Kapitel 3: Rebase meistern – Geschichte umschreiben wie ein Profi

Einleitung: `git rebase` ist einer der mächtigsten und gleichzeitig gefürchtetsten Befehle in Git. Richtig eingesetzt ermöglicht er dir eine saubere, lineare Commit-Historie ohne unnötige Merge-Commits. Mit Interactive Rebase kannst du sogar die Geschichte umschreiben: Commits zusammenfassen, aufteilen, umbenennen oder neu ordnen. Aber Vorsicht: Mit großer Macht kommt große Verantwortung! Das Umschreiben von bereits gepushter Historie kann zu ernsthaften Problemen führen. Dieses Kapitel lehrt dich, Rebase sicher und effektiv einzusetzen.

- Merge vs. Rebase – Der fundamentale Unterschied
- Interactive Rebase meistern – Commits aufräumen wie ein Profi □□
- Commits zusammenfassen (Squash) in PhpStorm □□
- Die „goldene Regel" des Rebasings □□

Merge vs. Rebase – Der fundamentale Unterschied

Der Unterschied zwischen `git merge` und `git rebase` ist eines der wichtigsten Konzepte, das du als fortgeschrittener Git-Nutzer wirklich verstanden haben solltest. Beide Befehle dienen demselben Zweck – **Änderungen aus einem Branch in einen anderen zu integrieren** – aber sie tun dies auf völlig unterschiedliche Weise, mit unterschiedlichen Auswirkungen auf deine Commit-Historie.

Die Ausgangssituation

Stellen wir uns folgendes Szenario vor: Du arbeitest an einem Feature-Branch, während dein Team weiterhin Commits auf `main` macht. Nach einiger Zeit sieht deine Repository-Struktur so aus:

```
gitGraph
    commit id: "A"
    commit id: "B"
    branch feature
    commit id: "X"
    commit id: "Y"
    checkout main
    commit id: "C"
    commit id: "D"
```

Du hast auf deinem `feature`-Branch die Commits **X** und **Y** erstellt, während auf `main` in der Zwischenzeit **C** und **D** hinzugekommen sind. Jetzt möchtest du die Änderungen von `main` in deinen Feature-Branch integrieren (oder umgekehrt deinen Feature-Branch in `main` einbringen). Hier kommen Merge und Rebase ins Spiel.

Was passiert bei einem Merge?

Ein **Merge** erstellt einen neuen *Merge-Commit*, der die beiden Entwicklungslinien zusammenführt. Dieser Merge-Commit hat **zwei Eltern-Commits** – er verbindet buchstäblich die beiden Branches

miteinander.

Der Merge-Befehl

```
# Du bist auf feature und möchtest main integrieren
git checkout feature
git merge main
```

In PhpStorm: **Git → Merge... → main auswählen**

Das Ergebnis nach dem Merge

```
gitGraph
    commit id: "A"
    commit id: "B"
    branch feature
    commit id: "X"
    commit id: "Y"
    checkout main
    commit id: "C"
    commit id: "D"
    checkout feature
    merge main id: "M" tag: "Merge-Commit"
```

Der neue Commit **M** ist der Merge-Commit. Er enthält keine eigenen Code-Änderungen (außer eventuell Konfliktlösungen), sondern dient als „Knotenpunkt", der die beiden Historien verbindet.

Eigenschaften eines Merge

Aspekt	Beschreibung
Commit-Historie	Bleibt vollständig erhalten, zeigt aber eine verzweigte Struktur
Originalcommits	X und Y behalten ihre ursprünglichen SHA-Hashes
Neuer Commit	Ein Merge-Commit mit zwei Eltern wird erstellt
Nicht-destruktiv	Die bestehende Historie wird niemals verändert

---

Was passiert bei einem Rebase?

Ein **Rebase** nimmt deine Commits und „verpflanzt" sie auf eine neue Basis. Statt eines Merge-Commits werden deine Commits *neu erstellt* – sie werden auf den aktuellen Stand des Ziel-Branches aufgebaut, als hättest du erst jetzt mit der Arbeit begonnen.

Der Rebase-Befehl

```
# Du bist auf feature und möchtest auf main "umbasieren"
git checkout feature
git rebase main
```

In PhpStorm: **Git → Rebase... → main auswählen**

Was genau passiert beim Rebase?

Der Rebase-Prozess läuft in mehreren Schritten ab:

1. **Git identifiziert die Commits**, die nur auf `feature` existieren (X und Y)
2. **Git speichert diese Commits** temporär (als Patches)
3. **Git setzt den `feature`-Branch** auf die Spitze von `main` (Commit D)
4. **Git wendet die gespeicherten Commits** nacheinander wieder an

Das Ergebnis sind *neue Commits* **X'** und **Y'**, die denselben Inhalt haben wie die Originale, aber:

- einen **anderen Eltern-Commit** (sie bauen jetzt auf D auf, nicht auf B)
- einen **anderen SHA-Hash** (weil sich der Eltern-Commit geändert hat)
- einen **anderen Zeitstempel** (optional, je nach Konfiguration)

Das Ergebnis nach dem Rebase

```
gitGraph
    commit id: "A"
    commit id: "B"
    commit id: "C"
    commit id: "D"
    commit id: "X'" tag: "neu erstellt"
    commit id: "Y'" tag: "neu erstellt"
```

Beachte: Die ursprünglichen Commits X und Y existieren technisch noch im Repository (erreichbar über `git reflog`), aber kein Branch zeigt mehr auf sie. Sie werden bei der nächsten Garbage Collection entfernt.

Visueller Vergleich: Merge vs. Rebase

Um den Unterschied noch deutlicher zu machen, hier beide Ergebnisse im direkten Vergleich:

```
flowchart TB
    subgraph vorher["Ausgangssituation"]
        direction LR
        A1["A"] --> B1["B"]
        B1 --> C1["C"] --> D1["D"]
        B1 --> X1["X"] --> Y1["Y"]
    end

    subgraph merge["Nach git merge main"]
        direction LR
        A2["A"] --> B2["B"]
        B2 --> C2["C"] --> D2["D"]
        B2 --> X2["X"] --> Y2["Y"]
        D2 --> M["M"]
        Y2 --> M
    end

    subgraph rebase["Nach git rebase main"]
        direction LR
        A3["A"] --> B3["B"] --> C3["C"] --> D3["D"]
        D3 --> X3["X'"] --> Y3["Y'"]
    end

    vorher --> merge
    vorher --> rebase

    style M fill:#90EE90,color:#000000
```

```
    style X3 fill:#FFD700,color:#000000
    style Y3 fill:#FFD700,color:#000000
```

Die tiefgreifenden Unterschiede im Detail

1. Integrität der Historie vs. Sauberkeit

Merge bewahrt die *exakte historische Wahrheit*: Es zeigt, dass parallel entwickelt wurde und wann die Zusammenführung stattfand. Das kann wertvoll sein, wenn du später nachvollziehen möchtest, wie die Entwicklung tatsächlich ablief.

Rebase erzeugt eine *saubere, lineare Historie*, als hätte es nie parallele Entwicklung gegeben. Das macht die Historie leichter lesbar, „lügt" aber technisch gesehen über den tatsächlichen Entwicklungsverlauf.

2. Commit-Identität

Dieser Punkt ist *entscheidend* für das Verständnis:

```
# Vor dem Rebase
git log --oneline feature
# abc1234 Y - Feature fertiggestellt
# def5678 X - Feature begonnen

# Nach dem Rebase
git log --oneline feature
# 111aaaa Y' - Feature fertiggestellt  ← NEUER HASH!
# 222bbbb X' - Feature begonnen        ← NEUER HASH!
```

Die Commits nach dem Rebase sind **komplett neue Git-Objekte**. Sie haben:

- denselben Autor
- dieselbe Commit-Message
- dieselben Code-Änderungen (Diffs)
- aber einen anderen SHA-Hash (weil der Parent-Commit anders ist)

3. Konfliktbehandlung

Bei einem Merge musst du Konflikte *einmal* lösen – im Merge-Commit. Die Konfliktlösung wird Teil dieses einen Commits.

Bei einem Rebase musst du Konflikte *für jeden Commit einzeln* lösen, während Git die Commits nacheinander auf die neue Basis anwendet. Das kann aufwändiger sein, führt aber dazu, dass jeder Commit für sich genommen „funktioniert".

```
# Rebase mit Konflikten
git rebase main
# CONFLICT in datei.php
# Konflikt lösen, dann:
git add datei.php
git rebase --continue
# Möglicherweise nächster Konflikt im nächsten Commit...
```

4. Auswirkungen auf andere Entwickler

Hier liegt der **kritischste Unterschied**:

Szenario	Merge	Rebase
Commits noch nicht gepusht	✅ Sicher	✅ Sicher
Commits bereits gepusht	✅ Sicher	⚠️ **Gefährlich!**
Andere arbeiten auf dem Branch	✅ Sicher	❌ **Sehr problematisch!**

Wenn du Commits rebasst, die bereits gepusht wurden, änderst du deren SHA-Hashes. Wenn ein Kollege diese Commits bereits hat, entstehen *Duplikate* in der Historie, weil Git die alten und neuen Commits als unterschiedlich betrachtet.

Wann sollte ich Merge verwenden?

Merge ist die richtige Wahl, wenn:

1. **Du öffentliche Historie integrierst** – Wenn du Änderungen aus `main` in deinen Feature-Branch holst und dieser Branch bereits gepusht wurde, ist Merge sicherer.
2. **Mehrere Entwickler am gleichen Branch arbeiten** – Ein Merge verändert keine bestehenden Commits, sodass niemand Probleme bekommt.

3. **Du die komplette Entwicklungshistorie bewahren möchtest** – Für Auditing oder Nachvollziehbarkeit kann die „echte" Historie wichtig sein.
4. **Du größere Feature-Branches in `main` integrierst** – Viele Teams bevorzugen hier einen Merge-Commit als „Markierung", dass ein Feature abgeschlossen wurde.
5. **Du unsicher bist** – Merge ist die sicherere Option. Im Zweifel: Merge verwenden.

Beispiel: Feature in main mergen

```
git checkout main
git merge feature --no-ff  # --no-ff erzwingt einen Merge-Commit
```

Das `--no-ff` (no fast-forward) Flag ist wichtig, wenn du *explizit* einen Merge-Commit möchtest, auch wenn ein Fast-Forward möglich wäre.

Wann sollte ich Rebase verwenden?

Rebase ist die richtige Wahl, wenn:

1. **Du deine lokale Arbeit aufräumen möchtest, bevor du sie teilst** – Vor dem ersten Push ist Rebase völlig sicher und macht deine Commits präsentabler.
2. **Du deinen Feature-Branch auf den neuesten Stand von `main` bringen möchtest** (und der Branch noch nicht gepusht wurde oder nur du daran arbeitest).
3. **Du eine lineare, saubere Historie bevorzugst** – Manche Teams haben die Konvention, dass Feature-Branches vor dem Merge rebased werden müssen.
4. **Du mit Interactive Rebase Commits zusammenfassen oder aufteilen möchtest** – Das ist nur mit Rebase möglich.

Beispiel: Feature-Branch aktualisieren

```
git checkout feature
git fetch origin
git rebase origin/main
# Jetzt ist dein Feature-Branch auf dem neuesten Stand
```

In PhpStorm kannst du das über **Git → Rebase...** machen oder über das Git-Log-Fenster (Rechtsklick auf `origin/main` → „Rebase Current onto Selected").

Der kombinierte Workflow: Das Beste aus beiden Welten

Viele professionelle Teams nutzen einen **kombinierten Workflow**, der die Vorteile beider Ansätze vereint:

```
flowchart TD
    A["Feature-Branch erstellen"] --> B["Lokal entwickeln\nmit kleinen Commits"]
    B --> C{"Bereit zum Teilen?"}
    C -->|Nein| B
    C -->|Ja| D["Interactive Rebase:\nCommits aufräumen"]
    D --> E["Rebase auf aktuellen main"]
    E --> F["Push Feature-Branch"]
    F --> G["Pull Request erstellen"]
    G --> H["Code Review"]
    H --> I{"Änderungen nötig?"}
    I -->|Ja| J["Änderungen committen"]
    J --> H
    I -->|Nein| K["Merge oder Squash-Merge\nin main"]

    style D fill:#FFD700,color:#000000
    style E fill:#FFD700,color:#000000
    style K fill:#90EE90,color:#000000
```

Die Schritte im Detail

1. **Während der Entwicklung:** Mache so viele kleine Commits wie nötig, ohne dir über die „Schönheit" der Historie Gedanken zu machen.
2. **Vor dem Push:** Nutze Interactive Rebase (`git rebase -i`), um deine Commits aufzuräumen – Work-in-Progress-Commits zusammenfassen, Commit-Messages verbessern.
3. **Vor dem Pull Request:** Rebase auf den aktuellen `main`, um sicherzustellen, dass dein Branch sauber auf dem neuesten Stand aufbaut.
4. **Beim Mergen:** Der Pull Request wird entweder mit einem Merge-Commit oder einem Squash-Merge in `main` integriert.

Die „goldene Regel" des Rebasings

> **Rebase niemals Commits, die bereits gepusht wurden und an denen andere arbeiten könnten!**

Diese Regel ist so wichtig, dass sie einen eigenen Abschnitt verdient. Hier ist, was passiert, wenn du sie brichst:

Das Katastrophen-Szenario

```
sequenceDiagram
    participant Du
    participant GitHub
    participant Kollege

    Du->>GitHub: Push commits X, Y
    GitHub->>Kollege: Pull commits X, Y
    Note over Kollege: Kollege arbeitet<br/>basierend auf Y
    Du->>Du: Rebase X, Y → X', Y'
    Du->>GitHub: Force Push X', Y'
    Note over GitHub: X, Y ersetzt durch X', Y'
    Kollege->>GitHub: Push neuer Commit Z
    Note over GitHub: KONFLIKT!<br/>Z basiert auf Y,<br/>aber Y existiert nicht mehr
    GitHub-->>Kollege: Rejected!
    Note over Kollege: Muss jetzt manuell<br/>reparieren
```

Dein Kollege hat jetzt ein ernsthaftes Problem: Seine Arbeit basiert auf Commits, die in der offiziellen Historie nicht mehr existieren.

Was tun, wenn es doch passiert ist?

Falls du versehentlich gepushte Commits rebased hast:

1. **Kommuniziere sofort mit deinem Team** – Informiere alle, die betroffen sein könnten.
2. **Koordiniere die Reparatur** – Alle müssen ihre lokalen Branches aktualisieren:

```
git fetch origin
git reset --hard origin/feature
# ACHTUNG: Lokale Änderungen gehen verloren!
```

3. **Lerne daraus** – Richte Protected Branches ein, die Force-Push verbieten.

Merge und Rebase in PhpStorm

Merge durchführen

1. Stelle sicher, dass du auf dem Ziel-Branch bist (z.B. `feature`)
2. Gehe zu **Git → Merge...**
3. Wähle den Branch, den du einmergen möchtest (z.B. `main`)
4. Klicke auf **Merge**
5. Bei Konflikten öffnet sich der Merge-Dialog

Rebase durchführen

1. Stelle sicher, dass du auf dem Branch bist, den du rebasen möchtest (z.B. `feature`)
2. Gehe zu **Git → Rebase...**
3. Wähle den Branch, auf den du rebasen möchtest (z.B. `main`)
4. Klicke auf **Rebase**
5. Bei Konflikten:
   - Löse jeden Konflikt im Editor
   - Klicke auf **Continue Rebase** in der Notification

Tipp: Rebase über das Git-Log-Fenster

Eine besonders intuitive Methode in PhpStorm:

1. Öffne das **Git**-Tool-Window (Alt+9 / Cmd+9)
2. Sieh dir das Commit-Log an
3. Rechtsklicke auf den Commit, auf den du rebasen möchtest
4. Wähle **Rebase Current onto Selected**

Praktisches Beispiel: Ein Tag im Leben eines Entwicklers

Lass mich einen typischen Workflow durchspielen:

Morgens: Feature-Branch erstellen

```
git checkout main
git pull
git checkout -b feature/user-profile
```

Während des Tages: Entwickeln mit vielen kleinen Commits

```
git commit -m "WIP: Grundstruktur angelegt"
git commit -m "WIP: Formular hinzugefügt"
git commit -m "Fix Typo"
git commit -m "WIP: Validierung"
git commit -m "Fertig mit Validierung"
```

Vor dem Feierabend: Aufräumen mit Interactive Rebase

```
git rebase -i HEAD~5
# Im Editor: squash die WIP-Commits zusammen
# Ergebnis: 2 saubere Commits statt 5 chaotische
```

Nächster Morgen: Auf aktuellen main rebasen

```
git fetch origin
git rebase origin/main
# Konflikte lösen, falls nötig
```

Push und Pull Request

```
git push -u origin feature/user-profile
# Pull Request auf GitHub erstellen
```

Nach dem Review: Merge in main

Auf GitHub: **Squash and Merge** oder **Merge Commit** – je nach Team-Konvention.

Zusammenfassung: Merge vs. Rebase auf einen Blick

Kriterium	Merge	Rebase
Historie	Verzweigt, zeigt parallele Entwicklung	Linear, als wäre sequentiell entwickelt worden
Neue Commits	Ein Merge-Commit	Keine neuen Commits (außer neu erstellten)
Original-Commits	Bleiben unverändert	Werden neu erstellt (neue SHA-Hashes)
Sicherheit	Immer sicher	Nur sicher für nicht-gepushte Commits
Konfliktlösung	Einmal im Merge-Commit	Für jeden Commit einzeln
Lesbarkeit	Kann unübersichtlich werden	Sehr übersichtlich
Empfohlen für	Öffentliche Branches, Team-Arbeit	Lokale Arbeit, Branch-Aktualisierung

Meine Empfehlung: Starte mit Merge als Standard – es ist sicherer und verzeihender. Nutze Rebase gezielt für lokale Aufräumarbeiten und Branch-Aktualisierungen, *bevor* du pushst. Mit der Zeit wirst du ein Gefühl dafür entwickeln, wann welcher Ansatz am besten passt. □□

Interactive Rebase meistern – Commits aufräumen wie ein Profi □□

Der **Interactive Rebase** ist eines der mächtigsten Werkzeuge in Git, um deine Commit-Historie nachträglich zu bearbeiten. Stell dir vor, du hättest eine Zeitmaschine für deine Commits: Du kannst sie umbenennen, zusammenfassen, aufteilen, neu ordnen oder komplett entfernen. Das Ergebnis ist eine saubere, verständliche Historie, die anderen (und deinem zukünftigen Ich) das Leben leichter macht.

Was ist Interactive Rebase und wann nutze ich ihn?

Beim normalen `git rebase` wird dein Branch auf einen anderen Basis-Commit „umgepflanzt". Der **Interactive Rebase** (`git rebase -i`) geht einen Schritt weiter: Er öffnet einen Editor, in dem du für *jeden einzelnen Commit* entscheiden kannst, was damit passieren soll.

Typische Anwendungsfälle:

- Du hast mehrere kleine „WIP"-Commits gemacht und möchtest sie vor dem Push zu einem sauberen Commit zusammenfassen
- Eine Commit-Message enthält einen Tippfehler oder ist nicht aussagekräftig genug
- Du hast versehentlich etwas committed, das nicht in die Historie gehört
- Die Reihenfolge deiner Commits ergibt logisch keinen Sinn
- Du möchtest einen großen Commit nachträglich in kleinere, thematisch getrennte Commits aufteilen

> ⚠ **Wichtig:** Interactive Rebase *schreibt die Historie um*. Das bedeutet, dass alle betroffenen Commits neue SHA-Hashes bekommen. **Nutze Interactive Rebase nur für Commits, die du noch nicht gepusht hast**, oder sei dir der

> Konsequenzen bewusst (Force-Push erforderlich, Probleme für andere Team-Mitglieder).

Die Grundlagen: Interactive Rebase auf der Kommandozeile

Bevor wir zu PhpStorm kommen, ist es wichtig, die Grundlagen auf der Kommandozeile zu verstehen. Das hilft dir, zu verstehen, was PhpStorm im Hintergrund macht.

Den Interactive Rebase starten

Um die letzten *n* Commits zu bearbeiten, verwendest du:

```
git rebase -i HEAD~n
```

Zum Beispiel, um die letzten 4 Commits zu bearbeiten:

```
git rebase -i HEAD~4
```

Alternativ kannst du auch einen spezifischen Commit-Hash angeben – dann werden alle Commits *nach* diesem Commit bearbeitet:

```
git rebase -i abc1234
```

Was passiert nach dem Start?

Git öffnet deinen konfigurierten Editor (z.B. Vim, VS Code, oder den von PhpStorm) mit einer Liste aller betroffenen Commits. Diese Liste sieht ungefähr so aus:

```
pick a1b2c3d Feature: Login-Formular hinzugefügt
pick e4f5g6h WIP: Validierung angefangen
pick i7j8k9l Tippfehler korrigiert
pick m0n1o2p Validierung fertiggestellt

# Rebase abc1234..m0n1o2p onto abc1234 (4 commands)
```

```
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's message
# d, drop = remove commit
```

Wichtig zu verstehen: Die Commits werden in *chronologischer Reihenfolge* angezeigt (ältester oben, neuester unten) – das ist umgekehrt zur Anzeige in `git log`!

Die sechs Rebase-Befehle im Detail

1. `pick` (p) – Commit unverändert übernehmen

```
pick a1b2c3d Feature: Login-Formular hinzugefügt
```

Was es macht: Der Commit wird genau so übernommen, wie er ist – gleicher Inhalt, gleiche Message. Das ist die Standardoption.

Wann verwenden: Wenn ein Commit bereits perfekt ist und keine Änderung braucht.

2. `reword` (r) – Commit-Message ändern ✏

```
reword a1b2c3d Feature: Login-Formular hinzugefügt
```

Was es macht: Der Commit-Inhalt bleibt unverändert, aber nach dem Speichern der Rebase-Liste öffnet Git einen Editor, in dem du die Commit-Message bearbeiten kannst.

Wann verwenden:

- Tippfehler in der Commit-Message korrigieren

- Eine aussagekräftigere Beschreibung nachträglich hinzufügen
- Ticket-Nummern oder Issue-Referenzen ergänzen

Beispiel-Workflow:

1. Du änderst `pick` zu `reword` in der Liste
2. Du speicherst und schließt die Liste
3. Git öffnet einen neuen Editor mit der alten Commit-Message
4. Du bearbeitest die Message und speicherst
5. Git setzt den Rebase fort

3. `edit` (e) – Commit anhalten und bearbeiten □□

```
edit a1b2c3d Feature: Login-Formular hinzugefügt
```

Was es macht: Git wendet den Commit an und *pausiert dann*. Du befindest dich in einem Zustand, in dem du:

- Dateien ändern kannst
- Weitere Commits hinzufügen kannst
- Den bestehenden Commit mit `git commit --amend` modifizieren kannst
- Den Commit sogar in mehrere Commits aufteilen kannst

Nach deinen Änderungen setzt du den Rebase mit `git rebase --continue` fort.

Wann verwenden:

- Einen großen Commit in mehrere kleinere aufteilen
- Eine vergessene Datei nachträglich zu einem Commit hinzufügen
- Eine Datei aus einem Commit entfernen (aber behalten)

Beispiel: Einen Commit in zwei aufteilen:

```
# Nach dem Pausieren beim edit-Commit:

# Schritt 1: Den Commit "aufmachen", aber Änderungen behalten
git reset HEAD~1

# Schritt 2: Selektiv committen
git add src/Login/Form.php
git commit -m "Feature: Login-Formular HTML-Struktur"
```

```
git add src/Login/Validation.php
git commit -m "Feature: Login-Formular Validierung"

# Schritt 3: Rebase fortsetzen
git rebase --continue
```

4. `squash` (s) – Mit vorherigem Commit verschmelzen □□

```
pick a1b2c3d Feature: Login-Formular hinzugefügt
squash e4f5g6h WIP: Validierung angefangen
squash i7j8k9l Validierung fertiggestellt
```

Was es macht: Der Commit wird mit dem *vorherigen* Commit (dem darüber in der Liste) verschmolzen. Die Änderungen beider Commits werden kombiniert. Git öffnet dann einen Editor, in dem du eine neue, kombinierte Commit-Message schreiben kannst – dabei siehst du die Messages aller beteiligten Commits.

Wann verwenden:

- Mehrere „Work in Progress"-Commits zu einem sauberen Commit zusammenfassen
- Commits, die logisch zusammengehören, vereinen
- Die Historie vor dem Push aufräumen

Der kombinierte Message-Editor sieht so aus:

```
# This is a combination of 3 commits.
# This is the 1st commit message:

Feature: Login-Formular hinzugefügt

# This is the commit message #2:

WIP: Validierung angefangen

# This is the commit message #3:

Validierung fertiggestellt
```

```
# Please enter the commit message for your changes.
```

Du löschst oder kommentierst die nicht benötigten Zeilen aus und schreibst eine saubere, neue Message:

```
Feature: Login-Formular mit Validierung hinzugefügt


- Formular-HTML-Struktur erstellt
- Client-seitige Validierung implementiert
- Server-seitige Validierung hinzugefügt
```

5. `fixup` (f) – Verschmelzen ohne Message □□

```
pick a1b2c3d Feature: Login-Formular hinzugefügt
fixup e4f5g6h WIP
fixup i7j8k9l Tippfehler
```

Was es macht: Genau wie `squash`, aber die Commit-Message des fixup-Commits wird *automatisch verworfen*. Es wird nur die Message des Basis-Commits (des `pick`-Commits darüber) verwendet. Kein Editor öffnet sich für die Message-Bearbeitung.

Wann verwenden:

- Kleine Korrekturen, die zu einem vorherigen Commit gehören
- „Oops, das habe ich vergessen"-Commits
- Commits mit wenig aussagekräftigen Messages wie „WIP", „fix", „typo"

Tipp: Du kannst im Voraus Commits erstellen, die für Fixup gedacht sind, mit:

```
git commit --fixup=abc1234
```

Git erstellt dann automatisch einen Commit mit der Message `fixup! <Original-Message>`. Später kannst du mit `git rebase -i --autosquash` diese Fixup-Commits automatisch an die richtige Stelle sortieren lassen!

6. `drop` (d) – Commit komplett entfernen □□

```
pick a1b2c3d Feature: Login-Formular hinzugefügt
drop e4f5g6h Debug-Code versehentlich committed
pick i7j8k9l Validierung fertiggestellt
```

Was es macht: Der Commit wird komplett aus der Historie entfernt. Die Änderungen dieses Commits sind danach *nicht mehr vorhanden* – weder die Dateien noch die Commit-Message.

Wann verwenden:

- Ein Commit enthält Testcode oder Debug-Ausgaben, die nicht in die Historie gehören
- Ein Commit war ein Fehler und sollte nie existiert haben
- Du möchtest Rauschen aus der Historie entfernen

Alternativ: Du kannst auch einfach die Zeile mit dem Commit aus der Liste löschen – das hat denselben Effekt wie `drop`.

> ⚠ **Vorsicht:** Wenn spätere Commits von den Änderungen des gelöschten Commits abhängen, wirst du Konflikte bekommen!

Commits umsortieren

Eine oft übersehene Funktion: Du kannst die Reihenfolge der Zeilen in der Rebase-Liste einfach ändern, um Commits umzusortieren!

Vorher:

```
pick a1b2c3d Feature A
pick e4f5g6h Feature C
pick i7j8k9l Feature B
```

Nachher:

```
pick a1b2c3d Feature A
pick i7j8k9l Feature B
pick e4f5g6h Feature C
```

Wann verwenden:

- Commits in eine logischere Reihenfolge bringen

- Verwandte Commits zusammenbringen, bevor du sie squashst
- Die Historie für Code-Reviews verständlicher machen

> ⚠ **Vorsicht:** Wenn Commits voneinander abhängen (Commit B ändert Code, den Commit A eingeführt hat), kann das Umsortieren zu Konflikten führen.

Übersicht: Alle Befehle auf einen Blick

Befehl	Kurzform	Änderungen	Message	Wann verwenden
`pick`	`p`	behalten	behalten	Commit ist perfekt
`reword`	`r`	behalten	ändern	Message korrigieren
`edit`	`e`	pausieren & ändern	ändern möglich	Commit aufteilen/erweitern
`squash`	`s`	mit vorherigem vereinen	kombinieren	Commits zusammenfassen
`fixup`	`f`	mit vorherigem vereinen	verwerfen	Kleine Korrekturen
`drop`	`d`	entfernen	entfernen	Commit löschen

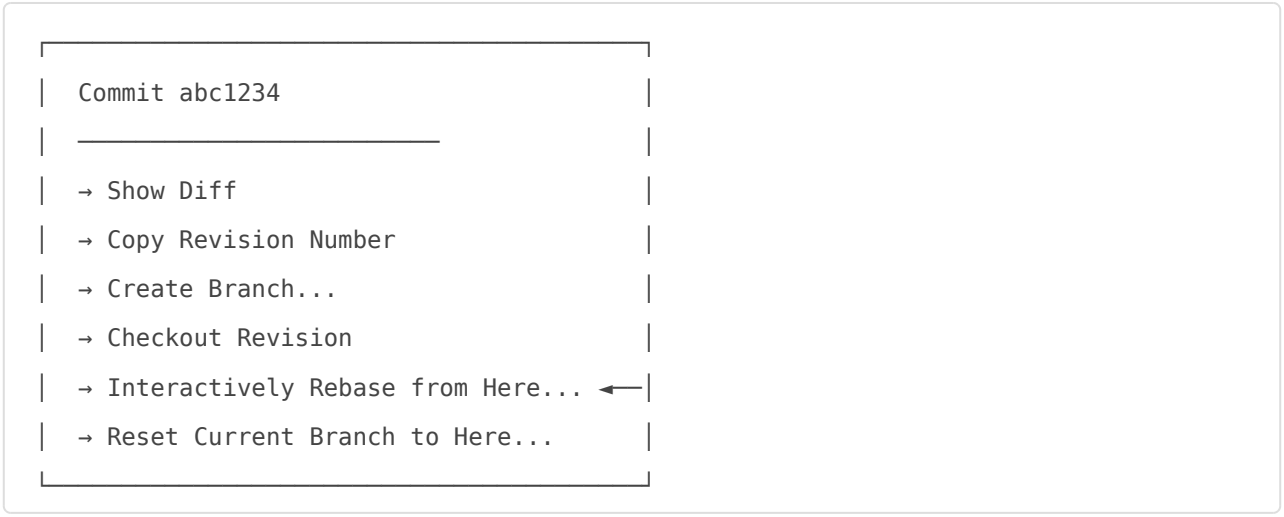
Interactive Rebase in PhpStorm □□

PhpStorm bietet eine komfortable grafische Oberfläche für Interactive Rebase, die besonders für Einsteiger viel zugänglicher ist als die Kommandozeile.

Methode 1: Über das Git-Log-Fenster

1. **Git-Log öffnen:** Gehe zu **View → Tool Windows → Git** oder klicke unten auf den Reiter „Git"
2. **Den Basis-Commit finden:** Scrolle im Log zu dem Commit, *ab dem* du die Historie bearbeiten möchtest (der Commit selbst wird nicht bearbeitet, nur alle danach)
3. **Kontextmenü öffnen:** Rechtsklick auf diesen Commit

4. **Rebase starten:** Wähle **„Interactively Rebase from Here..."**



Methode 2: Über das Hauptmenü

1. Gehe zu **Git → Rebase...**
2. Wähle im Dialog die Option **„Interactive"**
3. Wähle den Basis-Commit oder Branch aus

Methode 3: Über den aktuellen Branch

1. Klicke unten rechts auf den Branch-Namen in der Statusleiste
2. Rechtsklick auf deinen aktuellen Branch
3. Wähle **„Rebase Current onto Selected..."** und dann die Interactive-Option

Der PhpStorm Interactive Rebase Dialog

Nach dem Starten öffnet sich ein übersichtlicher Dialog:

```
│  │ fixup   ▼ │ i7j8k9l         │ typo                          │   │
│  │ pick    ▼ │ m0n1o2p         │ Validierung fertig            │   │
│  └───────────┴─────────────────┴───────────────────────────────┘   │
│                                                                    │
│  [↑ Move Up]  [↓ Move Down]  [Show Diff]                           │
│                                                                    │
├────────────────────────────────────────────────────────────────────┤
│                              [Cancel]  [Start Rebasing]            │
└────────────────────────────────────────────────────────────────────┘
```

Aktionen in PhpStorm ausführen

Action ändern: Klicke auf das Dropdown-Menü in der „Action"-Spalte neben jedem Commit. Du siehst alle verfügbaren Optionen:

- pick
- edit
- reword (wird in PhpStorm manchmal als „Edit Message" bezeichnet)
- squash
- fixup
- drop (oder „Skip")

Commits umsortieren: Wähle einen Commit aus und nutze die Buttons **„Move Up"** und **„Move Down"** oder ziehe die Commits per Drag & Drop an die gewünschte Position

Commit-Diff ansehen: Wähle einen Commit und klicke auf **„Show Diff"**, um zu sehen, welche Änderungen dieser Commit enthält – sehr hilfreich, um zu entscheiden, was damit passieren soll

Commit-Message vorab bearbeiten: Bei `reword` kannst du in manchen PhpStorm-Versionen die neue Message direkt im Dialog eingeben, ohne dass später ein separater Editor öffnet

Schritt-für-Schritt: Commits in PhpStorm squashen

Hier ein konkretes Beispiel – du hast diese Commits und möchtest sie aufräumen:

```
pick a1b2c3d Feature: User-Profil-Seite erstellt
pick e4f5g6h WIP
pick i7j8k9l noch mehr WIP
pick m0n1o2p Profil-Seite fertig
```

Ziel: Alle vier Commits zu einem einzigen, sauberen Commit zusammenfassen.

1. **Rebase starten:** Rechtsklick auf den Commit *vor* `a1b2c3d` → „Interactively Rebase from Here..."
2. **Actions konfigurieren:**

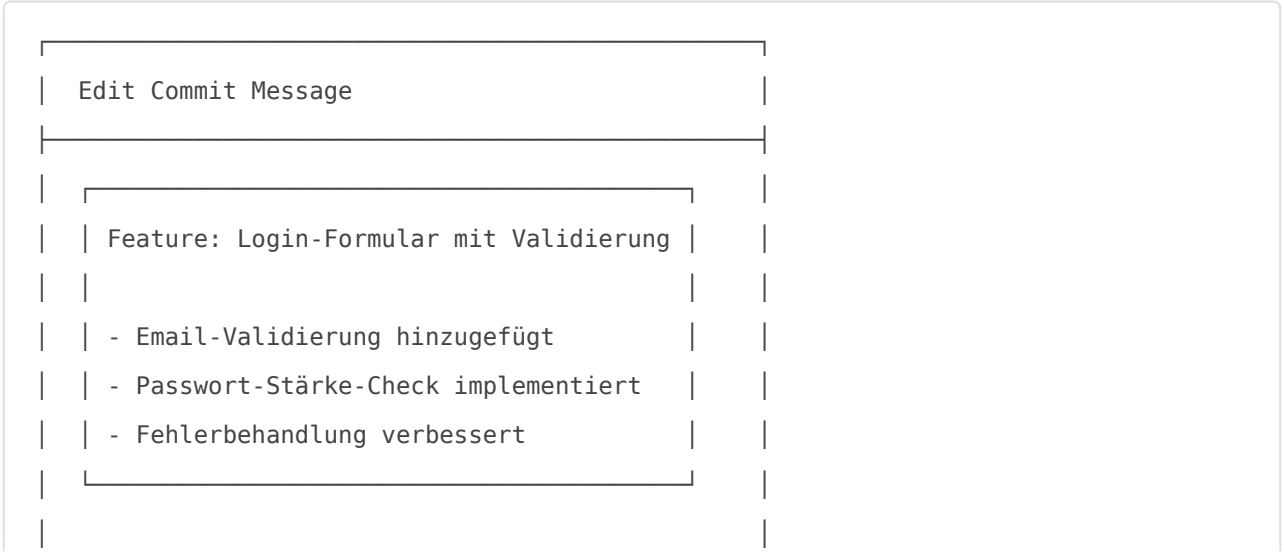
Commit	Aktion	Warum
a1b2c3d	`pick`	Basis-Commit, behält die Message
e4f5g6h	`fixup`	Soll zu a1b2c3d hinzugefügt werden, Message verwerfen
i7j8k9l	`fixup`	Soll auch hinzugefügt werden, Message verwerfen
m0n1o2p	`fixup`	Letzter Teil, Message verwerfen

3. **„Start Rebasing" klicken**
4. **Ergebnis:** Ein einziger Commit `a1b2c3d` mit der Message „Feature: User-Profil-Seite erstellt", der alle Änderungen enthält

Wenn du `squash` statt `fixup` verwendest: PhpStorm öffnet nach dem Rebase einen Editor, in dem du eine neue, kombinierte Message schreiben kannst.

Schritt-für-Schritt: Commit-Message in PhpStorm ändern

1. **Rebase starten**
2. **Action auf `reword` setzen:** Ändere die Action für den gewünschten Commit zu „reword"
3. **„Start Rebasing" klicken**
4. **Message bearbeiten:** PhpStorm öffnet einen Commit-Message-Editor für jeden `reword`-Commit



```
│                    [Cancel]  [Save Message]      │
└──────────────────────────────────────────────────┘
```

5. **Speichern:** Klicke auf „Save Message" oder drücke Ctrl+Enter

Den `edit`-Befehl in PhpStorm nutzen

Der `edit`-Befehl ist etwas spezieller, weil er den Rebase *pausiert* und dir Kontrolle gibt:

1. **Action auf `edit` setzen und Rebase starten**
2. **PhpStorm-Benachrichtigung:** Du siehst eine Notification:

   > „Interactive rebase stopped. You can amend the commit or continue rebasing."

3. **Änderungen vornehmen:**
   - Dateien bearbeiten
   - Neue Dateien zur Staging Area hinzufügen
   - `Commit` (ohne Push!) um den aktuellen Commit zu erweitern
   - Oder im Terminal: `git commit --amend`
4. **Rebase fortsetzen:**
   - Über die Notification: Klicke auf **„Continue Rebasing"**
   - Über das Menü: **Git → Continue Rebasing**
   - Im Terminal: `git rebase --continue`

Konflikte während des Interactive Rebase

Beim Umsortieren oder Squashen kann es zu Konflikten kommen, besonders wenn Commits voneinander abhängen.

Konflikte erkennen

PhpStorm zeigt dir eine klare Meldung:

```
┌──────────────────────────────────────────────────┐
│  ⚠ Rebasing conflict                             │
```

```
┌──────────────────────────────────────────────────────────┐
│                                                          │
│  Conflicts in the following files:                       │
│  • src/Login/Form.php                                    │
│  • src/Login/Validation.php                              │
│                                                          │
│  [Resolve Conflicts]  [Abort Rebase]                     │
└──────────────────────────────────────────────────────────┘
```

Konflikte lösen

1. **„Resolve Conflicts" klicken:** Der Merge-Dialog öffnet sich
2. **3-Way-Merge nutzen:** PhpStorm zeigt dir:
   - **Left (Yours/HEAD):** Der aktuelle Stand nach den bisherigen Rebase-Schritten
   - **Right (Theirs):** Die Änderungen aus dem Commit, der gerade angewendet wird
   - **Center (Result):** Das Ergebnis, das du zusammenbaust
3. **Konflikte einzeln lösen:** Nutze die Buttons „Accept Left", „Accept Right" oder bearbeite manuell
4. **Alle Konflikte gelöst:** Klicke auf **„Apply"**
5. **Rebase fortsetzen:** Klicke auf **„Continue Rebasing"** in der Notification

Rebase abbrechen

Wenn du merkst, dass der Rebase in eine Sackgasse führt:

- **In PhpStorm:** Klicke auf „Abort Rebase" oder gehe zu **Git → Abort Rebasing**
- **Im Terminal:** `git rebase --abort`

Das stellt den exakten Zustand vor dem Rebase wieder her – als wäre nichts passiert.

Praktisches Beispiel: Kompletter Workflow

Hier ein realistisches Szenario, das alles zusammenbringt:

Ausgangssituation

Du hast an einem Feature gearbeitet und folgende Commits erstellt:

```
1. abc1234 - WIP: Angefangen mit der API
2. def5678 - API Endpunkt erstellt
3. ghi9012 - FIXME: Debug-Ausgaben drin!!!
4. jkl3456 - API fertig, aber Typo in Message
5. mno7890 - Dokumentation hinzugefügt
```

Ziel

Bevor du pushst, möchtest du:

- Die ersten zwei Commits zusammenfassen
- Den Debug-Commit entfernen
- Den Typo in Commit 4 korrigieren
- Die Dokumentation nach oben verschieben (vor die API-Commits)

Schritt für Schritt in PhpStorm

1. **Rebase starten:** Rechtsklick auf den Commit *vor* abc1234 → „Interactively Rebase from Here..."
2. **Dialog konfigurieren:**

Original	Aktion	Neue Position	Ergebnis
mno7890	pick	↑ nach oben verschoben	bleibt
abc1234	pick	—	wird Basis für squash
def5678	squash	—	verschmilzt mit abc1234
ghi9012	drop	—	wird entfernt
jkl3456	reword	—	Message wird korrigiert

   Die neu sortierte Liste sieht so aus:

   ```
   pick    mno7890  Dokumentation hinzugefügt
   pick    abc1234  WIP: Angefangen mit der API
   squash  def5678  API Endpunkt erstellt
   reword  jkl3456  API fertig, aber Typo in Message
   ```

3. **„Start Rebasing" klicken**
4. **Squash-Message bearbeiten:** PhpStorm öffnet einen Editor für die kombinierte Message:

```
Feature: REST-API implementiert


- Endpunkt /api/users erstellt
- Authentifizierung hinzugefügt
- Response-Format definiert
```

5. **Reword-Message bearbeiten:** Danach öffnet sich der Editor für den Typo-Fix
6. **Ergebnis:** Saubere Historie mit drei aussagekräftigen Commits:

```
1. Dokumentation hinzugefügt
2. Feature: REST-API implementiert
3. API-Tests und finale Korrekturen
```

Best Practices für Interactive Rebase □□

Vor dem Rebase:

- Stelle sicher, dass dein Working Directory „clean" ist (keine uncommitteten Änderungen)
- Erstelle ggf. einen Backup-Branch: `git branch backup-vor-rebase`
- Nutze Interactive Rebase *vor* dem Push, nicht danach

Während des Rebase:

- Arbeite von oben nach unten durch die Liste
- Squashe immer in den älteren Commit (der oben steht)
- Bei Unsicherheit: lieber abbrechen und neu starten
- Nutze `edit` sparsam – er ist mächtig, aber auch komplex

Commit-Messages nach dem Squash:

- Schreibe eine neue, zusammenfassende Message
- Lösche die alten WIP-Messages komplett
- Folge dem Conventional Commits Format, wenn dein Team es nutzt

Wenn etwas schiefgeht:

- `git rebase --abort` ist dein Freund
- Das Reflog hat alles gespeichert: `git reflog` zeigt dir alle Zustände
- Mit `git reset --hard HEAD@{n}` kannst du zu jedem vorherigen Zustand zurück

Zusammenfassung

Interactive Rebase ist ein unverzichtbares Werkzeug für professionelle Git-Nutzung. Mit den sechs Befehlen `pick`, `reword`, `edit`, `squash`, `fixup` und `drop` kannst du deine Commit-Historie perfekt aufräumen, bevor du sie mit der Welt teilst. PhpStorm macht diesen Prozess durch seine grafische Oberfläche besonders zugänglich – du siehst auf einen Blick alle Commits, kannst Actions per Dropdown wählen und Commits per Drag & Drop umsortieren.

Die wichtigsten Takeaways:

- □□ **Nutze Interactive Rebase nur für lokale, noch nicht gepushte Commits**
- □□ **Squash/Fixup** sind deine Werkzeuge für saubere, atomare Commits
- ✏ **Reword** ermöglicht nachträgliche Message-Korrekturen ohne Aufwand
- □□ **Drop** entfernt ungewollte Commits komplett aus der Historie
- □□ **Edit** gibt dir volle Kontrolle, ist aber auch am komplexesten
- ↩ **Im Zweifel:** `git rebase --abort` und von vorne beginnen

Commits zusammenfassen (Squash) in PhpStorm □□

Du hast also fleißig gearbeitet und dabei mehrere kleine Commits erstellt – vielleicht mit Nachrichten wie *„WIP*", *„Fix typo*", *„Noch ein Fix*" oder *„Jetzt aber wirklich fertig*". Bevor du diese „Commits des Schreckens" auf GitHub pushst, möchtest du sie zu einem einzigen, sauberen Commit zusammenfassen. Das ist nicht nur guter Stil, sondern macht die Historie für dich und dein Team deutlich lesbarer. Lass mich dich Schritt für Schritt durch den gesamten Prozess führen.

Warum überhaupt Commits squashen?

Bevor wir in die Praxis einsteigen, kurz zum **Warum**: Eine saubere Commit-Historie ist wie ein gut geschriebenes Tagebuch deines Projekts. Wenn du später (oder ein Kollege) nachvollziehen möchtest, welche Änderung wann und warum gemacht wurde, helfen aussagekräftige Commits enorm. Niemand möchte sich durch zehn *„WIP*"-Commits wühlen, um zu verstehen, was eigentlich passiert ist.

Vorteile des Squashens:

- **Lesbarkeit:** Ein Commit pro logischer Änderung statt vieler Mikro-Commits
- **Einfacheres Debugging:** Mit `git bisect` findest du Bugs schneller, wenn jeder Commit einen sinnvollen Zustand repräsentiert
- **Saubere Pull Requests:** Reviewer können die Änderungen besser nachvollziehen
- **Professioneller Eindruck:** Zeigt, dass du dir Gedanken über deine Arbeit machst

---

Die Ausgangssituation

Nehmen wir an, du hast in den letzten Stunden an einem Login-Formular gearbeitet und dabei folgende Commits erstellt (vom ältesten zum neuesten):

```
a1b2c3d  WIP: Login-Formular angefangen
d4e5f6g  Passwort-Feld hinzugefügt
h7i8j9k  Typo im Label gefixt
l0m1n2o  CSS angepasst
p3q4r5s  Validation hinzugefügt
```

Diese fünf Commits sollen zu einem einzigen werden:

```
x9y8z7w  feat: Login-Formular mit Validierung implementiert
```

Methode 1: Der komfortable Weg über das Git-Log in PhpStorm

Dies ist die **empfohlene Methode** für die meisten Situationen, da sie visuell und intuitiv ist.

1. **Öffne das Git-Tool-Window**
   Navigiere zu **View → Tool Windows → Git** oder nutze die Tastenkombination `Alt + 9` (Windows/Linux) bzw. `⌘ + 9` (macOS). Du siehst jetzt das Git-Log mit allen Commits deines Repositories.
2. **Filtere auf deinen aktuellen Branch**
   In der Toolbar des Git-Fensters siehst du ein Dropdown-Menü für Branches. Stelle sicher, dass du nur die Commits deines Feature-Branches siehst. Du kannst auch im Suchfeld nach deinem Branch-Namen filtern.
3. **Identifiziere den Commit-Bereich**
   Scrolle durch die Liste und identifiziere die Commits, die du zusammenfassen möchtest. In unserem Beispiel sind das die fünf Commits von *„WIP: Login-Formular angefangen*" bis *„Validation hinzugefügt*".
4. **Wähle den Basis-Commit aus**
   Hier wird es wichtig: Du musst den Commit **vor** deinem ersten zu squashenden Commit finden. Das ist der Commit, auf dem deine Arbeit aufbaut – typischerweise der letzte Commit, bevor du mit dem Feature angefangen hast. Dieser Commit selbst wird **nicht** verändert, er dient nur als Referenzpunkt.
5. **Starte den Interactive Rebase**
   Klicke mit der **rechten Maustaste** auf diesen Basis-Commit und wähle im Kontextmenü: **Interactively Rebase from Here...** (auf Deutsch: *„Interaktiv Rebase von hier...*").
6. **Das Interactive Rebase-Fenster**
   PhpStorm öffnet jetzt ein Fenster mit dem Titel *„Rebasing Commits*" oder *„Git Interactive Rebase*". Du siehst eine Liste aller Commits, die nach deinem gewählten Basis-Commit kommen – also genau die Commits, die du bearbeiten möchtest. Die Liste zeigt:

- Die **Aktion** (standardmäßig „pick" für jeden Commit)
- Den **Commit-Hash** (verkürzt)
- Die **Commit-Nachricht**

7. **Commits zum Squashen markieren**
   Jetzt kommt der entscheidende Schritt. Du hast mehrere Möglichkeiten, die Aktion für jeden Commit zu ändern:
   - **Doppelklick** auf die Aktion öffnet ein Dropdown-Menü
   - **Rechtsklick** auf einen Commit zeigt alle verfügbaren Aktionen
   - Wähle mehrere Commits mit `Strg + Klick` (Windows/Linux) oder `⌘ + Klick` (macOS) und ändere die Aktion für alle gleichzeitig

   Für das Squashen gehst du so vor:
   - Der **erste Commit** (der älteste, auf den die anderen aufbauen) behält die Aktion **„pick"**
   - Alle **folgenden Commits**, die du zusammenfassen möchtest, änderst du auf **„squash"** oder **„fixup"**
8. **Der Unterschied zwischen „squash" und „fixup"**

Aktion	Beschreibung
squash	Kombiniert den Commit mit dem vorherigen. Die Commit-Nachricht wird beibehalten, und du kannst sie im nächsten Schritt bearbeiten.
fixup	Kombiniert den Commit mit dem vorherigen, aber **verwirft** die Commit-Nachricht. Nutze das für Commits wie *„Typo gefixt*", deren Nachricht nicht erhalten bleiben muss.

   Für unser Beispiel könntest du es so konfigurieren:

```
pick    a1b2c3d  WIP: Login-Formular angefangen
squash  d4e5f6g  Passwort-Feld hinzugefügt
fixup   h7i8j9k  Typo im Label gefixt
squash  l0m1n2o  CSS angepasst
squash  p3q4r5s  Validation hinzugefügt
```

9. **Starte den Rebase**
   Klicke auf **„Start Rebasing"** (oder den entsprechenden Button in deiner PhpStorm-Version). PhpStorm beginnt jetzt, die Commits neu zu schreiben.
10. **Bearbeite die kombinierte Commit-Nachricht**
    Nach dem Rebase öffnet PhpStorm ein Fenster, in dem du die **neue, kombinierte Commit-Nachricht** eingeben kannst. Du siehst dort alle Nachrichten der Commits, die mit „squash" markiert waren (nicht die mit „fixup" – deren Nachrichten wurden verworfen). Das sieht ungefähr so aus:

```
# This is a combination of 5 commits.
# The first commit's message is:
```

```
WIP: Login-Formular angefangen

# This is the 2nd commit message:
Passwort-Feld hinzugefügt

# This is the 4th commit message:
CSS angepasst

# This is the 5th commit message:
Validation hinzugefügt
```

Lösche diesen ganzen Text und schreibe stattdessen eine saubere, aussagekräftige Commit-Nachricht:

```
feat: Login-Formular mit Validierung implementiert


- Login-Formular mit Benutzername- und Passwort-Feld erstellt
- Client-seitige Validierung für beide Felder hinzugefügt
- CSS-Styling für konsistentes Design angepasst
```

11. **Bestätige die Nachricht**
    Klicke auf **„OK"** oder **„Commit"**, um den Prozess abzuschließen.

Methode 2: Der schnelle Weg mit „Squash Commits" direkt in PhpStorm

PhpStorm bietet auch eine **direkte Squash-Funktion**, die noch schneller ist, wenn du weißt, welche Commits du zusammenfassen möchtest.

1. **Öffne das Git-Log** wie in Methode 1 beschrieben (`Alt + 9` oder `⌘ + 9`).
2. **Wähle die Commits aus**, die du zusammenfassen möchtest. Halte `Strg` (Windows/Linux) oder `⌘` (macOS) gedrückt und klicke auf jeden Commit, den du squashen möchtest. Die Commits müssen **aufeinanderfolgend** sein.
3. **Rechtsklick auf die Auswahl** und wähle im Kontextmenü: **Squash Commits...** (auf Deutsch: *„Commits zusammenfassen..."*).
4. **Bearbeite die Commit-Nachricht** im erscheinenden Dialog und bestätige mit **„OK"**.

Diese Methode ist schneller, bietet aber weniger Kontrolle als der Interactive Rebase – du kannst zum Beispiel nicht einzelne Commits mit „fixup" statt „squash" markieren.

Methode 3: Über das Terminal in PhpStorm

Falls du lieber mit dem Terminal arbeitest oder die grafischen Methoden aus irgendeinem Grund nicht funktionieren, kannst du den Interactive Rebase auch direkt über die Kommandozeile durchführen.

1. **Öffne das Terminal in PhpStorm**
   Navigiere zu **View → Tool Windows → Terminal** oder nutze `Alt + F12` (Windows/Linux) bzw. `⌥ + F12` (macOS).
2. **Finde heraus, wie viele Commits du zusammenfassen möchtest**
   Mit `git log --oneline` siehst du eine kompakte Liste deiner letzten Commits:

   ```
   git log --oneline -10
   ```

   Das zeigt dir die letzten 10 Commits. Zähle, wie viele du squashen möchtest – in unserem Beispiel sind es 5.
3. **Starte den Interactive Rebase**

   ```
   git rebase -i HEAD~5
   ```

   Damit sagst du Git: „Ich möchte die letzten 5 Commits interaktiv bearbeiten."
4. **Bearbeite die Commit-Liste im Editor**
   Git öffnet einen Text-Editor (in PhpStorm normalerweise der integrierte Editor oder der von dir konfigurierte). Du siehst:

   ```
   pick a1b2c3d WIP: Login-Formular angefangen
   pick d4e5f6g Passwort-Feld hinzugefügt
   pick h7i8j9k Typo im Label gefixt
   pick l0m1n2o CSS angepasst
   pick p3q4r5s Validation hinzugefügt
   ```

   Ändere es zu:

   ```
   pick a1b2c3d WIP: Login-Formular angefangen
   squash d4e5f6g Passwort-Feld hinzugefügt
   fixup h7i8j9k Typo im Label gefixt
   squash l0m1n2o CSS angepasst
   ```

```
squash p3q4r5s Validation hinzugefügt
```

Du kannst auch die Kurzformen verwenden: `p` für pick, `s` für squash, `f` für fixup.

5. **Speichere und schließe den Editor**
   In PhpStorms integriertem Editor klickst du einfach auf den Bestätigungs-Button. Bei Vim-Editoren: `:wq` und Enter.
6. **Bearbeite die kombinierte Commit-Nachricht** wie bei Methode 1 beschrieben.

Was passiert, wenn etwas schief geht? □□

Keine Panik! Ein Rebase kann abgebrochen werden, solange er noch läuft:

Im Terminal:

```
git rebase --abort
```

In PhpStorm: Wenn der Rebase fehlschlägt (z.B. wegen Konflikten), zeigt PhpStorm eine Benachrichtigung an. Du kannst dort **„Abort Rebase"** wählen, um zum Zustand vor dem Rebase zurückzukehren.

Wenn du den Rebase schon abgeschlossen hast und merkst, dass etwas nicht stimmt, hilft dir das **Reflog**:

```
git reflog
```

Dort siehst du alle Zustände deines Repositories, auch die vor dem Rebase. Mit `git reset --hard HEAD@{n}` (wobei `n` die Nummer aus dem Reflog ist) kannst du zu einem früheren Zustand zurückkehren.

Wichtige Hinweise und Warnungen ⚠

1. **Nur unpushte Commits squashen!**

Die goldene Regel des Rebasings: **Schreibe niemals die Historie um, die bereits gepusht wurde** (es sei denn, du bist dir absolut sicher, was du tust, und hast das mit deinem Team abgesprochen). Wenn du bereits gepushte Commits squashst, müsstest du mit `git push --force` pushen, was die Historie für alle anderen Entwickler kaputt macht.

2. **Force-Push nur bei Feature-Branches**
   Wenn du an einem Feature-Branch arbeitest, den nur du nutzt, und du bereits gepusht hast, kannst du nach dem Squash einen Force-Push machen:

   ```
   git push --force-with-lease origin feature/login-formular
   ```

   `--force-with-lease` ist sicherer als `--force`, weil es prüft, ob jemand anders in der Zwischenzeit gepusht hat.
3. **Mache vorher ein Backup (optional)**
   Wenn du unsicher bist, erstelle vorher einen temporären Branch als Backup:

   ```
   git branch backup-vor-squash
   ```

   Falls etwas schief geht, kannst du immer zu diesem Branch zurückkehren.

Ein Beispiel-Workflow zusammengefasst

```
flowchart TD
    A["Du hast 5 WIP-Commits\nim Feature-Branch"] --> B["Oeffne Git-Log\nAlt+9 oder Cmd+9"]
    B --> C["Finde den Commit VOR\ndeiner Arbeit"]
    C --> D["Rechtsklick:\nInteractively Rebase from Here"]
    D --> E["Markiere Commits\nmit squash oder fixup"]
    E --> F["Klicke: Start Rebasing"]
    F --> G["Schreibe eine saubere\nkombinierte Commit-Nachricht"]
    G --> H["Bestaetigen"]
    H --> I["Fertig: 1 sauberer Commit\nstatt 5 WIP-Commits"]

    style A fill:#ffcccc,color:#000000
    style I fill:#ccffcc,color:#000000
```

Zusammenfassung

Schritt	Aktion
1	Git-Log öffnen (`Alt + 9` / `⌘ + 9`)
2	Basis-Commit identifizieren (der Commit **vor** deiner Arbeit)
3	Rechtsklick → *„Interactively Rebase from Here..."*
4	Commits mit „squash" oder „fixup" markieren
5	*„Start Rebasing"* klicken
6	Neue, saubere Commit-Nachricht schreiben
7	Bestätigen – fertig!

Mit etwas Übung wird das Squashen zur Routine und deine Commit-Historie wird es dir danken. Dein zukünftiges Ich (und deine Kollegen) werden dich dafür lieben, wenn sie durch eine saubere, nachvollziehbare Historie scrollen können, statt sich durch zwanzig *„WIP"*- und *„Fix"*-Commits zu kämpfen!

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!** □□