Die „goldene Regel" des Rebasings □□

Die **goldene Regel des Rebasings** lässt sich in einem Satz zusammenfassen:

> *„Schreibe niemals die Historie von Commits um, die bereits gepusht und mit anderen geteilt wurden."*

Diese Regel ist so fundamental, dass sie in der Git-Community fast schon als Gesetz gilt. Um zu verstehen, *warum* diese Regel so wichtig ist und was passiert, wenn man sie bricht, müssen wir zunächst verstehen, was beim Rebase technisch passiert.

Was passiert beim Rebase technisch?

Wenn du `git rebase` ausführst, **erstellt Git neue Commits** – auch wenn der Inhalt identisch bleibt. Das liegt daran, wie Git-Commits aufgebaut sind:

Ein Commit ist durch seinen **SHA-1-Hash** eindeutig identifiziert, und dieser Hash wird berechnet aus:

- dem Inhalt aller Dateien (Tree)
- der Commit-Message
- dem Autor und Zeitstempel
- dem **Parent-Commit** (Vorgänger)

Beim Rebase änderst du den Parent-Commit – deine Commits bekommen einen neuen „Vorgänger". Selbst wenn sich sonst nichts ändert, erzeugt das einen **komplett neuen Hash**. Für Git sind das völlig neue Commits!

```
flowchart TD
    subgraph vorher["Vor dem Rebase"]
```

```
        A1["Commit A\nHash: abc123"] --> B1["Commit B\nHash: def456"]
        B1 --> C1["Dein Commit X\nHash: 111aaa\nParent: def456"]
        C1 --> D1["Dein Commit Y\nHash: 222bbb\nParent: 111aaa"]
    end

    subgraph nachher["Nach dem Rebase"]
        A2["Commit A\nHash: abc123"] --> B2["Commit B\nHash: def456"]
        B2 --> E2["Commit C - NEU\nHash: ghi789"]
        E2 --> C2["Dein Commit X - NEU\nHash: 333ccc\nParent: ghi789"]
        C2 --> D2["Dein Commit Y - NEU\nHash: 444ddd\nParent: 333ccc"]
    end

    vorher -.->|"git rebase"| nachher

    style C1 fill:#ffcccc,color:#000000
    style D1 fill:#ffcccc,color:#000000
    style C2 fill:#ccffcc,color:#000000
    style D2 fill:#ccffcc,color:#000000
```

Die roten Commits (alte Hashes) existieren nach dem Rebase nicht mehr in deinem Branch – stattdessen gibt es die grünen Commits mit **neuen Hashes**. Für Git sind das völlig unterschiedliche Commits, auch wenn sie inhaltlich identisch sein können.

Warum ist das Umschreiben öffentlicher Historie so problematisch?

1. **Divergierende Historien im Team** □□

Stell dir folgendes Szenario vor:

1. Du hast Commits A → B → C gepusht
2. Dein Kollege hat diese Commits bereits gepullt
3. Du machst einen Rebase und pusht mit `--force`
4. Jetzt hat dein Kollege A → B → C, aber auf dem Server liegt A → B' → C'

Für Git sind das **zwei völlig unterschiedliche Branches**, die zufällig einen gemeinsamen Vorfahren haben. Wenn dein Kollege jetzt pullt oder pusht, passiert Chaos:

```
flowchart TD
    subgraph server["GitHub - nach deinem Force-Push"]
        S1["A"] --> S2["B'"] --> S3["C'"]
    end

    subgraph kollege["Dein Kollege - lokales Repo"]
        K1["A"] --> K2["B"] --> K3["C"] --> K4["D - seine neue Arbeit"]
    end

    subgraph problem["Das Problem beim Pull/Push"]
        P1["Git sieht zwei divergierende\nBranches mit gleichem Namen"]
        P2["Merge-Konflikte oder\nduplizierte Commits"]
        P1 --> P2
    end

    server -.->|"git pull"| problem
    kollege -.->|"betrifft"| problem

    style S2 fill:#ccffcc,color:#000000
    style S3 fill:#ccffcc,color:#000000
    style K2 fill:#ffcccc,color:#000000
    style K3 fill:#ffcccc,color:#000000
    style P2 fill:#ffdddd,color:#000000
```

2. **Duplizierte Commits** □□

Wenn dein Kollege nach deinem Force-Push versucht, seine Arbeit zu synchronisieren, kann es passieren, dass dieselben Änderungen **zweimal in der Historie** auftauchen – einmal mit dem alten Hash, einmal mit dem neuen. Die Historie wird unverständlich und aufgebläht.

3. **Verlorene Arbeit** □□

Noch schlimmer: Wenn jemand auf den „alten" Commits aufgebaut hat und dann deine neue Historie überschreibt, kann dessen Arbeit **komplett verloren gehen** – oder es entstehen massive Merge-Konflikte, die sehr schwer aufzulösen sind.

4. **Kaputte Referenzen überall** □□

Commits werden an vielen Stellen referenziert:

- **Pull Requests** und Code Reviews verweisen auf spezifische Commit-Hashes
- **Issue-Tracker** und Commit-Messages referenzieren andere Commits
- **CI/CD-Pipelines** haben Build-Ergebnisse für bestimmte Commits gespeichert
- **Deployment-Logs** dokumentieren, welcher Commit deployed wurde
- **Dokumentation** und **Changelogs** können Commit-Hashes enthalten

Nach einem Force-Push zeigen all diese Referenzen ins Leere – die Commits mit diesen Hashes existieren nicht mehr (oder nur noch im Reflog einzelner Entwickler).

5. **Vertrauensverlust in die Historie** □□

Die Git-Historie ist nicht nur ein technisches Artefakt – sie ist auch **Dokumentation**. Teams verlassen sich darauf, dass die Historie korrekt widerspiegelt, was passiert ist. Wenn regelmäßig umgeschrieben wird, verliert die Historie ihren Wert als verlässliche Quelle der Wahrheit.

Was genau bedeutet „öffentlich" oder „geteilt"?

Die Frage ist nicht, ob ein Commit *gepusht* wurde, sondern ob **andere darauf aufbauen könnten**:

Situation	Rebase erlaubt?	Begründung
Lokale Commits, nie gepusht	□ Ja	Niemand kennt diese Commits
Gepusht auf *deinen eigenen* Feature-Branch, noch kein PR	⚠ Mit Vorsicht	Wenn du sicher bist, dass niemand anders den Branch gepullt hat
Feature-Branch mit offenem PR	□ Besser nicht	Reviewer haben möglicherweise den Branch ausgecheckt
Feature-Branch, der bereits gemerged wurde	□ Nein	Die Commits sind jetzt Teil der Haupthistorie
`main`, `develop` oder andere geteilte Branches	□ Niemals	Hier baut das ganze Team drauf auf

---

Die Ausnahme: Force-Push auf eigene Feature-Branches

Es gibt **eine legitime Situation** für Force-Push, die in vielen Teams akzeptiert oder sogar erwünscht ist:

> **Das Aufräumen deines eigenen Feature-Branches *vor* dem Merge.**

Viele Teams praktizieren folgenden Workflow:

1. Du arbeitest auf einem Feature-Branch und machst viele kleine „WIP"-Commits
2. Du pushst regelmäßig als Backup
3. **Vor dem finalen Review** räumst du auf: Commits zusammenfassen, Messages verbessern
4. Du machst einen Force-Push auf *deinen* Feature-Branch
5. Dann erst wird der PR final reviewed und gemerged

Voraussetzungen dafür:

- Der Branch gehört dir allein (niemand anders arbeitet darauf)
- Der PR wurde noch nicht approved/gemerged
- Das Team hat sich auf diesen Workflow geeinigt

In PhpStorm findest du die Force-Push-Option über **Git → Push** und dann den Haken bei **Force Push** (⚠ nur setzen, wenn du dir sicher bist!).

Was tun, wenn du versehentlich öffentliche Historie umgeschrieben hast? □□

Okay, es ist passiert. Du hast einen Force-Push auf einen Branch gemacht, auf dem andere aufbauen. Keine Panik – es gibt Lösungsansätze, aber handle schnell!

Sofortmaßnahme: Team informieren □□

Kommuniziere sofort mit deinem Team über Slack, Teams, oder wie auch immer ihr kommuniziert:

> „⚠ Achtung: Ich habe versehentlich einen Force-Push auf [branch-name] gemacht. Bitte noch nicht pullen/pushen, bis wir das geklärt haben!"

Je schneller du kommunizierst, desto weniger Leute sind betroffen.

Option 1: Alten Zustand wiederherstellen (beste Option, wenn möglich)

Wenn noch niemand (oder fast niemand) die neue Historie gepullt hat, ist die sauberste Lösung, den **alten Zustand wiederherzustellen**:

1. **Finde den alten Commit-Hash** – Der alte Zustand existiert noch im Reflog (lokal) oder vielleicht bei einem Kollegen:

   ```
   # In deinem lokalen Repository:
   git reflog show origin/main
   ```

   Du siehst eine Liste wie:

   ```
   abc1234 origin/main@{0}: update by push --force
   def5678 origin/main@{1}: update by push
   ```

   Der Hash `def5678` ist der Zustand *vor* deinem Force-Push.
2. **Setze den Branch zurück** auf den alten Zustand:

   ```
   git push --force origin def5678:main
   ```

   Damit überschreibst du den Branch wieder mit dem ursprünglichen Zustand.
3. **In PhpStorm:**
   - Öffne **Git → Show Git Log**
   - Aktiviere **Show All Branches** und suche im Reflog
   - Rechtsklick auf den gewünschten Commit → **Reset Current Branch to Here**
   - Dann **Git → Push** mit Force-Push-Option

Option 2: Kollegen haben bereits gepullt – Koordiniertes Recovery

Wenn einige Teammitglieder bereits die neue Historie gepullt haben, wird es komplizierter. Hier ist ein koordinierter Ansatz:

1. **Einigt euch auf den „korrekten" Zustand** – Welche Version soll gelten? Die alte oder die neue?
2. **Wenn die alte Version gilt:**
   - Du stellst den alten Zustand wieder her (siehe Option 1)
   - Jeder, der die neue Version hat, muss seinen lokalen Branch zurücksetzen:

     ```
     git fetch origin
     git reset --hard origin/main
     ```

   - **Achtung:** Lokale Änderungen, die auf der neuen Historie aufbauen, müssen vorher gesichert werden!
3. **Wenn die neue Version gilt:**
   - Die neue Historie bleibt auf dem Server
   - Jeder muss seinen lokalen Branch an die neue Historie anpassen:

     ```
     git fetch origin
     git reset --hard origin/main
     ```

   - Wer eigene Commits auf der alten Historie gemacht hat, muss diese **cherry-picken** oder **rebasen**

Option 3: Der „Merge-Workaround" für komplizierte Situationen

Wenn das Chaos zu groß ist und ein Reset nicht praktikabel, gibt es einen pragmatischen Workaround:

1. Erstelle einen Branch vom alten Zustand (aus dem Reflog eines Kollegen)
2. Merge diesen Branch in den aktuellen Zustand
3. Damit sind beide Historien zusammengeführt

Das Ergebnis ist keine saubere Historie, aber alle Änderungen sind erhalten und jeder kann normal weiterarbeiten.

Option 4: Arbeit von betroffenen Kollegen retten

Wenn ein Kollege bereits neue Commits auf der alten Historie gemacht hat, müssen diese gerettet werden:

```
flowchart TD
    subgraph problem["Ausgangssituation"]
        A1["origin/main - neue Historie\nA → B' → C'"]
        A2["Kollege lokal\nA → B → C → D → E"]
        A3["D und E sind seine neue Arbeit\nauf der alten Historie"]
    end

    subgraph loesung["Lösung"]
        B1["1. Kollege notiert sich die Hashes\nvon D und E"]
        B2["2. git fetch origin"]
        B3["3. git reset --hard origin/main"]
        B4["4. git cherry-pick D-hash E-hash"]
        B1 --> B2 --> B3 --> B4
    end

    problem --> loesung

    style A3 fill:#ffdddd,color:#000000
    style B4 fill:#ccffcc,color:#000000
```

Der Kollege führt aus:

```
# 1. Hashes der eigenen Commits notieren
git log --oneline  # z.B. abc1234 (E) und def5678 (D)

# 2. Aktuelle Remote-Historie holen
git fetch origin

# 3. Auf die neue Historie wechseln (ACHTUNG: lokale Änderungen gehen verloren!)
git reset --hard origin/main

# 4. Eigene Commits wieder anwenden
git cherry-pick def5678 abc1234  # D und E
```

In PhpStorm:

1. **Git → Fetch** ausführen
2. Im Git-Log die eigenen Commits finden und deren Hashes notieren
3. Rechtsklick auf `origin/main` → **Reset Current Branch to Here** (Hard)
4. Im Git-Log die notierten Commits finden → Rechtsklick → **Cherry-Pick**

Präventivmaßnahmen: Wie vermeide ich diesen Fehler in Zukunft? □□

1. GitHub Branch Protection aktivieren

Für wichtige Branches wie `main` oder `develop` solltest du **Force-Pushes verbieten**:

1. Gehe zu **Settings → Branches → Add rule**
2. Branch name pattern: `main`
3. Aktiviere: □ **Require a pull request before merging**
4. Aktiviere: □ **Do not allow force pushes** (unter „Rules applied to everyone")

Damit kann *niemand* (auch keine Admins) einen Force-Push auf diesen Branch machen.

2. Git-Konfiguration anpassen

Du kannst Git so konfigurieren, dass Force-Pushes schwieriger werden:

```
# Statt --force nutze --force-with-lease
# Dieser Befehl schlägt fehl, wenn jemand anders in der Zwischenzeit gepusht hat
git config --global push.default simple
git config --global alias.pushf "push --force-with-lease"
```

`--force-with-lease` ist eine „sicherere" Version von `--force`: Sie prüft, ob der Remote-Branch noch dem entspricht, was du erwartest. Wenn jemand in der Zwischenzeit gepusht hat, schlägt der Push fehl.

3. Lokale Git-Hooks nutzen

Du kannst einen **pre-push Hook** einrichten, der dich warnt:

Erstelle die Datei `.git/hooks/pre-push`:

```
#!/bin/bash

# Warnung bei Force-Push auf geschützte Branches
protected_branches="main master develop"

for branch in $protected_branches; do
    if echo "$@" | grep -q "$branch"; then
        echo "⚠  WARNUNG: Du versuchst auf den geschützten Branch '$branch' zu pushen!"
        echo "   Bist du sicher? (y/n)"
        read -r answer
        if [ "$answer" != "y" ]; then
            echo "Push abgebrochen."
            exit 1
        fi
    fi
done
```

4. Workflow-Regel etablieren

Etabliere im Team eine klare Regel:

> **„Rebase nur für lokale Commits oder eigene, noch nicht gemergte Feature-Branches."**

Für alles andere: **Merge verwenden.**

Zusammenfassung: Die wichtigsten Punkte

Aspekt	Empfehlung
Goldene Regel	Niemals gepushte/geteilte Historie umschreiben
Lokale Commits	Rebase erlaubt und empfohlen für saubere Historie
Eigene Feature-Branches	Mit Vorsicht, wenn niemand anders darauf arbeitet
Geteilte Branches	Niemals rebasen – immer mergen
Wenn es passiert ist	Schnell kommunizieren, alten Zustand aus Reflog wiederherstellen
Prävention	Branch Protection, `--force-with-lease`, Team-Kommunikation

Die goldene Regel schützt nicht nur dein Team vor Chaos, sondern auch dich selbst: Nichts ist frustrierender, als Stunden damit zu verbringen, ein Repository-Chaos zu entwirren, das durch einen unbedachten Force-Push entstanden ist. **Im Zweifel: Merge statt Rebase!** □□

Revision #1
Created 2026-06-29 15:55:04 UTC by art10m
Updated 2026-06-29 15:55:22 UTC by art10m