Long-Running Branches verstehen und synchron halten

Was sind Long-Running Branches?

Long-Running Branches (auch *langlebige* oder *permanente Branches* genannt) sind Branches, die über die gesamte Lebensdauer eines Projekts oder zumindest über längere Zeiträume hinweg bestehen bleiben – im Gegensatz zu **kurzlebigen Feature-Branches**, die nach dem Merge gelöscht werden.

Die typischen Long-Running Branches

In den meisten professionellen Projekten findest du folgende Struktur:

Branch	Zweck	Lebensdauer
`main` (oder `master`)	Produktionscode – enthält nur stabilen, deploybaren Code	Permanent
`develop`	Integrationsbranch für die nächste Version – hier fließen Feature-Branches zusammen	Permanent
`release/*`	Vorbereitung einer neuen Version, nur noch Bugfixes erlaubt	Mittelfristig (Wochen)
`hotfix/*`	Kritische Fixes für Produktion	Kurzfristig (Tage)

```
gitGraph
    commit id: "Initial"
    branch develop
    commit id: "Setup"
    branch feature/login
```

```
    commit id: "Login-Form"
    commit id: "Validation"
    checkout develop
    merge feature/login id: "Merge Login"
    branch release/1.0
    commit id: "Version bump"
    commit id: "Bugfix"
    checkout main
    merge release/1.0 id: "Release 1.0" tag: "v1.0.0"
    checkout develop
    merge release/1.0 id: "Sync Release"
```

Warum ist Synchronisation so wichtig? ⚠

Stell dir folgendes Szenario vor:

1. Du erstellst am Montag einen Feature-Branch von `main`
2. Du arbeitest zwei Wochen an deinem Feature
3. In der Zwischenzeit wurden 50 Commits in `main` gemergt (von anderen Entwicklern oder anderen Features)
4. Dein Feature-Branch ist jetzt **zwei Wochen und 50 Commits hinter** **`main`**

Die Probleme, die entstehen können:

- **Merge-Konflikte:** Je länger du wartest, desto mehr Konflikte häufen sich an. Ein Merge nach zwei Wochen kann dutzende Konflikte haben, die du alle auf einmal lösen musst.
- **Inkompatibilitäten:** Vielleicht hat jemand eine Funktion geändert, die du verwendest. Dein Code funktioniert nicht mehr mit dem aktuellen Stand.
- **Doppelte Arbeit:** Möglicherweise wurde bereits Code geschrieben, den du gerade selbst implementierst.
- **Veraltete Basis:** Du testest dein Feature gegen eine veraltete Codebasis – es könnte in Produktion anders funktionieren.

Die Lösung: Regelmäßige Synchronisation – idealerweise *täglich* oder zumindest bei jedem größeren Meilenstein in `main`.

Die zwei Strategien: Merge vs. Rebase

Es gibt zwei grundlegend verschiedene Ansätze, um Änderungen aus `main` in deinen Feature-Branch zu integrieren. Beide haben ihre Berechtigung, und die Wahl hängt von deinem Workflow und deinen Team-Konventionen ab.

Strategie 1: Merge (main → Feature-Branch)

Bei einem **Merge** werden die Änderungen aus `main` in deinen Feature-Branch integriert, wobei ein **Merge-Commit** entsteht, der beide Historien verbindet.

```
gitGraph
    commit id: "A"
    commit id: "B"
    branch feature/shop
    commit id: "F1"
    commit id: "F2"
    checkout main
    commit id: "C"
    commit id: "D"
    checkout feature/shop
    merge main id: "Merge main" type: HIGHLIGHT
    commit id: "F3"
```

Was passiert hier?

- Dein Feature-Branch enthält nach dem Merge sowohl deine Commits (F1, F2) als auch die neuen Commits aus `main` (C, D)
- Ein Merge-Commit dokumentiert, *wann* du synchronisiert hast
- Die ursprüngliche Historie bleibt vollständig erhalten

Strategie 2: Rebase (Feature-Branch auf main)

Bei einem **Rebase** werden deine Feature-Commits „abgelöst" und auf die Spitze von `main` *neu aufgesetzt*. Es entsteht eine **lineare Historie** ohne Merge-Commits.

```
gitGraph
    commit id: "A"
```

```
    commit id: "B"
    commit id: "C"
    commit id: "D"
    branch feature/shop
    commit id: "F1 neu" type: HIGHLIGHT
    commit id: "F2 neu" type: HIGHLIGHT
    commit id: "F3"
```

Was passiert hier?

- Deine Commits werden „verschoben" – sie basieren jetzt auf dem aktuellen Stand von `main`
- Es entstehen *technisch gesehen* neue Commits (mit neuen Hashes), auch wenn der Inhalt identisch ist
- Die Historie sieht aus, als hättest du erst jetzt mit deiner Arbeit begonnen

---

Ausführliche Anleitung: Synchronisation mit Merge

Szenario

Du arbeitest an einem Feature-Branch `feature/newsletter` und möchtest die neuesten Änderungen aus `main` integrieren.

Schritt 1: Sicherstellen, dass du einen sauberen Arbeitsstand hast

Bevor du irgendetwas synchronisierst, sollte dein Working Directory „sauber" sein – also keine uncommitteten Änderungen enthalten.

In PhpStorm prüfen:

- Schau in der linken unteren Ecke auf den „Git"-Tab
- Unter „Local Changes" sollten keine Dateien stehen

Falls du unfertige Änderungen hast, nutze Stash:

```
# Terminal-Variante
git stash push -m "WIP: Newsletter-Formular"
```

In PhpStorm:

1. Menü: **Git → Uncommitted Changes → Stash Changes...**
2. Gib eine aussagekräftige Nachricht ein
3. Klicke **Create Stash**

Schritt 2: Den neuesten Stand von `main` holen (Fetch)

Zuerst musst du sicherstellen, dass dein lokales Repository die neuesten Informationen vom Remote hat.

In PhpStorm:

1. Klicke auf das blaue ↓ **Pfeil-Symbol** in der Toolbar (oder **Ctrl+T** / **Cmd+T**)
2. Oder: Menü **Git → Fetch**

Dies aktualisiert `origin/main` mit dem Stand auf GitHub, ohne deinen lokalen `main`-Branch zu verändern.

Schritt 3: main in deinen Feature-Branch mergen

Methode A: Über das Git-Log in PhpStorm *(empfohlen)*

1. Öffne das **Git-Log**: Klicke unten auf den Tab „Git" und dann auf „Log"
2. Stelle sicher, dass du auf deinem Feature-Branch bist (sichtbar unten rechts in der Statusleiste)
3. Im Log siehst du alle Branches – finde `origin/main`
4. **Rechtsklick auf den neuesten Commit von `origin/main`**
5. Wähle **„Merge 'origin/main' into 'feature/newsletter'"**

```
flowchart TD
    A["1. Git-Log offnen\n Ansicht: Git - Log"] --> B["2. Branch prufen\n Bist du auf feature/newsletter?"]
    B --> C["3. origin/main finden\n Im Log-Graphen"]
    C --> D["4. Rechtsklick auf\n neuesten Commit"]
```

```
    D --> E["5. Merge origin/main\n into feature/newsletter"]
    E --> F{Konflikte?}
    F -->|Ja| G["Konflikte losen\n 3-Way-Merge-Editor"]
    F -->|Nein| H["Fertig!\n Merge-Commit erstellt"]
    G --> H

    style A fill:#e1f5fe,color:#000000
    style E fill:#fff3e0,color:#000000
    style H fill:#e8f5e9,color:#000000
```

Methode B: Über das Branch-Menü

1. Klicke unten rechts auf den aktuellen Branch-Namen (z.B. `feature/newsletter`)
2. Im Dropdown: Finde **„origin/main"** unter „Remote Branches"
3. Klicke darauf und wähle **„Merge into Current"**

Methode C: Über das Hauptmenü

1. Menü: **Git → Merge...**
2. Wähle `origin/main` aus der Liste
3. Klicke **Merge**

Schritt 4: Konflikte lösen (falls vorhanden)

Wenn es Konflikte gibt, öffnet PhpStorm automatisch ein Dialogfenster „Files Merged with Conflicts":

1. Klicke auf **„Merge..."** neben der konfliktbehafteten Datei
2. Der **3-Way-Merge-Editor** öffnet sich:
   - **Links:** Deine Version (feature/newsletter)
   - **Rechts:** Ihre Version (main)
   - **Mitte:** Das Ergebnis, das du zusammenbaust
3. Nutze die `>>` und `<<` Buttons, um Änderungen zu übernehmen
4. Wenn du fertig bist: **Apply**
5. Nachdem alle Konflikte gelöst sind: Der Merge-Commit wird automatisch erstellt

Schritt 5: Ergebnis überprüfen

Nach dem Merge solltest du kurz prüfen:

1. **Schau ins Git-Log:** Du siehst jetzt einen Merge-Commit mit zwei Eltern-Commits
2. **Führe deine Tests aus:** Stelle sicher, dass alles noch funktioniert

3. **Pushe deinen Branch:** `Ctrl+Shift+K` oder **Git → Push**

Terminal-Befehle (zur Referenz)

```
# 1. Auf Feature-Branch wechseln (falls nicht schon dort)
git checkout feature/newsletter

# 2. Neueste Daten vom Remote holen
git fetch origin

# 3. main in Feature-Branch mergen
git merge origin/main

# 4. Bei Konflikten: Nach dem manuellen Lösen
git add .
git commit  # Merge-Commit abschließen

# 5. Pushen
git push origin feature/newsletter
```

Ausführliche Anleitung: Synchronisation mit Rebase

Wichtiger Hinweis vorab ⚠

Rebase schreibt die Historie um – deine Commits bekommen **neue SHA-Hashes**. Das bedeutet:

- **Wenn du deinen Branch noch nicht gepusht hast:** Kein Problem, rebase frei.
- **Wenn du schon gepusht hast und alleine am Branch arbeitest:** Möglich, aber du brauchst `git push --force-with-lease`.
- **Wenn andere auch an deinem Branch arbeiten: □□ Kein Rebase!** Du würdest die Arbeit der anderen durcheinanderbringen.

Szenario

Du arbeitest an `feature/checkout` und möchtest deine Commits auf den aktuellen Stand von `main` neu aufsetzen.

Schritt 1: Vorbereitung (wie beim Merge)

- Sicherstellen, dass das Working Directory sauber ist
- Unfertige Änderungen ggf. stashen
- `git fetch` ausführen, um `origin/main` zu aktualisieren

Schritt 2: Rebase in PhpStorm starten

Methode A: Über das Git-Menü *(empfohlen für Einsteiger)*

1. Stelle sicher, dass du auf deinem Feature-Branch bist
2. Menü: **Git → Rebase...**
3. Im Dialog „Rebase":
   - „Onto": Wähle `origin/main`
   - Die anderen Optionen kannst du auf Standard lassen
4. Klicke **Rebase**

Methode B: Über das Branch-Popup

1. Klicke unten rechts auf deinen Branch-Namen
2. Finde `origin/main` unter „Remote Branches"
3. Klicke darauf und wähle **„Rebase Current onto Selected"**

Methode C: Über das Git-Log

1. Öffne das Git-Log
2. Rechtsklick auf den neuesten Commit von `origin/main`
3. Wähle **„Rebase 'feature/checkout' onto 'origin/main'"**

Schritt 3: Konflikte während des Rebases lösen

Beim Rebase werden deine Commits *einzeln* auf `main` neu angewendet. Das bedeutet: Wenn es Konflikte gibt, musst du sie **für jeden betroffenen Commit einzeln lösen**.

Der Ablauf bei Konflikten:

```
flowchart TD
    A["Rebase starten"] --> B["Commit 1 anwenden"]
    B --> C{Konflikt?}
    C -->|Nein| D["Commit 2 anwenden"]
    C -->|Ja| E["Konflikt losen"]
    E --> F["Git - Rebase - Continue"]
    F --> D
    D --> G{Konflikt?}
    G -->|Nein| H["Commit 3 anwenden"]
    G -->|Ja| I["Konflikt losen"]
    I --> J["Git - Rebase - Continue"]
    J --> H
    H --> K["Rebase abgeschlossen!"]

    style A fill:#e1f5fe,color:#000000
    style E fill:#ffcccc,color:#000000
    style I fill:#ffcccc,color:#000000
    style K fill:#e8f5e9,color:#000000
```

Konkret in PhpStorm:

1. Bei einem Konflikt zeigt PhpStorm eine Notification: „Rebase stopped due to conflicts"
2. Löse die Konflikte im 3-Way-Merge-Editor (wie beim Merge)
3. Nach dem Lösen: **Git → Rebase → Continue**
4. PhpStorm wendet den nächsten Commit an
5. Wiederhole, bis alle Commits neu angewendet wurden

Wenn du abbrechen möchtest:

- **Git → Rebase → Abort** – Stellt den Zustand vor dem Rebase wieder her

Schritt 4: Force-Push (falls bereits gepusht)

Da Rebase neue Commits erstellt, musst du mit `--force` pushen, wenn der Branch bereits auf GitHub existiert.

In PhpStorm:

1. `Ctrl+Shift+K` (oder **Git → Push**)
2. PhpStorm zeigt dir, dass ein Force-Push nötig ist

3. Klicke auf den kleinen Pfeil neben „Push" und wähle **„Force Push"**

Besser: Verwende `--force-with-lease`:

```
git push --force-with-lease origin feature/checkout
```

Dies ist sicherer als `--force`, weil es prüft, ob jemand anderes in der Zwischenzeit gepusht hat.

In PhpStorm aktivieren:

1. **Settings → Version Control → Git**
2. Aktiviere **„Use --force-with-lease for force push"**

Terminal-Befehle (zur Referenz)

```
# 1. Auf Feature-Branch wechseln
git checkout feature/checkout

# 2. Fetch
git fetch origin

# 3. Rebase starten
git rebase origin/main

# 4a. Bei Konflikten: Nach dem Lösen weitermachen
git add .
git rebase --continue

# 4b. Oder abbrechen
git rebase --abort

# 5. Force-Push (nur wenn schon gepusht)
git push --force-with-lease origin feature/checkout
```

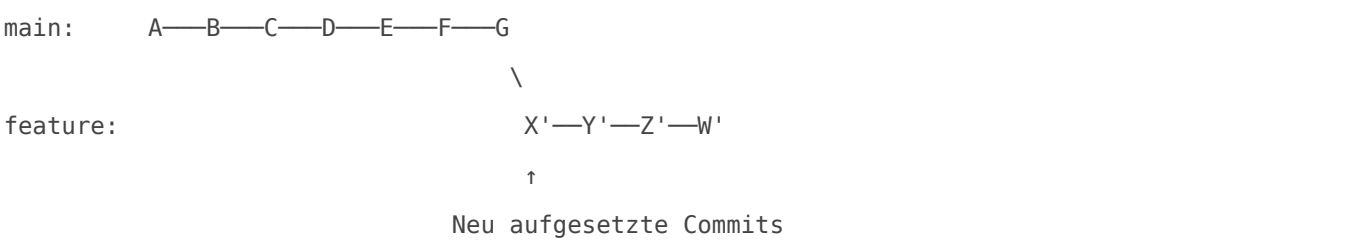
Merge vs. Rebase: Direkter Vergleich

Aspekt	Merge	Rebase
Historie	Bewahrt die komplette, echte Historie mit allen Verzweigungen	Erzeugt eine lineare, „saubere" Historie
Merge-Commits	Ja, ein Merge-Commit pro Synchronisation	Nein, keine zusätzlichen Commits
Commit-Hashes	Bleiben unverändert	Ändern sich (neue Commits)
Konfliktlösung	Alle Konflikte auf einmal	Konflikte pro Commit einzeln
Force-Push nötig?	Nein	Ja, wenn Branch bereits gepusht
Sicher bei Team-Branches?	✅ Ja	⚠ Nur mit Vorsicht
Nachvollziehbarkeit	Sehr gut (wann wurde was integriert?)	Geschichte wird „umgeschrieben"
Komplexität	Einfacher	Komplexer

Visuelle Gegenüberstellung

Nach mehreren Merge-Synchronisationen:

Nach Rebase (vor dem finalen Merge):



Welche Strategie solltest du wählen?

Wähle **Merge**, wenn...

- □□ **Mehrere Personen am selben Branch arbeiten** – Rebase würde deren Arbeit zerstören
- □□ **Du eine vollständige Audit-Trail brauchst** – z.B. in regulierten Umgebungen
- □□ **Du noch unsicher mit Git bist** – Merge ist verzeihender
- □□ **Dein Team Merge bevorzugt** – Konsistenz ist wichtiger als persönliche Vorliebe

Wähle **Rebase**, wenn...

- □□ **Du alleine am Feature-Branch arbeitest**
- □ **Du eine saubere, lineare Historie bevorzugst**
- □□ **Dein Team „Rebase before Merge" als Konvention hat**
- □□ **Du deine Commits vor dem PR aufräumen möchtest** (Interactive Rebase)

Der Kompromiss: „Rebase lokal, Merge für Integration"

Viele Teams nutzen eine **hybride Strategie**:

1. **Während der Feature-Entwicklung:** Rebase nutzen, um den Feature-Branch aktuell zu halten (solange nicht gepusht oder alleine am Branch)
2. **Für den finalen Merge:** Einen normalen Merge (oder Squash-Merge) in `main` machen

```
flowchart LR
    A["Feature-Branch\n erstellen"] --> B["Entwickeln"]
    B --> C{"Neue Commits\n in main?"}
    C -->|Ja| D["Rebase auf main\n lokal"]
    D --> B
    C -->|Nein| E{"Feature\n fertig?"}
    E -->|Nein| B
    E -->|Ja| F["Pull Request\n erstellen"]
    F --> G["Code Review"]
    G --> H["Merge PR\n in main"]

    style D fill:#fff3e0,color:#000000
    style H fill:#e8f5e9,color:#000000
```

Praktische Tipps für die tägliche Arbeit □□

Tipp 1: Synchronisiere regelmäßig

Mache es dir zur Gewohnheit, **mindestens einmal täglich** (oder bei jedem PR, der in `main` gemergt wird) zu synchronisieren:

```
# Schneller Check am Morgen
git fetch origin
git log HEAD..origin/main --oneline
# Zeigt dir, wie viele Commits du "hinterher" bist
```

In PhpStorm siehst du das im Git-Log: Wenn `origin/main` weiter ist als der Verzweigungspunkt deines Branches, ist es Zeit zu synchronisieren.

Tipp 2: Nutze PhpStorms „Update Project"

Der schnellste Weg für den täglichen Workflow:

1. `Ctrl+T` (oder **Git → Update Project**)
2. Wähle **Rebase** oder **Merge** (du kannst eine Standardeinstellung setzen)
3. PhpStorm führt Fetch + Merge/Rebase in einem Schritt aus

Tipp 3: Konfiguriere deine Standard-Strategie

In PhpStorm:

- **Settings → Version Control → Git**
- Unter „Update Method": Wähle deine bevorzugte Methode

Global in Git:

```
# Für Rebase als Standard
git config --global pull.rebase true
```

```
# Für Merge als Standard (default)
git config --global pull.rebase false
```

Tipp 4: Erstelle einen Alias für den Workflow

Wenn du häufig synchronisierst, erstelle einen Git-Alias:

```
# In ~/.gitconfig oder via Befehl:
git config --global alias.sync-main '!git fetch origin && git rebase origin/main'

# Nutzung:
git sync-main
```

Tipp 5: Kommuniziere mit deinem Team

Die wichtigste „Technik" ist eigentlich keine technische: **Sprich mit deinem Team!**

- Einigt euch auf **eine** Strategie (Merge oder Rebase)
- Dokumentiert sie in der `CONTRIBUTING.md`
- Nutzt **Branch Protection Rules**, um die Strategie durchzusetzen

---

Zusammenfassung □□

Long-Running Branches sind permanente Branches wie `main` und `develop`, die als Integrationspunkte dienen und niemals gelöscht werden.

Regelmäßige Synchronisation ist essentiell, um:

- Merge-Konflikte klein zu halten
- Gegen aktuellen Code zu entwickeln
- Integrationsprobleme früh zu erkennen

Zwei Strategien stehen zur Verfügung:

1. **Merge** (`git merge origin/main`):
   - Erzeugt Merge-Commits

- Bewahrt die echte Historie
- Sicher für Team-Branches

2. **Rebase** (`git rebase origin/main`):
   - Erzeugt lineare Historie
   - Ändert Commit-Hashes (Force-Push nötig)
   - Nur für Branches, an denen du alleine arbeitest

Die beste Strategie ist die, auf die sich dein Team einigt und die konsistent angewendet wird.

Revision #1
Created 2026-06-29 15:31:31 UTC by art10m
Updated 2026-06-29 15:31:53 UTC by art10m