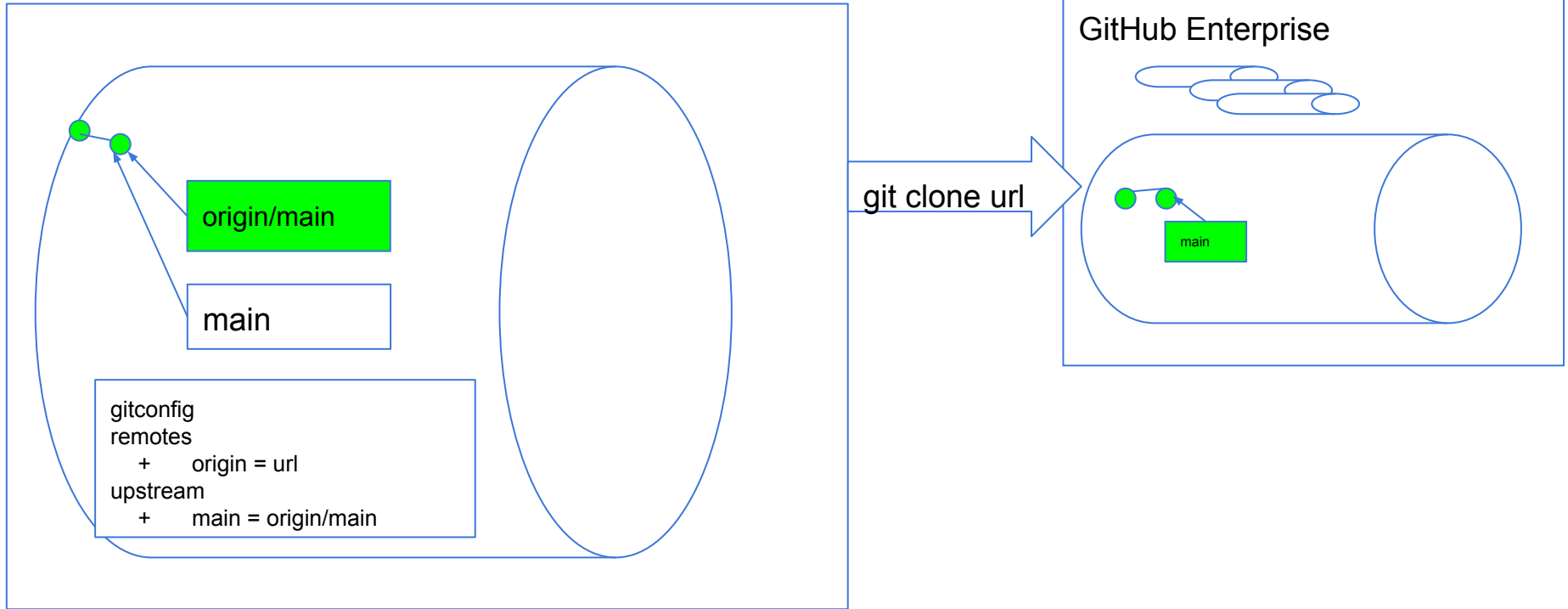
30 days The token will expire on Sun, Aug 27 2023

Select scopes

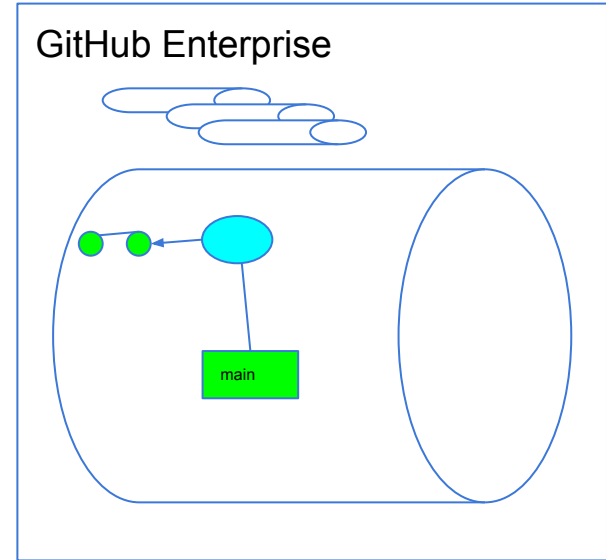
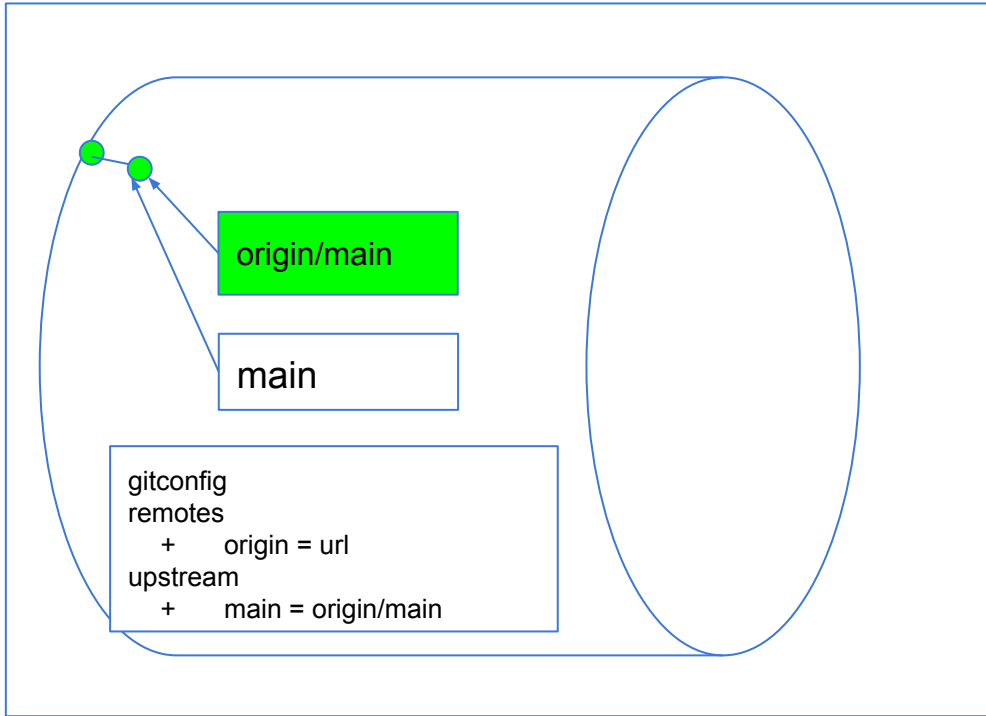
Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

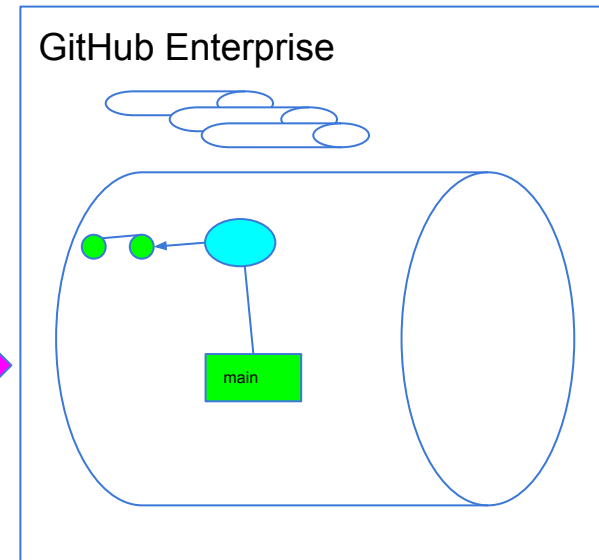
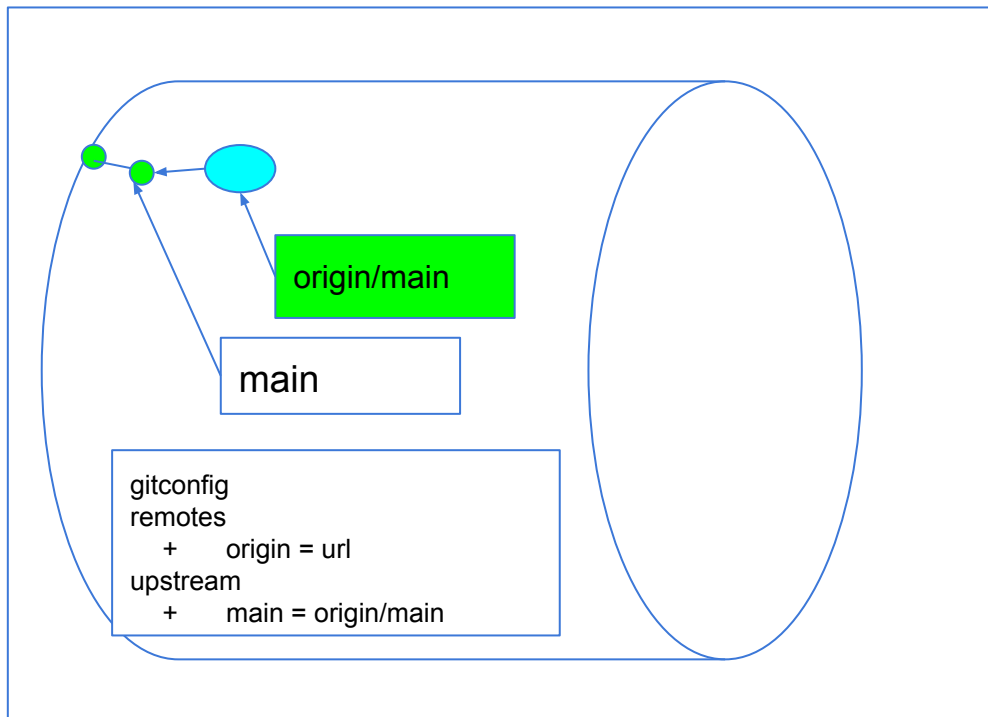
<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events

# Clone eines Server-Repositories





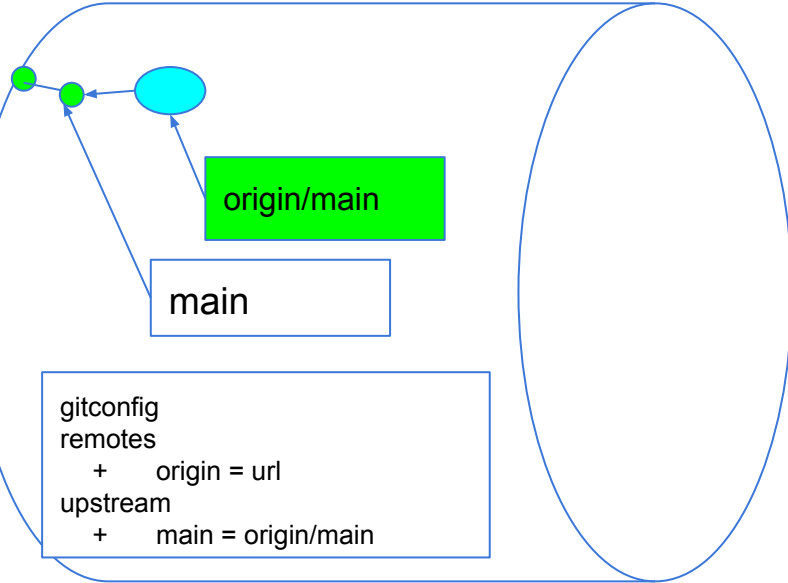






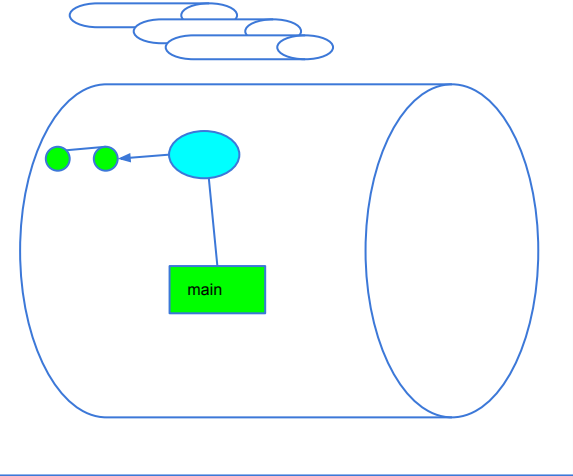
- Ein Remote Branch zeigt immer auf ein Commit-Geflecht, das zum Zeitpunkt des clone/fetch auf Server-Seite vorhanden ist
- Ein Remote Branch ist aus Sicht des Clone-Repositories unveränderlich / read only
  - checkout eines remote branches führt zu Detached HEAD

# Nach dem Fetch: Up to you...



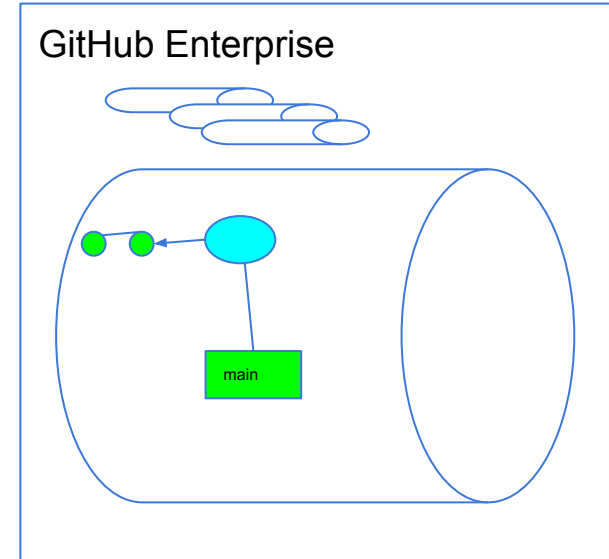
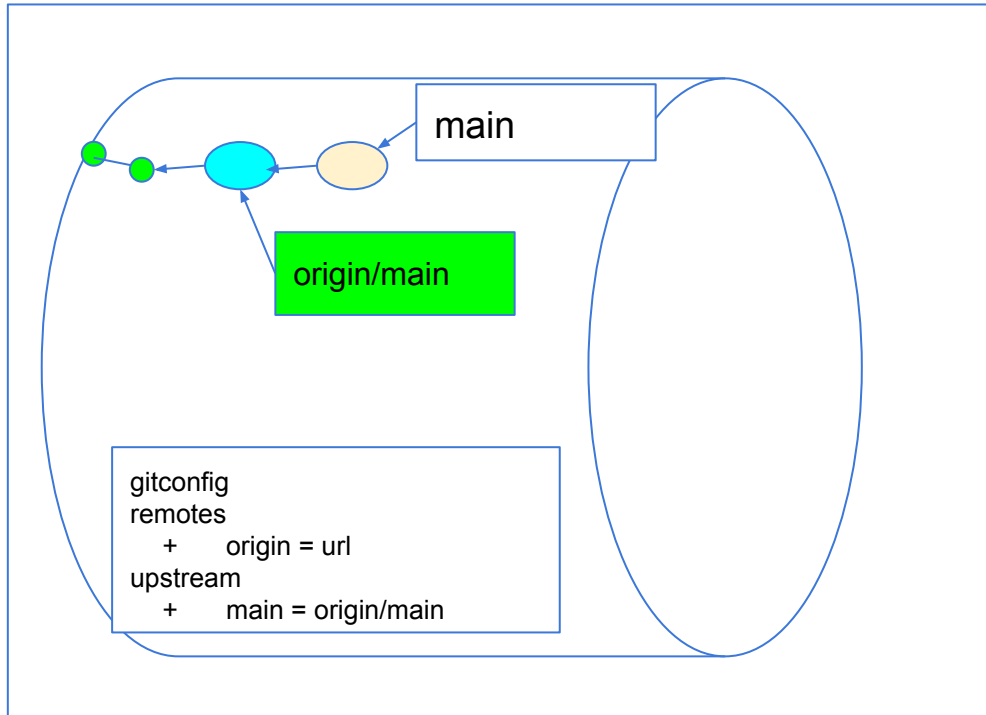
merge  
rebase

## GitHub Enterprise

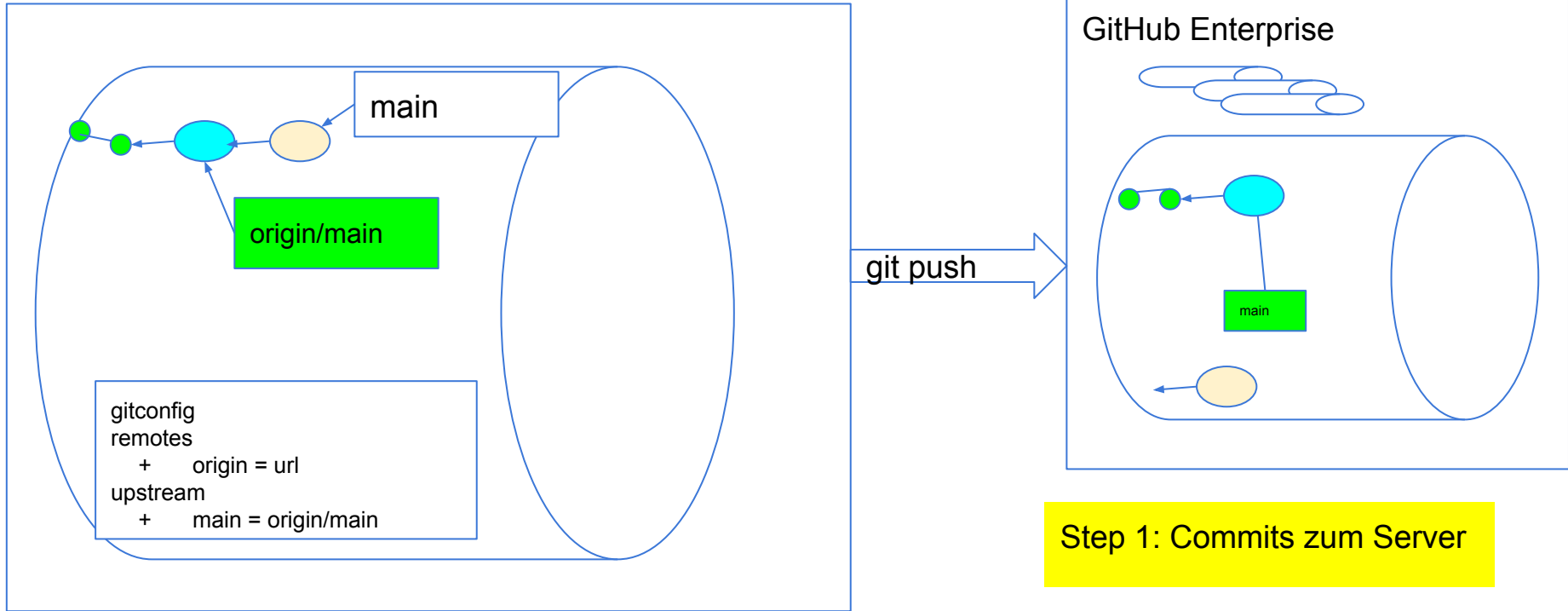


- Ein Kombi-Befehl
- `git pull`
  - `git fetch`
  - `git merge`
    - Da können natürlich Konflikte auftreten...
- `git pull --rebase`
  - `git fetch`
  - `git rebase`
    - Da können natürlich Konflikte auftreten...

# Änderung im Clone

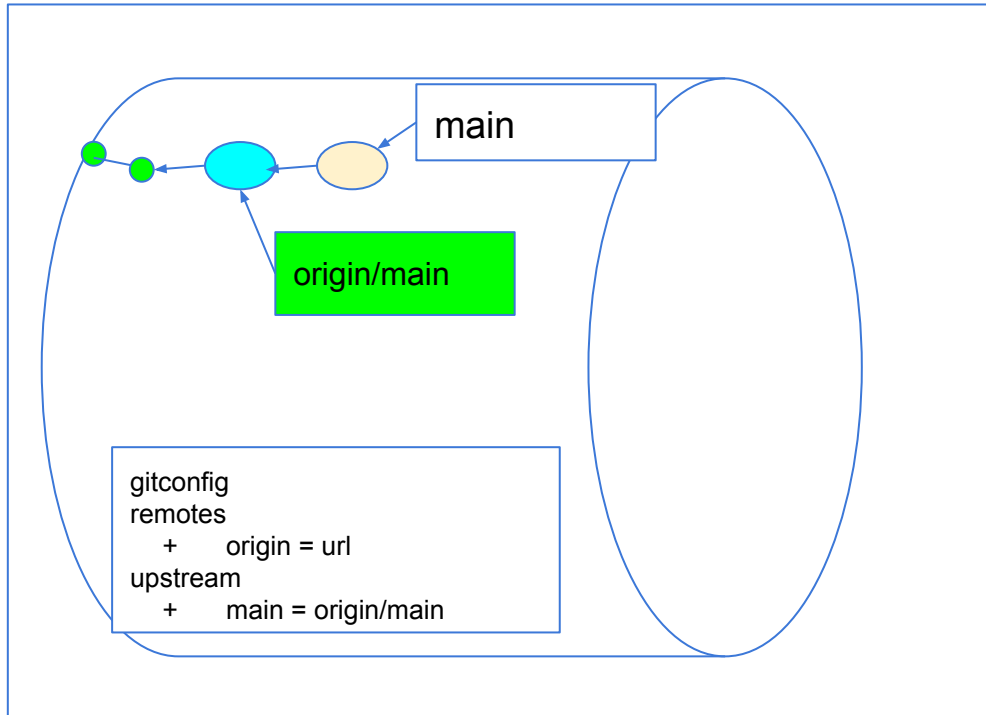


# Pushen der Commits zum Server



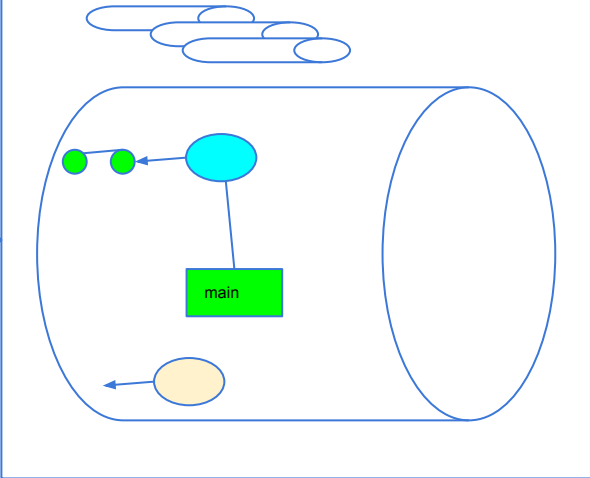
Step 1: Commits zum Server

# Pushen der Commits zum Server



git push

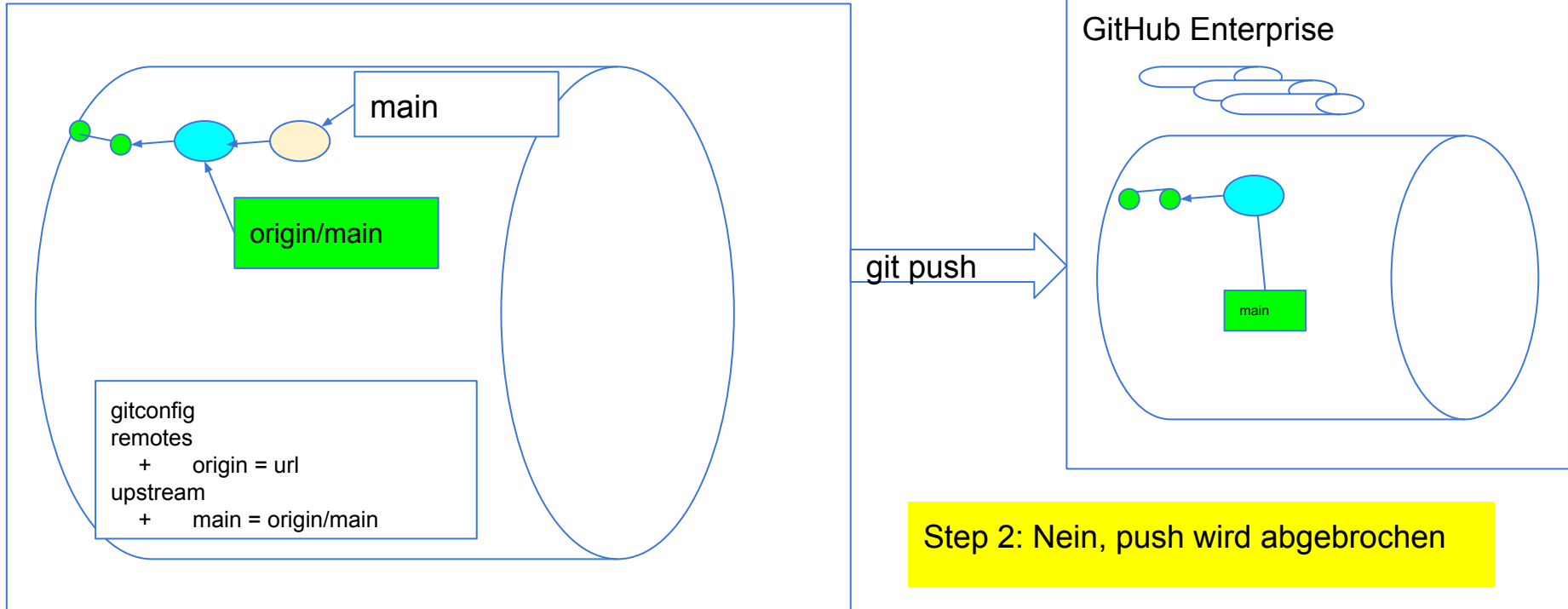
## GitHub Enterprise



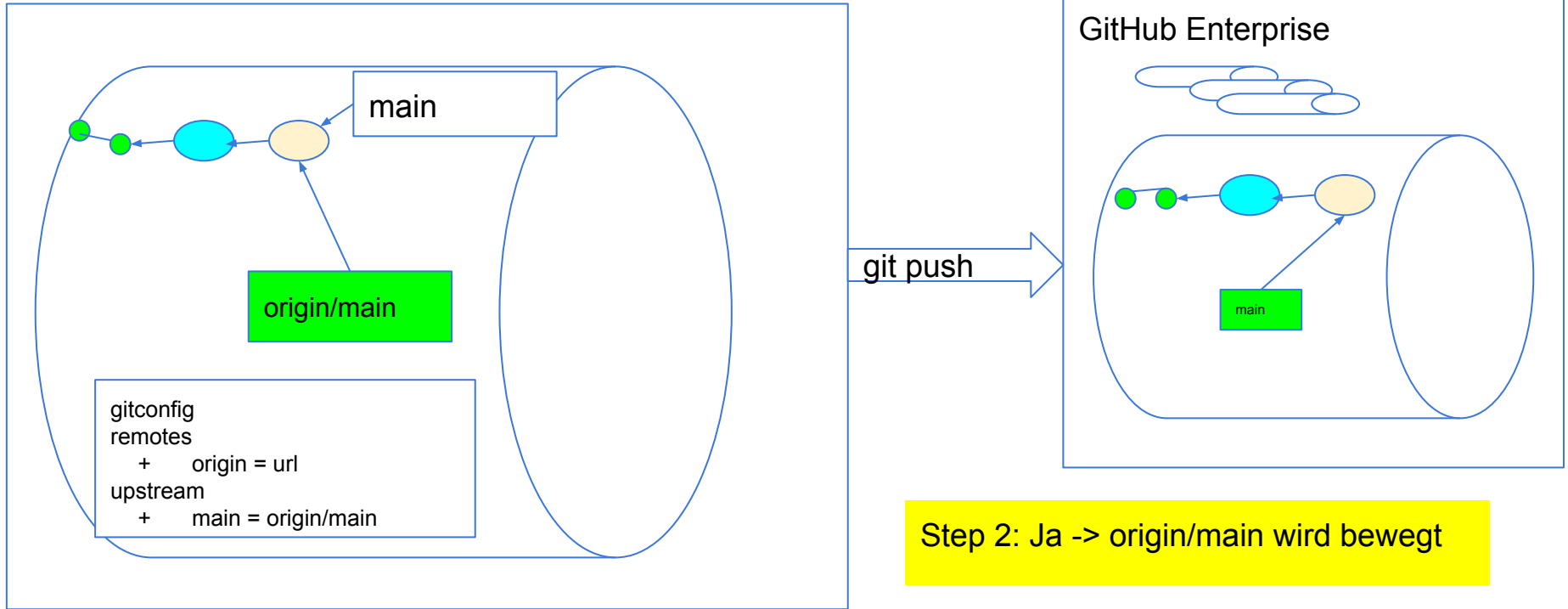
Step 2: Check, ist ein Konfliktfreies  
mergen möglich? -> Fast Forward



# Pushen der Commits zum Server

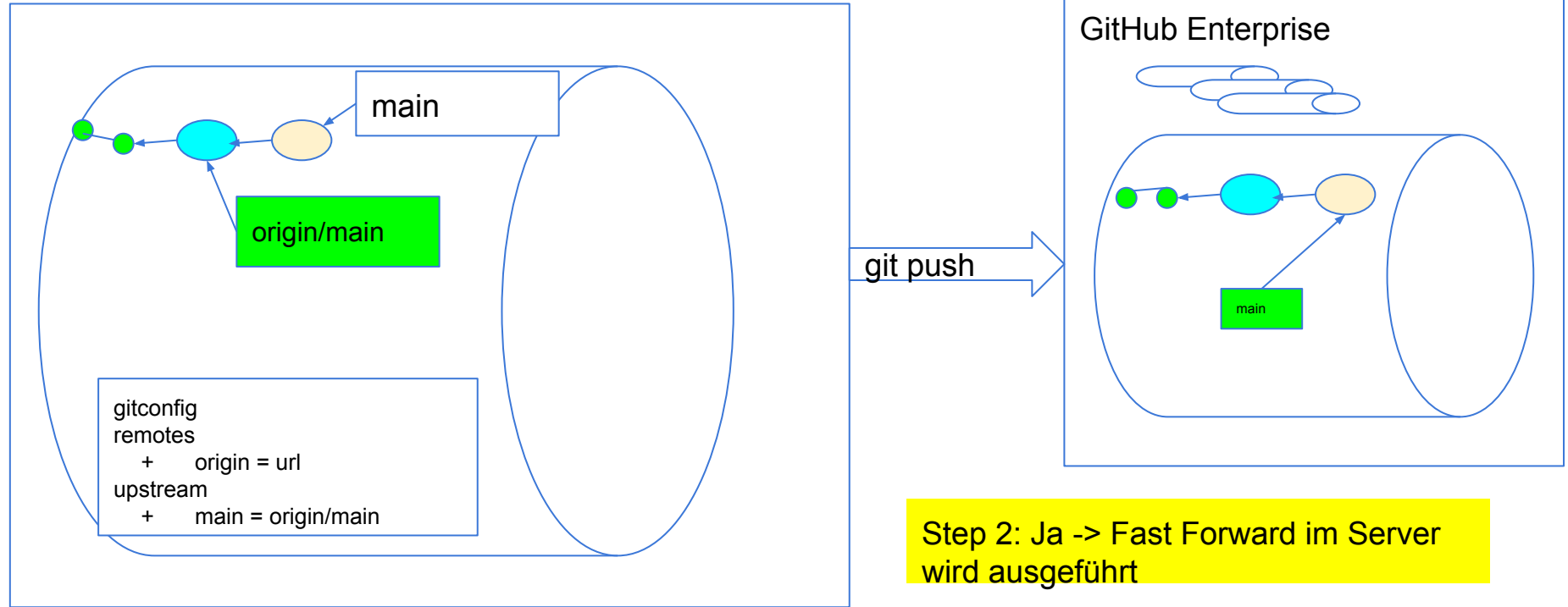


# Pushen der Commits zum Server



Step 2: Ja -> origin/main wird bewegt

# Pushen der Commits zum Server



Step 2: Ja -> Fast Forward im Server wird ausgeführt