Kapitel 2: Fortgeschrittenes Branching – Strategien für professionelle Entwicklung

Einleitung: Im Einsteigerkurs hast du gelernt, Branches zu erstellen und zu mergen. In der professionellen Entwicklung reicht das nicht – du brauchst eine durchdachte Branching-Strategie, die zu deinem Projekt und Team passt. Git Flow, GitHub Flow, Trunk-Based Development – jede Strategie hat ihre Vor- und Nachteile. Außerdem wirst du lernen, wie du mit Long-Running Branches umgehst, Release-Branches verwaltest und Hotfixes in mehrere Branches gleichzeitig einpflegst. Dieses Kapitel verwandelt dich vom Branch-Nutzer zum Branch-Strategen.

- Branching-Strategien im Vergleich: Git Flow, GitHub Flow und Trunk-Based Development
- Long-Running Branches verstehen und synchron halten
- Der Hotfix-Workflow: Kritische Bugfixes sicher in alle Branches bringen □□
- Protected Branches auf GitHub – Dein Sicherheitsnetz für kritische Branches

Branching-Strategien im Vergleich: Git Flow, GitHub Flow und Trunk-Based Development

Die Wahl der richtigen Branching-Strategie ist eine der wichtigsten architektonischen Entscheidungen für dein Projekt. Sie beeinflusst, wie dein Team zusammenarbeitet, wie schnell ihr Features ausliefern könnt und wie stabil eure Releases sind. Lass mich dir die drei populärsten Strategien im Detail vorstellen.

□□ Git Flow – der strukturierte Klassiker

Git Flow wurde 2010 von Vincent Driessen vorgestellt und war lange Zeit *der* Standard für professionelle Softwareentwicklung. Es ist eine ausgeklügelte, aber komplexe Strategie mit klar definierten Branch-Typen und strengen Regeln.

Die Branch-Struktur

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

```
    commit id: "Login Logic"
    checkout develop
    merge feature/login id: "Merge Login"
    branch release/1.0
    checkout release/1.0
    commit id: "Bugfix"
    commit id: "Version bump"
    checkout main
    merge release/1.0 id: "Release 1.0" tag: "v1.0"
    checkout develop
    merge release/1.0 id: "Back-merge"
```

Git Flow definiert **fünf Branch-Typen** mit jeweils spezifischen Aufgaben:

1. **`main`** (früher `master`)
   Der heilige Gral deines Repositories. Dieser Branch enthält *ausschließlich* produktionsreifen Code. Jeder Commit auf `main` repräsentiert eine Release-Version und wird typischerweise mit einem Tag versehen (z.B. `v1.0.0`, `v1.1.0`). Direktes Committen auf `main` ist streng verboten – Änderungen kommen nur durch Merges von Release- oder Hotfix-Branches.
2. **`develop`**
   Der Integrations-Branch für alle neuen Entwicklungen. Hier fließen alle abgeschlossenen Features zusammen. `develop` repräsentiert den aktuellen Entwicklungsstand und ist die Basis für neue Feature-Branches. Man könnte sagen: `main` ist „was der Kunde sieht", `develop` ist „was als nächstes kommt".
3. **`feature/*`** (z.B. `feature/user-authentication`, `feature/shopping-cart`)
   Für jedes neue Feature wird ein eigener Branch von `develop` abgezweigt. Hier findet die eigentliche Entwicklungsarbeit statt. Feature-Branches können beliebig viele Commits enthalten und existieren so lange, bis das Feature fertig ist. Nach Abschluss wird der Branch zurück in `develop` gemergt und gelöscht.
4. **`release/*`** (z.B. `release/1.2.0`)
   Wenn `develop` genügend Features für eine neue Version angesammelt hat, wird ein Release-Branch abgezweigt. Ab diesem Zeitpunkt kommen *keine neuen Features* mehr hinzu – nur noch Bugfixes, Dokumentation und Release-Vorbereitungen (Versionsnummern aktualisieren, Changelog schreiben). Nach Fertigstellung wird der Release-Branch sowohl in `main` als auch zurück in `develop` gemergt.
5. **`hotfix/*`** (z.B. `hotfix/1.0.1-security-patch`)
   Für kritische Bugs in der Produktion, die nicht auf den nächsten regulären Release warten können. Hotfix-Branches werden direkt von `main` abgezweigt, enthalten nur die minimale Änderung zur Fehlerbehebung und werden nach Fertigstellung sowohl in `main` (mit neuem Tag) als auch in `develop` gemergt.

Der typische Workflow in Git Flow

Ein neues Feature durchläuft folgenden Weg:

```
develop → feature/xyz → develop → release/1.0 → main + develop
                                        ↑
                                   (Bugfixes)
```

Und ein Hotfix:

```
main → hotfix/1.0.1 → main + develop
```

Vor- und Nachteile von Git Flow

✅ Vorteile	❌ Nachteile
Klare Trennung zwischen Entwicklung und Produktion	Hohe Komplexität durch viele Branch-Typen
Parallele Arbeit an Release und neuen Features möglich	Lange Feature-Branches führen zu schwierigen Merges
Gut geeignet für versionierte Software mit Releases	Langsamerer Entwicklungszyklus
Stabile `main`-Branch garantiert	Erfordert Disziplin und Tooling
Hotfixes können unabhängig eingespielt werden	Overkill für kleine Teams oder Web-Apps

Wann Git Flow verwenden?

Git Flow eignet sich besonders für:

- **Versionierte Software** mit klaren Release-Zyklen (z.B. Desktop-Apps, Mobile Apps, Libraries)
- **Projekte mit Support für mehrere Versionen** gleichzeitig
- **Größere Teams** (5+ Entwickler) mit spezialisierten Rollen
- **Regulierte Umgebungen**, die Nachvollziehbarkeit und Audits erfordern
- **Software mit längeren Release-Zyklen** (Wochen bis Monate)

□□ GitHub Flow – schlank und kontinuierlich

GitHub Flow wurde von GitHub selbst als Reaktion auf die Komplexität von Git Flow entwickelt. Es ist radikal einfacher: Es gibt nur `main` und Feature-Branches. Die Philosophie ist „ship early, ship

often" – kontinuierliche Auslieferung kleiner Änderungen.

Die Branch-Struktur

```
gitGraph
    commit id: "Initial"
    branch feature/login
    checkout feature/login
    commit id: "Login UI"
    commit id: "Login Tests"
    checkout main
    merge feature/login id: "PR #1" tag: "deployed"
    branch feature/dashboard
    checkout feature/dashboard
    commit id: "Dashboard"
    checkout main
    merge feature/dashboard id: "PR #2" tag: "deployed"
    branch bugfix/header
    checkout bugfix/header
    commit id: "Fix Header"
    checkout main
    merge bugfix/header id: "PR #3" tag: "deployed"
```

Die sechs Regeln von GitHub Flow

1. **`main` ist immer deploybar**
   Der `main`-Branch muss zu *jedem Zeitpunkt* in Produktion deployt werden können. Das bedeutet: Nur getesteter, funktionierender Code kommt hinein.
2. **Für jede Änderung einen Branch erstellen**
   Egal ob Feature, Bugfix oder Experiment – erstelle einen beschreibend benannten Branch von `main` (z.B. `add-user-profile`, `fix-login-timeout`, `experiment/new-checkout`).
3. **Regelmäßig committen und pushen**
   Kleine, häufige Commits mit klaren Nachrichten. Push regelmäßig zu GitHub, damit andere den Fortschritt sehen und du ein Backup hast.
4. **Pull Request erstellen, wenn bereit für Review**
   Sobald deine Änderung bereit für Feedback ist (oder du Diskussion brauchst), öffne einen Pull Request. Dies ist der zentrale Ort für Code Review und Diskussion.
5. **Nach Review und Approval: Merge in `main`**
   Sobald der PR approved ist und alle Tests grün sind, wird er in `main` gemergt. Bei GitHub Flow gibt es keine Zwischenstationen.

6. **Sofort nach dem Merge deployen**
   Der Merge in `main` triggert (idealerweise automatisch) ein Deployment. Das bedeutet, dass jeder Merge innerhalb von Minuten in Produktion ist.

Der typische Workflow in GitHub Flow

```
main → feature-branch → Pull Request → Code Review → main → Deploy
           ↑                                  ↓
      (entwickeln)                (automatische Tests)
```

Vor- und Nachteile von GitHub Flow

✅ Vorteile	❌ Nachteile
Extrem einfach zu verstehen und umzusetzen	Keine native Unterstützung für versionierte Releases
Fördert kontinuierliche Integration	Erfordert sehr gute Testabdeckung und CI/CD
Schnelle Iteration und Feedback-Zyklen	Hotfixes sind „normal" – keine spezielle Behandlung
Pull Requests als zentrale Review-Plattform	Weniger geeignet, wenn mehrere Versionen unterstützt werden müssen
Ideal für Web-Applikationen und SaaS	Kann bei großen Features unübersichtlich werden

Wann GitHub Flow verwenden?

GitHub Flow eignet sich besonders für:

- **Web-Applikationen und SaaS**, die kontinuierlich deployt werden
- **Kleine bis mittlere Teams** (1–10 Entwickler)
- **Projekte mit guter CI/CD-Pipeline** und automatisierten Tests
- **Agile Teams** mit kurzen Sprints und häufigen Releases
- **Startups und schnell iterierende Produkte**

□□ Trunk-Based Development – der radikale Ansatz

Trunk-Based Development (TBD) geht noch einen Schritt weiter als GitHub Flow. Die Idee: Alle Entwickler committen direkt in einen einzigen Branch (den „Trunk", typischerweise `main`). Feature-Branches existieren entweder gar nicht oder sind extrem kurzlebig (maximal 1–2 Tage).

Die Branch-Struktur

```
gitGraph
    commit id: "Feature A - Part 1"
    commit id: "Feature A - Part 2"
    commit id: "Bugfix"
    commit id: "Feature B" tag: "deployed"
    commit id: "Refactoring"
    commit id: "Feature A - Part 3"
    commit id: "Feature A complete" tag: "deployed"
```

Die Kernprinzipien

1. **Ein einziger Branch für alle**
   Es gibt nur `main` (den „Trunk"). Alle Entwickler integrieren ihre Änderungen hier. Lange lebende Feature-Branches sind *verboten*.
2. **Sehr kleine, sehr häufige Commits**
   Statt eines großen Commits am Ende eines Features gibt es viele kleine Commits während der Entwicklung. Jeder Commit muss den Build grün halten.
3. **Feature Flags für unfertige Features**
   Wie bringt man Code für ein halbfertiges Feature in `main`, ohne dass Nutzer es sehen? Durch **Feature Flags** (auch Feature Toggles genannt):

   ```
   if ($featureFlags->isEnabled('new-checkout-process')) {
       return $this->newCheckoutProcess($cart);
   } else {
       return $this->legacyCheckout($cart);
   }
   ```

   Das Feature wird erst „eingeschaltet", wenn es fertig ist – obwohl der Code schon lange in Produktion ist.
4. **Kurzlebige Feature-Branches (optional)**
   Manche Teams erlauben Feature-Branches, aber mit strikter Regel: Sie dürfen maximal 1–2 Tage existieren und müssen dann gemergt werden. Das erzwingt kleine, inkrementelle Änderungen.
5. **Exzellente CI/CD ist Pflicht**

> Da jeder Commit potenziell in Produktion geht, muss die Test-Pipeline *schnell* und *zuverlässig* sein. Typischerweise < 10 Minuten für den kompletten Build.

Branch by Abstraction

Für größere Refactorings, die nicht in 1–2 Tagen erledigt sind, nutzt TBD das Pattern „**Branch by Abstraction**":

1. Erstelle eine Abstraktionsschicht (Interface) vor dem zu ändernden Code
2. Implementiere die neue Version hinter dem Interface
3. Schalte schrittweise um (Feature Flag oder schrittweise Migration)
4. Entferne die alte Implementierung, sobald die neue stabil ist

So bleibt der Code immer lauffähig, obwohl du parallel zwei Implementierungen hast.

Vor- und Nachteile von Trunk-Based Development

✅ Vorteile	❌ Nachteile
Keine Merge-Hölle durch lang lebende Branches	Erfordert erfahrene, disziplinierte Entwickler
Maximale Kontinuierliche Integration	Feature Flags erhöhen Code-Komplexität
Sehr schnelles Feedback	Sehr gute Testabdeckung ist Voraussetzung
Fördert kleine, fokussierte Änderungen	Kann für Junior-Entwickler überfordernd sein
Von Google, Facebook und anderen Tech-Giganten verwendet	Weniger Review-Möglichkeiten vor dem Merge

Wann Trunk-Based Development verwenden?

Trunk-Based Development eignet sich besonders für:

- **Erfahrene, hochperformante Teams** mit starker Ingenieurskultur
- **Unternehmen mit exzellenter CI/CD-Infrastruktur**
- **Projekte, die mehrmals täglich deployen** wollen
- **Teams, die Pair Programming praktizieren** (ersetzt Code Review)
- **Microservices-Architekturen** mit kleinen, fokussierten Repositories

---

□□ Der große Vergleich

Aspekt	Git Flow	GitHub Flow	Trunk-Based
Komplexität	Hoch	Niedrig	Mittel*
Branch-Typen	5 (main, develop, feature, release, hotfix)	2 (main, feature)	1 (main)
Lebensdauer Feature-Branch	Tage bis Wochen	Stunden bis Tage	Keine oder < 2 Tage
Release-Modell	Geplante Versionen	Kontinuierlich	Kontinuierlich
Merge-Frequenz	Selten, dafür größer	Häufig	Sehr häufig (mehrmals täglich)
CI/CD-Anforderung	Optional	Empfohlen	Pflicht
Feature Flags nötig?	Nein	Selten	Ja
Parallele Versionen	Ja	Nein	Nein
Team-Größe	Mittel bis groß	Klein bis mittel	Klein bis mittel (erfahren)
Lernkurve	Steil	Flach	Mittel

**Die Komplexität bei TBD liegt nicht im Branching, sondern in den Praktiken (Feature Flags, kleine Commits, CI/CD).*

□□ Meine Empfehlung für dein PHP-Webprojekt

Du beschreibst ein **mittelgroßes PHP-Webprojekt mit regelmäßigen Releases**. Lass mich die Faktoren analysieren:

Analyse deiner Situation

- **„Mittelgroß"** deutet auf ein Team von 2–8 Entwicklern hin
- **„PHP-Webprojekt"** ist typischerweise eine Web-Applikation, die deployt wird (nicht installierte Software)
- **„Regelmäßige Releases"** können unterschiedlich interpretiert werden:
  - Wöchentlich/zweiwöchentlich → GitHub Flow geeignet
  - Monatlich mit festen Versionen → Git Flow könnte passen

    - Kontinuierlich → Trunk-Based möglich

Meine Empfehlung: **GitHub Flow mit Release-Tags** □□

Für dein Szenario empfehle ich **GitHub Flow**, aber mit einer kleinen Erweiterung für die Versionierung:

```
flowchart LR
    subgraph Entwicklung
        A["Feature-Branch\nerstellen"] --> B["Entwickeln\nund testen"]
        B --> C["Pull Request\nerstellen"]
        C --> D["Code Review"]
        D --> E["Merge in main"]
    end

    subgraph Release
        E --> F{"Release\nfällig?"}
        F -->|Ja| G["Tag erstellen\nv1.2.0"]
        F -->|Nein| H["Weiter entwickeln"]
        G --> I["Deployment"]
    end

    style G fill:#90EE90,color:#000000
    style I fill:#87CEEB,color:#000000
```

Warum GitHub Flow für dich?

1. **Einfachheit:** Du und dein Team müsst nicht fünf verschiedene Branch-Typen jonglieren. Die Regeln passen auf einen Bierdeckel.
2. **Web-Applikation:** PHP-Webprojekte werden deployt, nicht installiert. Du brauchst keine parallelen Versionen zu unterstützen – es gibt nur „was gerade live ist".
3. **Pull Requests als Qualitätssicherung:** Jede Änderung durchläuft einen Review-Prozess. Das ist Gold wert für die Code-Qualität.
4. **Flexibilität bei Releases:** Du kannst so oft oder selten releasen, wie du möchtest. Ein Release ist einfach ein Tag auf `main`.
5. **Einfacher Einstieg:** Falls Teammitglieder noch nicht so Git-erfahren sind, ist GitHub Flow viel leichter zu erlernen als Git Flow.

Wie du „regelmäßige Releases" mit GitHub Flow umsetzt

Anstatt Release-Branches zu nutzen, arbeitest du mit **Tags** und einem **Release-Rhythmus**:

```
# Nach dem Merge aller Features für Version 1.2.0
git tag -a v1.2.0 -m "Release 1.2.0: User Dashboard, Performance Improvements"
git push origin v1.2.0
```

In GitHub kannst du dann einen formalen Release erstellen, der auf diesen Tag verweist – mit Changelog, Release Notes und ggf. Assets.

Dein angepasster Workflow

1. **Für jede Aufgabe:** Erstelle einen Branch von `main`

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

2. **Entwickle und committe** regelmäßig mit aussagekräftigen Nachrichten
3. **Erstelle einen Pull Request** auf GitHub, wenn bereit für Review
4. **Nach Approval:** Merge in `main` (ich empfehle „Squash and Merge" für eine saubere Historie)
5. **Für Releases:** Wenn genügend Features gemergt sind, erstelle einen Tag und deploye:

   ```
   git checkout main
   git pull
   git tag -a v1.2.0 -m "Release 1.2.0"
   git push origin v1.2.0
   # Deployment triggern (manuell oder automatisch via GitHub Actions)
   ```

6. **Für Hotfixes:** Behandle sie wie normale Features – Branch erstellen, fixen, PR, mergen. Danach ggf. Patch-Version taggen (`v1.2.1`).

Wann du doch Git Flow in Betracht ziehen solltest

Überdenke meine Empfehlung, wenn *einer* dieser Punkte zutrifft:

- Du musst **mehrere Versionen parallel unterstützen** (z.B. v1.x und v2.x gleichzeitig pflegen)
- Du hast **sehr lange Release-Zyklen** (> 1 Monat) mit umfangreichen QA-Phasen
- Du arbeitest in einer **regulierten Branche** (Medizin, Finanzen) mit strengen Audit-Anforderungen
- Dein Team ist **sehr groß** (> 10 Entwickler) und braucht klare Abgrenzungen

Wann Trunk-Based Development interessant wäre

- Du möchtest **mehrmals täglich deployen**
- Dein Team ist **sehr erfahren** und praktiziert Pair Programming
- Du hast eine **exzellente CI/CD-Pipeline** mit < 10 Minuten Build-Zeit
- Du bist bereit, in **Feature Flags** zu investieren

□□ Zusammenfassung

Strategie	Motto	Ideal für
Git Flow	„Struktur und Kontrolle"	Versionierte Software, große Teams, lange Zyklen
GitHub Flow	„Ship early, ship often"	Web-Apps, kleine-mittlere Teams, kontinuierliche Delivery
Trunk-Based	„Immer integriert"	Hochperformante Teams, mehrfach tägliches Deployment

Für dein mittelgroßes PHP-Webprojekt ist **GitHub Flow** der Sweet Spot: Einfach genug, um schnell produktiv zu sein, aber strukturiert genug für professionelle Zusammenarbeit. Mit Tags und GitHub Releases bekommst du die Versionierung, die du für „regelmäßige Releases" brauchst – ohne die Komplexität von Git Flow.

Starte mit GitHub Flow, und wenn du merkst, dass du mehr Struktur brauchst, kannst du immer noch zu Git Flow wechseln. Der umgekehrte Weg (von komplex zu einfach) ist erfahrungsgemäß schwieriger! □□

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
```

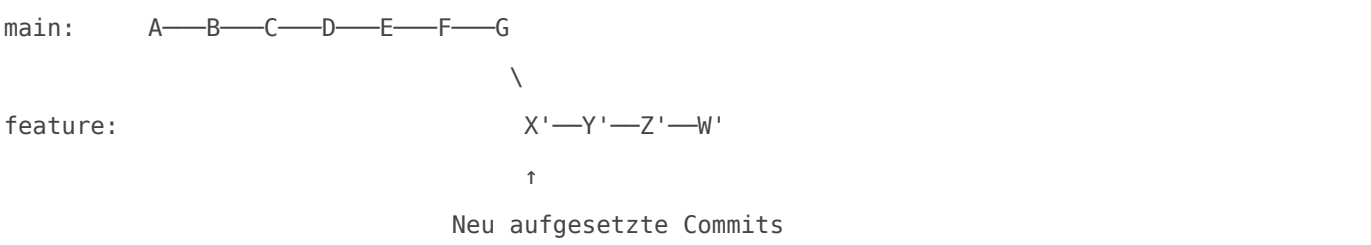
Merge vs. Rebase: Direkter Vergleich

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

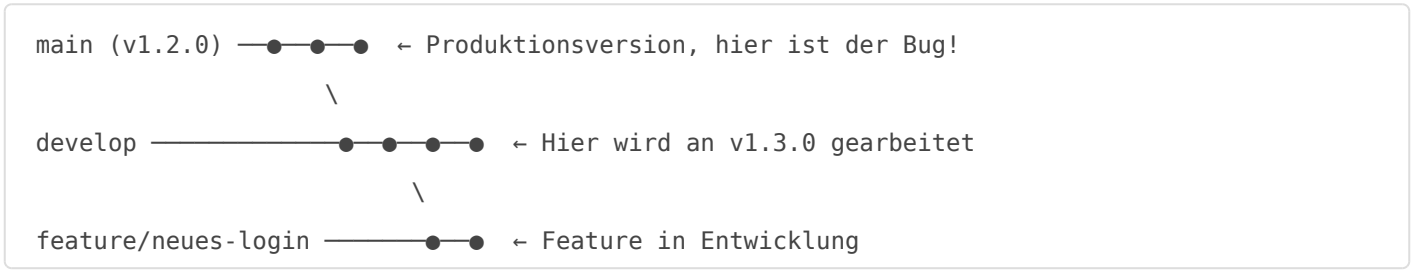
Die beste Strategie ist die, auf die sich dein Team einigt und die konsistent angewendet wird.

Der Hotfix-Workflow: Kritische Bugfixes sicher in alle Branches bringen □□

Ein Hotfix ist eine der stressigsten Situationen in der Softwareentwicklung: Ein kritischer Bug ist in der Produktion aufgetaucht und muss *sofort* behoben werden – ohne dabei die laufende Entwicklungsarbeit zu gefährden oder den Fix irgendwo zu „vergessen". In dieser Lektion zeige ich dir, wie du einen professionellen Hotfix-Workflow umsetzt und dabei sicherstellst, dass der Fix in allen relevanten Branches ankommt.

Das Grundproblem verstehen

Stell dir folgendes Szenario vor:



Der Bug existiert in `main` (der Produktionsversion), aber auch in `develop` und allen Feature-Branches, die von `main` oder `develop` abgezweigt wurden. Wenn du den Fix nur in `main` einspielst, wird er bei der nächsten Entwicklung wieder überschrieben. Spielst du ihn nur in `develop` ein, dauert es bis zum nächsten Release, bis er in Produktion kommt.

Die Lösung: Ein dedizierter Hotfix-Branch, der von `main` abzweigt und nach der Behebung in *alle* relevanten Branches gemergt wird.

Der klassische Hotfix-Workflow (Schritt für Schritt)

1. Hotfix-Branch von `main` erstellen

Der Hotfix-Branch wird **immer von** `main` (oder dem Branch, der die Produktion repräsentiert) erstellt – niemals von `develop`! So stellst du sicher, dass du exakt den Code-Stand der Produktion als Basis hast.

In PhpStorm:

1. Öffne das **Git-Tool-Window** (`Alt+9` / `Cmd+9`)
2. Stelle sicher, dass du auf `main` bist (Rechtsklick auf `main` → *Checkout*)
3. Führe einen **Pull** durch, um sicherzustellen, dass du den aktuellen Stand hast
4. Klicke auf **New Branch** (oder `Ctrl+Shift+`` / `Cmd+Shift+``)
5. Benenne den Branch nach dem Schema: `hotfix/kurze-beschreibung` oder `hotfix/v1.2.1`

Auf der Kommandozeile:

```
# Sicherstellen, dass main aktuell ist
git checkout main
git pull origin main

# Hotfix-Branch erstellen
git checkout -b hotfix/kritischer-login-bug
```

Wichtige Namenskonventionen:

- `hotfix/beschreibung` – z.B. `hotfix/sql-injection-fix`
- `hotfix/vX.Y.Z` – z.B. `hotfix/v1.2.1` (wenn du semantische Versionierung nutzt)
- `hotfix/issue-123` – wenn du ein Issue-Tracking-System verwendest

---

2. Den Bug beheben und committen

Jetzt behebst du den Bug. Dabei gelten einige wichtige Regeln:

Nur den Bug fixen – nichts anderes! Ein Hotfix sollte so minimal wie möglich sein. Jede zusätzliche Änderung erhöht das Risiko, neue Probleme einzuführen. Widerstehe der Versuchung,

„schnell noch" andere kleine Dinge zu korrigieren.

Commit-Message mit Kontext:

```
git add src/Auth/LoginController.php
git commit -m "fix: SQL-Injection-Schwachstelle im Login behoben

- Prepared Statements statt String-Konkatenation
- Betrifft Login und Passwort-Reset
- Fixes #247"
```

In PhpStorm:

1. Mache deine Änderungen im Code
2. Öffne das **Commit-Tool-Window** (`Ctrl+K` / `Cmd+K`)
3. Wähle nur die relevanten Dateien aus
4. Schreibe eine aussagekräftige Commit-Message
5. Nutze optional *Amend* wenn du nachbessern musst

---

3. Hotfix testen

Bevor du den Hotfix irgendwohin mergst, **teste ihn gründlich**:

- Führe deine automatisierten Tests aus
- Teste den spezifischen Fix manuell
- Wenn möglich, deploye auf eine Staging-Umgebung

In PhpStorm kannst du Tests direkt ausführen:

- Rechtsklick auf den Test-Ordner → *Run Tests*
- Oder nutze die Run-Konfiguration für PHPUnit

---

4. Hotfix in `main` mergen und taggen

Jetzt kommt der kritische Teil: Der Fix muss zurück nach `main`, damit er in die Produktion deployed werden kann.

In PhpStorm:

1. Wechsle zu `main` (Rechtsklick → *Checkout*)
2. Rechtsklick auf deinen Hotfix-Branch → *Merge into Current*

3. PhpStorm zeigt dir den Merge-Dialog – überprüfe die Änderungen
4. Bestätige den Merge

Auf der Kommandozeile:

```
# Zu main wechseln
git checkout main

# Hotfix mergen (mit --no-ff für einen expliziten Merge-Commit)
git merge --no-ff hotfix/kritischer-login-bug -m "Merge hotfix/kritischer-login-bug: SQL-Injection behoben"
```

Warum `--no-ff`? Die Option `--no-ff` (*no fast-forward*) erzwingt einen Merge-Commit, auch wenn ein Fast-Forward möglich wäre. Das hat zwei Vorteile:

1. Der Hotfix bleibt in der Historie als eigenständiger „Block" erkennbar
2. Du hast einen klaren Merge-Commit, der dokumentiert, wann der Hotfix eingespielt wurde

Version-Tag erstellen:

Nach einem Hotfix solltest du einen neuen Version-Tag erstellen:

```
# Tag erstellen
git tag -a v1.2.1 -m "Hotfix: SQL-Injection-Schwachstelle behoben"

# Tag auf GitHub pushen
git push origin v1.2.1
```

In PhpStorm:

- Menü: *Git* → *New Tag*
- Oder im Git-Log: Rechtsklick auf den Commit → *New Tag*

---

5. Hotfix in `develop` mergen ⚠

Das ist der Schritt, der am häufigsten vergessen wird! Wenn du den Hotfix nicht auch in `develop` mergst, wird der Bug beim nächsten Release wieder auftauchen, weil `develop` dann ohne den Fix nach `main` gemergt wird.

In PhpStorm:

1. Wechsle zu `develop` (Rechtsklick → *Checkout*)

2. Rechtsklick auf `main` (oder den Hotfix-Branch) → *Merge into Current*
3. Löse eventuelle Konflikte im Merge-Tool

Auf der Kommandozeile:

```
git checkout develop
git pull origin develop  # Sicherstellen, dass develop aktuell ist
git merge --no-ff hotfix/kritischer-login-bug -m "Merge hotfix in develop: SQL-Injection behoben"
```

Oder alternativ – den Tag mergen:

```
git checkout develop
git merge --no-ff v1.2.1 -m "Merge v1.2.1 hotfix in develop"
```

---

6. Alles pushen

Jetzt müssen alle Änderungen auf GitHub landen:

```
# main mit dem Fix pushen
git push origin main

# develop mit dem Fix pushen
git push origin develop

# Tags pushen (falls noch nicht geschehen)
git push origin --tags
```

In PhpStorm:

- `Ctrl+Shift+K` / `Cmd+Shift+K` öffnet den Push-Dialog
- Stelle sicher, dass *Push Tags* aktiviert ist

---

7. Hotfix-Branch aufräumen

Nach erfolgreichem Merge in beide Branches kann der Hotfix-Branch gelöscht werden:

Lokal löschen:

```
git branch -d hotfix/kritischer-login-bug
```

Auf GitHub löschen:

```
git push origin --delete hotfix/kritischer-login-bug
```

In PhpStorm:

- Im Git-Tool-Window: Rechtsklick auf den Branch → *Delete*
- Häkchen setzen bei *Delete Tracking Branch* um ihn auch remote zu löschen

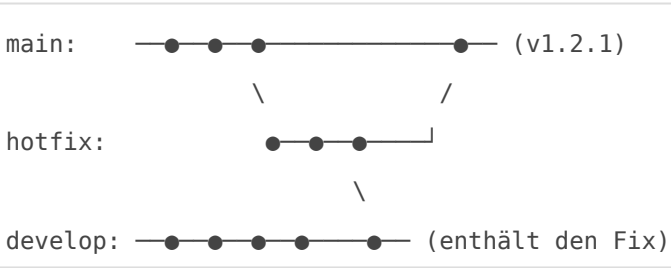
---

Visualisierung des Workflows

```
flowchart TD
    subgraph Ausgangssituation
        A["main\n v1.2.0 - Bug vorhanden"]
        B["develop\n Aktive Entwicklung"]
        A -.->|"abgezweigt"| B
    end

    subgraph Hotfix-Prozess
        C["1. Hotfix-Branch erstellen\n von main"]
        D["2. Bug beheben\n und committen"]
        E["3. Testen"]
        F["4. Merge in main\n Tag: v1.2.1"]
        G["5. Merge in develop"]
        H["6. Push und Cleanup"]
    end

    A --> C
    C --> D
    D --> E
    E --> F
    F --> G
    G --> H

    style C fill:#ffcccc,color:#000000
    style F fill:#ccffcc,color:#000000
```

```
    style G fill:#ccffcc,color:#000000
```

Die Commit-Historie nach dem Hotfix:



Umgang mit Konflikten beim Merge in `develop`

Es ist sehr wahrscheinlich, dass beim Merge des Hotfixes in `develop` Konflikte auftreten – schließlich wurde in `develop` weiterentwickelt, während der Hotfix auf dem älteren `main`-Stand basiert.

Typisches Konflikt-Szenario:

Der Hotfix hat eine Funktion in `LoginController.php` geändert, aber in `develop` wurde dieselbe Funktion für ein neues Feature erweitert.

Konfliktlösung in PhpStorm:

1. Nach dem Merge-Versuch zeigt PhpStorm die konfligierenden Dateien an
2. Doppelklick auf eine Datei öffnet den **3-Way-Merge-Editor**
3. Du siehst drei Spalten:
   - **Links (Yours):** Der aktuelle Stand von `develop`
   - **Rechts (Theirs):** Der Hotfix
   - **Mitte (Result):** Das gewünschte Ergebnis
4. Übernimm die Sicherheitsänderungen aus dem Hotfix
5. Behalte die neuen Features aus `develop`
6. Stelle sicher, dass beides zusammen funktioniert

Wichtig: Nach dem Lösen von Konflikten immer testen! Ein schlecht gelöster Konflikt kann den Fix unwirksam machen.

Was ist mit Feature-Branches?

Wenn zum Zeitpunkt des Hotfixes Feature-Branches existieren, die von `develop` (oder `main`) abgezweigt wurden, enthalten auch diese den Bug. Hier gibt es mehrere Strategien:

Option A: Feature-Branches auf `develop` rebasen

Nachdem der Hotfix in `develop` gemergt wurde, können Feature-Branch-Entwickler ihren Branch auf den neuen `develop`-Stand rebasen:

```
git checkout feature/neues-login
git fetch origin
git rebase origin/develop
```

Vorteil: Der Feature-Branch enthält automatisch den Fix.
Nachteil: Erfordert einen Force-Push, wenn der Branch bereits gepusht wurde.

Option B: `develop` in Feature-Branches mergen

Alternativ können die Feature-Branch-Entwickler `develop` in ihren Branch mergen:

```
git checkout feature/neues-login
git fetch origin
git merge origin/develop
```

Vorteil: Kein Force-Push nötig.
Nachteil: Zusätzliche Merge-Commits.

Option C: Nichts tun und beim PR lösen

Wenn der Feature-Branch ohnehin bald gemergt wird, kann der Konflikt auch beim finalen Merge/PR gelöst werden. Der Hotfix landet dann automatisch im Feature-Branch, sobald dieser nach `develop` gemergt wurde.

Empfehlung: Für sicherheitskritische Hotfixes ist Option A oder B besser, damit alle aktiv entwickelten Branches sofort geschützt sind.

Der Hotfix-Workflow mit Pull Requests

In professionellen Teams wird der Hotfix-Workflow oft mit Pull Requests kombiniert, um Code Reviews auch für kritische Fixes sicherzustellen:

Workflow mit PRs

1. **Hotfix-Branch erstellen und pushen:**

   ```
   git checkout -b hotfix/kritischer-bug main
   # Fix implementieren
   git push -u origin hotfix/kritischer-bug
   ```

2. **Ersten PR erstellen: Hotfix → main**
   - Auf GitHub: *New Pull Request*
   - Base: `main`, Compare: `hotfix/kritischer-bug`
   - Als *Critical* oder *Urgent* markieren
   - Reviewer zuweisen (idealerweise jemand, der sofort verfügbar ist)
3. **Review und Merge in main**
   - Schnelles, fokussiertes Review
   - Nach Approval: **Merge** (nicht Squash, damit der Commit-Hash erhalten bleibt)
   - Tag erstellen
4. **Zweiten PR erstellen: main → develop** (oder Hotfix → develop)
   - Base: `develop`, Compare: `main`
   - Dieser PR dokumentiert, dass der Hotfix auch in `develop` übernommen wurde

Automatisierung mit GitHub Actions

Du kannst einen GitHub Actions Workflow erstellen, der automatisch einen PR von `main` nach `develop` erstellt, sobald ein Hotfix-Tag gepusht wird:

```
name: Create Hotfix Backport PR

on:
```

```
  push:
    tags:
      - 'v*.*.*'  # Triggert bei Version-Tags

jobs:
  create-backport-pr:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Create Pull Request to develop
        uses: peter-evans/create-pull-request@v5
        with:
          token: ${{ secrets.GITHUB_TOKEN }}
          branch: backport/${{ github.ref_name }}-to-develop
          base: develop
          title: "Backport ${{ github.ref_name }} to develop"
          body: |
            Automatischer Backport-PR für Hotfix ${{ github.ref_name }}.

            **Bitte prüfen und mergen, damit der Fix in develop landet!**
          labels: hotfix, backport
```

Checkliste für Hotfixes

Hier ist eine praktische Checkliste, die du bei jedem Hotfix abarbeiten solltest:

Vor dem Hotfix

- ☐ Bug ist verifiziert und reproduzierbar
- ☐ Priorität ist klar (ist es wirklich ein *kritischer* Hotfix?)
- ☐ Alle Stakeholder sind informiert

Während des Hotfixes

- [ ] Hotfix-Branch von `main` erstellt (nicht von `develop`!)
- [ ] Nur den Bug behoben, keine anderen Änderungen
- [ ] Aussagekräftige Commit-Message mit Issue-Referenz
- [ ] Tests geschrieben oder angepasst
- [ ] Alle Tests laufen durch

Nach dem Hotfix

- [ ] In `main` gemergt
- [ ] Version-Tag erstellt
- [ ] **In `develop` gemergt** ← *Wird am häufigsten vergessen!*
- [ ] Alle Branches gepusht
- [ ] Tag gepusht
- [ ] Hotfix-Branch gelöscht (lokal und remote)
- [ ] Deployment in Produktion durchgeführt
- [ ] Fix in Produktion verifiziert
- [ ] Team informiert (welche Feature-Branches sollten aktualisiert werden?)

Häufige Fehler und wie du sie vermeidest

Fehler 1: Hotfix von `develop` statt `main` erstellen

Problem: Du erstellst den Hotfix von `develop`, das bereits neue, ungetestete Features enthält. Wenn du jetzt nach `main` mergst, landen diese Features ungewollt in der Produktion.

Lösung: Immer zuerst `git checkout main` und dann den Hotfix-Branch erstellen.

Fehler 2: Vergessen, den Hotfix in `develop` zu mergen

Problem: Der Fix ist in Produktion, aber bei der nächsten großen Release-Integration wird der Bug wieder eingeführt, weil `develop` den Fix nicht enthält.

Lösung: Nutze die Checkliste oder automatisiere den Backport mit GitHub Actions.

Fehler 3: Zu viele Änderungen im Hotfix

Problem: Du nutzt den Hotfix als Gelegenheit, „schnell noch" andere Dinge zu fixen. Der Hotfix wird groß und schwer zu reviewen, das Risiko für neue Bugs steigt.

Lösung: Disziplin! Andere Fixes kommen in reguläre Feature-Branches. Ein Hotfix ist *nur* für den kritischen Bug.

Fehler 4: Keinen Tag erstellen

Problem: Ohne Tag ist später schwer nachzuvollziehen, welcher exakte Code-Stand in Produktion deployed wurde.

Lösung: Immer einen semantischen Version-Tag erstellen (`v1.2.1` für einen Hotfix auf `v1.2.0`).

Hotfixes in PhpStorm – Die wichtigsten Shortcuts

Aktion	Windows/Linux	macOS
Git-Tool-Window öffnen	`Alt+9`	`Cmd+9`
Neuen Branch erstellen	`Ctrl+Shift+```	`Cmd+Shift+```
Commit-Dialog öffnen	`Ctrl+K`	`Cmd+K`
Push-Dialog öffnen	`Ctrl+Shift+K`	`Cmd+Shift+K`
Pull (Update Project)	`Ctrl+T`	`Cmd+T`
Branches anzeigen	`Ctrl+Shift+```	`Cmd+Shift+```

Aktion	Windows/Linux	macOS
Git-Log anzeigen	`Alt+9`, dann Tab *Log*	`Cmd+9`, dann Tab *Log*

Zusammenfassung

Der Hotfix-Workflow folgt einem klaren Muster:

1. **Branch von** `main` – Nicht von `develop`!
2. **Minimal fixen** – Nur den Bug, nichts anderes
3. **Testen** – Automatisiert und manuell
4. **Merge in** `main` – Mit `--no-ff` und Version-Tag
5. **Merge in** `develop` – Der kritische Schritt, der oft vergessen wird
6. **Aufräumen** – Branch löschen, Team informieren

Mit diesem Workflow stellst du sicher, dass kritische Bugfixes schnell in die Produktion gelangen *und* in allen Entwicklungszweigen ankommen, sodass der Bug nicht bei der nächsten Release-Integration wieder auftaucht.

Protected Branches auf GitHub – Dein Sicherheitsnetz für kritische Branches

Protected Branches sind eines der wichtigsten Features, um die Integrität deines Codes zu schützen. Sie verhindern, dass versehentlich (oder absichtlich) schädliche Änderungen direkt in wichtige Branches wie `main` oder `develop` gelangen. Stell dir Protected Branches als **Türsteher** vor, die genau prüfen, wer unter welchen Bedingungen Änderungen einbringen darf.

Was sind Protected Branches? □□

Ein Protected Branch ist ein Branch, für den du auf GitHub spezielle Regeln definierst, die eingehalten werden müssen, bevor Änderungen akzeptiert werden. Ohne Schutzregeln kann *jeder* mit Schreibrechten direkt auf `main` pushen – ein einziger falscher Befehl wie `git push --force origin main` könnte die gesamte Projekthistorie zerstören.

Mit Protected Branches kannst du unter anderem festlegen:

- **Wer** überhaupt auf den Branch pushen darf
- **Ob** direkte Pushes erlaubt sind oder nur über Pull Requests
- **Welche Prüfungen** (Tests, Reviews) bestanden sein müssen
- **Ob** die Branch-Historie überschrieben werden darf (Force Push)

Warum sind sie so wichtig?

Szenario ohne Schutz	Mögliche Konsequenz
Entwickler pusht ungetesteten Code direkt auf `main`	Produktionssystem fällt aus
Jemand führt versehentlich `git push --force` aus	Commit-Historie geht verloren

Szenario ohne Schutz	Mögliche Konsequenz
Merge ohne Code Review	Bugs und Sicherheitslücken gelangen unbemerkt in die Produktion
Commits von unverifizierten Accounts	Supply-Chain-Angriffe werden möglich

---

Die wichtigsten Schutzregeln im Detail

GitHub bietet eine Vielzahl von Schutzregeln, die du kombinieren kannst. Hier sind die wichtigsten mit Erklärungen und Empfehlungen:

1. „Require a pull request before merging"

Diese Regel ist das **Herzstück** der Branch Protection. Sie verhindert, dass irgendjemand direkt auf den geschützten Branch pushen kann. Alle Änderungen *müssen* über einen Pull Request laufen.

Unteroptionen:

- **„Require approvals"** – Legt fest, wie viele Personen den PR genehmigen müssen, bevor er gemerged werden kann. Du kannst 1 bis 6 erforderliche Approvals einstellen.
  Empfehlung: Für kleine Teams (1–3 Entwickler) reicht **1 Approval**. Für größere Teams oder kritische Projekte empfehle ich **2 Approvals**.
- **„Dismiss stale pull request approvals when new commits are pushed"** – Wenn diese Option aktiviert ist, werden bestehende Approvals *ungültig*, sobald neue Commits zum PR hinzugefügt werden. Das ist wichtig, weil ein Reviewer vielleicht Code genehmigt hat, der danach noch verändert wurde.
  Empfehlung: **Unbedingt aktivieren!** Sonst könnte jemand nach dem Approval noch problematischen Code hinzufügen.
- **„Require review from Code Owners"** – Wenn dein Repository eine `CODEOWNERS`-Datei hat, müssen die dort definierten Verantwortlichen den PR genehmigen. Mehr dazu später.
- **„Require approval of the most recent reviewable push"** – Verhindert, dass der Autor des letzten Commits seinen eigenen Code genehmigt (relevant, wenn Maintainer selbst zum PR beitragen).

2. „Require status checks to pass before merging"

Diese Regel stellt sicher, dass automatisierte Prüfungen (wie Tests oder Linting) erfolgreich durchlaufen müssen, bevor ein Merge möglich ist. Das ist dein **Qualitätstor**.

Unteroptionen:

- **„Require branches to be up to date before merging"** – Der Feature-Branch muss auf dem aktuellen Stand des Ziel-Branches sein. Das verhindert, dass Code gemerged wird, der zwar mit einer *alten* Version von `main` funktioniert, aber mit den neuesten Änderungen kollidieren könnte.
  Empfehlung: **Aktivieren**, auch wenn es manchmal nervig ist, den Branch aktualisieren zu müssen. Es verhindert das „aber auf meinem Branch hat es funktioniert"-Problem.
- **Status Checks auswählen** – Du wählst aus, welche Checks bestanden sein müssen. Diese erscheinen erst in der Liste, nachdem sie mindestens einmal gelaufen sind (z.B. durch einen GitHub Actions Workflow).

```
# Beispiel: Workflow, der als Required Check verwendet werden kann
name: Tests
on: [push, pull_request]
jobs:
  phpunit:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Run PHPUnit
        run: vendor/bin/phpunit
```

3. „Require conversation resolution before merging"

Wenn Reviewer Kommentare oder Änderungswünsche hinterlassen, müssen diese als „resolved" markiert werden, bevor der PR gemerged werden kann. Das stellt sicher, dass **kein Feedback ignoriert** wird.

Empfehlung: **Aktivieren.** Es ist frustrierend als Reviewer, wenn deine Anmerkungen einfach übergangen werden.

4. „Require signed commits"

Mit dieser Regel müssen alle Commits kryptografisch signiert sein (GPG, SSH oder S/MIME). Signierte Commits zeigen auf GitHub ein grünes „Verified"-Badge und beweisen, dass der Commit wirklich von der angegebenen Person stammt.

Empfehlung: Für Open-Source-Projekte oder sicherheitskritische Anwendungen **empfohlen**. Für kleine, interne Projekte optional. Die Einrichtung erfordert etwas Aufwand (siehe Kapitel 10 des Kurses).

5. „Require linear history"

Diese Regel erzwingt eine lineare Commit-Historie, indem sie *nur* Squash-Merges oder Rebase-Merges erlaubt – keine regulären Merge-Commits mit zwei Parents.

Empfehlung: **Geschmackssache.** Eine lineare Historie ist übersichtlicher, aber manche Teams bevorzugen explizite Merge-Commits, weil sie zeigen, wann welcher Feature-Branch integriert wurde. Ich empfehle es für Projekte, die eine saubere, leicht lesbare Historie priorisieren.

6. „Do not allow bypassing the above settings"

Normalerweise können Repository-Administratoren alle Schutzregeln umgehen. Mit dieser Option wird **auch Admins** das Umgehen verboten.

Empfehlung: Für die meisten Projekte **nicht aktivieren**, da Admins manchmal legitime Gründe haben, Regeln zu umgehen (z.B. bei Notfall-Hotfixes). Bei sehr kritischen Projekten oder aus Compliance-Gründen kann es sinnvoll sein.

7. „Restrict who can push to matching branches"

Du kannst explizit definieren, welche Personen, Teams oder Apps überhaupt auf den Branch pushen dürfen (selbst über PRs). Das ist nützlich, wenn du z.B. nur bestimmten Maintainern erlauben möchtest, PRs zu mergen.

8. „Allow force pushes" und „Allow deletions"

Diese Optionen sind standardmäßig **deaktiviert** bei Protected Branches – und das sollte auch so bleiben!

- **Force Pushes** können die gesamte Commit-Historie überschreiben

- **Deletions** würden erlauben, den Branch komplett zu löschen

Empfehlung: **Niemals aktivieren** für `main` oder andere kritische Branches.

Schritt-für-Schritt-Anleitung: Branch Protection einrichten

So richtest du die Schutzregeln für deinen `main`-Branch ein:

1. **Navigiere zu den Repository-Einstellungen**
   Öffne dein Repository auf GitHub und klicke auf **„Settings"** (Zahnrad-Symbol) in der oberen Navigationsleiste.
2. **Öffne die Branch-Einstellungen**
   In der linken Seitenleiste findest du unter „Code and automation" den Punkt **„Branches"**. Klicke darauf.
3. **Füge eine Branch Protection Rule hinzu**
   Unter „Branch protection rules" klickst du auf **„Add branch protection rule"** (oder „Add rule" bei neueren Rulesets).
4. **Definiere das Branch-Muster**
   Im Feld „Branch name pattern" gibst du den Namen des zu schützenden Branches ein. Du kannst:
   - Einen exakten Namen eingeben: `main`
   - Wildcards verwenden: `release/*` schützt alle Branches, die mit „release/" beginnen
5. **Wähle deine Schutzregeln**
   Aktiviere die gewünschten Optionen (siehe detaillierte Beschreibungen oben).
6. **Speichere die Regel**
   Scrolle nach unten und klicke auf **„Create"** oder **„Save changes"**.

Empfohlene Konfiguration für verschiedene Szenarien

□□□ Solo-Entwickler (persönliches Projekt)

Auch als Solo-Entwickler können Protected Branches sinnvoll sein – sie schützen dich vor dir selbst!

Regel	Empfehlung
Require pull request	☐ Optional (kann Workflow verlangsamen)
Require status checks	☐ Aktivieren, wenn du CI/CD hast
Require signed commits	☐ Optional
Allow force pushes	☐ Deaktiviert lassen
Allow deletions	☐ Deaktiviert lassen

Als Minimalkonfiguration empfehle ich: **Keine Force Pushes, keine Deletions, Status Checks falls vorhanden.**

□□ Kleines Team (2–5 Entwickler)

Regel	Empfehlung
Require pull request	☐ Aktivieren
Require approvals	☐ 1 Approval
Dismiss stale approvals	☐ Aktivieren
Require status checks	☐ Aktivieren
Require branch up to date	☐ Aktivieren
Require conversation resolution	☐ Aktivieren
Allow force pushes	☐ Deaktiviert
Allow deletions	☐ Deaktiviert

□□ Größeres Team oder Open-Source-Projekt

Regel	Empfehlung
Require pull request	☐ Aktivieren
Require approvals	☐ 2 Approvals
Dismiss stale approvals	☐ Aktivieren
Require review from Code Owners	☐ Aktivieren
Require status checks	☐ Aktivieren (mehrere Checks)
Require branch up to date	☐ Aktivieren
Require conversation resolution	☐ Aktivieren

Regel	Empfehlung
Require signed commits	□ Aktivieren
Require linear history	⚠ Team-Entscheidung
Do not allow bypassing	⚠ Bei hohen Compliance-Anforderungen

CODEOWNERS – Automatische Review-Zuweisung □□

Die `CODEOWNERS`-Datei ist ein mächtiges Feature, das perfekt mit Protected Branches zusammenarbeitet. Du definierst darin, wer für welche Teile des Codes verantwortlich ist. Diese Personen werden automatisch als Reviewer zu PRs hinzugefügt, die „ihre" Dateien betreffen.

So erstellst du eine CODEOWNERS-Datei

Die Datei muss an einem dieser Orte liegen:

- Repository-Root: `CODEOWNERS`
- `.github/CODEOWNERS` *(empfohlen)*
- `docs/CODEOWNERS`

Syntax und Beispiele

```
# Dies ist ein Kommentar

# Standard-Owner für alles, was nicht anders definiert ist
* @standard-reviewer

# Bestimmte Dateien/Ordner einem Team zuweisen
/src/api/           @backend-team
/src/frontend/      @frontend-team

# Bestimmte Dateitypen
*.js                @javascript-expert
*.css               @design-team
```

```
# Kritische Konfigurationsdateien
/config/            @team-lead @senior-developer
.github/workflows/  @devops-team

# Bestimmte Dateien mehreren Reviewern zuweisen (alle müssen reviewen)
/security/          @security-team @team-lead
```

CODEOWNERS in Kombination mit Branch Protection

Wenn du in den Branch Protection Rules **„Require review from Code Owners"** aktivierst, *müssen* die definierten Code Owners den PR genehmigen. Das stellt sicher, dass z.B.:

- Änderungen an der Datenbank-Schicht immer vom DBA geprüft werden
- Sicherheitskritischer Code vom Security-Team abgesegnet wird
- Frontend-Änderungen vom Frontend-Lead reviewed werden

Rulesets – Die moderne Alternative □□

GitHub hat 2023 **Rulesets** eingeführt, eine modernere und flexiblere Alternative zu den klassischen Branch Protection Rules. Rulesets bieten einige Vorteile:

- **Auf Organisations-Ebene definierbar** – Eine Regel für alle Repositories
- **Bessere Wildcards** – Komplexere Muster möglich
- **Tag Protection** – Nicht nur Branches, auch Tags schützen
- **Bypass-Listen** – Feingranulare Kontrolle, wer Regeln umgehen darf

Rulesets vs. klassische Branch Protection

Feature	Branch Protection	Rulesets
Repository-spezifisch	□	□
Organisations-weit	□	□

Feature	Branch Protection	Rulesets
Tag Protection	❌	✅
Bypass-Ausnahmen	Eingeschränkt	Flexibel
API-Steuerung	❌	✅

Empfehlung: Für einzelne Repositories sind die klassischen Branch Protection Rules einfacher. Für Organisationen mit vielen Repositories lohnt es sich, Rulesets zu evaluieren.

Visualisierung: Typischer PR-Workflow mit Branch Protection

```
flowchart TD
    A["Entwickler erstellt\nFeature-Branch"] --> B["Entwickler pusht\nCommits"]
    B --> C["Pull Request\nwird erstellt"]
    C --> D{"Status Checks\nerfolgreich?"}
    D -->|Nein| E["Entwickler fixt\nProbleme"]
    E --> B
    D -->|Ja| F{"Code Review\nbestanden?"}
    F -->|Nein| G["Entwickler arbeitet\nFeedback ein"]
    G --> B
    F -->|Ja| H{"Branch\naktuell?"}
    H -->|Nein| I["Branch mit main\naktualisieren"]
    I --> D
    H -->|Ja| J{"Alle Conversations\nresolved?"}
    J -->|Nein| K["Offene Diskussionen\nklaeren"]
    K --> J
    J -->|Ja| L["Merge in main\nmoeglich"]
    L --> M["Feature-Branch\nloeschen"]

    style A fill:#e1f5fe,color:#000000
    style L fill:#c8e6c9,color:#000000
    style M fill:#c8e6c9,color:#000000
    style E fill:#ffcdd2,color:#000000
    style G fill:#ffcdd2,color:#000000
```

Häufige Fragen und Probleme □□

„Ich bin Admin, kann aber nicht pushen – warum?"

Wenn **„Do not allow bypassing the above settings"** aktiviert ist, gelten die Regeln auch für dich. Du musst dann ebenfalls einen PR erstellen.

„Mein Status Check taucht nicht in der Auswahlliste auf"

Status Checks erscheinen erst, nachdem sie mindestens einmal gelaufen sind. Erstelle einen Test-PR oder pushe auf einen Testbranch, um den Workflow auszulösen.

„Wie kann ich in einem Notfall trotzdem direkt pushen?"

Wenn du Admin bist und **„Do not allow bypassing"** *nicht* aktiviert ist, kannst du die Regeln umgehen. Alternativ:

1. Die Regel temporär deaktivieren
2. Den Notfall-Push machen
3. Die Regel sofort wieder aktivieren
4. **Dokumentieren**, warum das nötig war

Besser: Auch Notfall-Hotfixes über PRs, aber mit minimaler Review-Zeit und einem speziellen `hotfix`-Label.

„Wie schütze ich mehrere Branches gleichzeitig?"

Nutze Wildcards im Branch-Pattern:

- `release/*` – Schützt alle Release-Branches

- `main` und `develop` – Erstelle zwei separate Regeln
- Mit Rulesets kannst du auch `main` und `develop` in einer Regel kombinieren

---

Checkliste für deine Branch Protection □□

Nutze diese Checkliste, um deine `main`-Branch-Protection einzurichten:

- ☐ Branch Protection Rule für `main` erstellt
- ☐ „Require pull request before merging" aktiviert
- ☐ Anzahl der erforderlichen Approvals festgelegt
- ☐ „Dismiss stale approvals" aktiviert
- ☐ Status Checks definiert (Tests, Linting)
- ☐ „Require branch to be up to date" aktiviert
- ☐ „Require conversation resolution" aktiviert
- ☐ Force Pushes deaktiviert (Standard)
- ☐ Branch-Deletion deaktiviert (Standard)
- ☐ CODEOWNERS-Datei erstellt (optional)
- ☐ Team über die neuen Regeln informiert

Fazit

Protected Branches sind **kein bürokratisches Hindernis**, sondern ein essentielles Werkzeug für professionelle Softwareentwicklung. Sie schützen nicht nur vor böswilligen Änderungen, sondern vor allem vor Versehen und menschlichen Fehlern. Die initiale Einrichtung dauert nur wenige Minuten, kann aber stundenlange Debugging-Sessions oder sogar Datenverlust verhindern.

Selbst als Solo-Entwickler empfehle ich dir, zumindest Force Pushes und Deletions für `main` zu verbieten. Sobald du im Team arbeitest, sollten Pull Requests mit mindestens einem Review zur Pflicht werden. Die Zeit, die du in Reviews „verlierst", gewinnst du mehrfach zurück durch früh entdeckte Bugs und bessere Code-Qualität.