Typische Git-Workflows im Überblick

Lernziele

Nach diesem Abschnitt kannst du:

- typische Git-Workflows voneinander unterscheiden,
- für Einzelarbeit, kleine Teams und Open-Source-Projekte einen passenden Ablauf wählen,
- lokale Arbeitsschritte von der Zusammenarbeit über ein Remote-Repository trennen,
- verstehen, warum Branches und Pull Requests zentrale Werkzeuge professioneller Teams sind,
- erkennen, dass ein Workflow eine *Teamvereinbarung* ist — nicht bloß eine Befehlsfolge.

Was bedeutet „Git-Workflow“?

Ein **Git-Workflow** beschreibt die vereinbarten Schritte, nach denen ein Team Änderungen entwickelt, prüft, integriert und veröffentlicht.

Git selbst schreibt keinen bestimmten Ablauf vor. Es stellt Werkzeuge bereit:

- Commits speichern nachvollziehbare Änderungen.
- Branches ermöglichen parallele Arbeit.
- Merges oder Rebases führen Entwicklungslinien zusammen.
- Remotes verbinden lokale Repositories mit gemeinsamen Servern wie GitHub.
- Pull Requests strukturieren Diskussionen und Reviews.

Ein Workflow beantwortet vor allem organisatorische Fragen:

- Auf welchem Branch beginnt neue Arbeit?
- Wie werden Änderungen benannt und in kleine Einheiten zerlegt?
- Wann dürfen Änderungen in den Hauptbranch?
- Wer prüft eine Änderung?
- Welche automatischen Tests müssen bestehen?

- Wie werden Releases und dringende Fehlerkorrekturen behandelt?

> Git ist das Werkzeug. Der Workflow ist die gemeinsame Arbeitsweise.

Die gemeinsame Grundlage fast aller Workflows

Unabhängig vom konkreten Modell durchläuft eine Änderung meist dieselben Stationen:

1. **Ausgangszustand aktualisieren**
   Die Entwicklerin oder der Entwickler holt aktuelle Änderungen aus dem gemeinsamen Repository.
2. **Isoliert entwickeln**
   Die Arbeit findet in einem passenden Branch statt — bei sehr kleinen Projekten manchmal direkt auf dem Hauptbranch.
3. **Kleine Commits erstellen**
   Jede logisch zusammenhängende Änderung wird als eigener Commit gespeichert.
4. **Änderung prüfen**
   Lokale Tests, statische Analyse, Linter und ein eigener Blick auf den Diff reduzieren Fehler frühzeitig.
5. **Änderung veröffentlichen**
   Der Branch wird zu GitHub oder einem anderen Remote gepusht.
6. **Integration vorbereiten**
   Ein Pull Request macht die Änderung sichtbar, diskutierbar und überprüfbar.
7. **Automatisch und menschlich prüfen**
   CI prüft beispielsweise Tests und Codequalität. Teammitglieder führen Reviews durch.
8. **In den Zielbranch integrieren**
   Nach erfolgreicher Prüfung wird die Änderung gemergt, gesquasht oder rebased.
9. **Branch aufräumen**
   Nicht mehr benötigte Feature-Branches werden gelöscht.

Dieser Ablauf lässt sich als allgemeines Muster lesen:

```
Hauptbranch aktualisieren
        ↓
Arbeitsbranch erstellen
        ↓
Änderung entwickeln
```

```
        ↓
Kleine Commits erstellen
        ↓
Tests und Selbstprüfung
        ↓
Branch veröffentlichen
        ↓
Pull Request und Review
        ↓
Integration in Hauptbranch
        ↓
Branch löschen
```

Der direkte Workflow für Einzelprojekte

Bei einem persönlichen Lern-, Demo- oder Kleinstprojekt arbeitet häufig nur eine Person am Repository. Dann kann die Entwicklung direkt auf dem Hauptbranch stattfinden, der heute meist `main` heißt.

Ein vereinfachter Ablauf:

```
Änderung vornehmen
        ↓
Änderung testen
        ↓
Commit erstellen
        ↓
Nach GitHub pushen
```

Beispiel:

```
git status
git add src/Calculator.php
git commit -m "Addiere Methode für Multiplikation"
git push
```

Vorteile

- Sehr wenig organisatorischer Aufwand.
- Schnell und leicht verständlich.
- Für Übungsprojekte und private Experimente gut geeignet.

Nachteile

- Es gibt keine isolierte Arbeitslinie für unfertige Änderungen.
- Fehler können direkt auf dem Hauptbranch landen.
- Reviews und automatisierte Qualitätskontrollen werden leicht übersprungen.
- Bei mehreren Mitwirkenden entstehen schnell Konflikte.

Für ein persönliches Repository ist direkteres Arbeiten oft angemessen. Sobald jedoch eine Änderung riskant, größer oder gemeinsam überprüft werden soll, lohnt sich ein eigener Branch.

Der zentrale Workflow

Der **zentrale Workflow** ähnelt der Arbeitsweise klassischer zentraler Versionskontrollsysteme: Alle Teammitglieder arbeiten direkt mit einem gemeinsamen Hauptbranch.

```
Entwicklerin A ── Commit und Push ──→ main im Remote-Repository
Entwickler B  ── Commit und Push ──→ main im Remote-Repository
Entwickler C  ── Commit und Push ──→ main im Remote-Repository
```

Obwohl Git ein verteiltes System ist, kann ein Team es auf diese zentralisierte Weise verwenden. Das Remote-Repository ist dann die maßgebliche gemeinsame Stelle.

Typischer Ablauf

1. Aktuellen Stand abrufen.
2. Direkt auf `main` entwickeln.
3. Änderungen committen.
4. Vor dem Push neue Änderungen anderer Personen integrieren.
5. Änderungen nach `main` pushen.

Stärken

- Einfaches Modell mit wenigen Branches.
- Für sehr kleine, eng abgestimmte Teams möglich.
- Wenig Prozessaufwand.

Risiken

- Unfertige oder fehlerhafte Änderungen können den Hauptbranch beeinträchtigen.
- Mehrere parallele Änderungen erzeugen häufiger Integrationskonflikte.
- Code Reviews vor der Integration sind nicht automatisch Teil des Prozesses.
- Ein kaputter Hauptbranch kann alle Teammitglieder blockieren.

Der zentrale Workflow ist für Lernzwecke nützlich, aber für die meisten professionellen Teams zu riskant. Moderne Plattformen wie GitHub fördern deshalb branchbasierte Zusammenarbeit mit Pull Requests.

Der Feature-Branch-Workflow

Beim **Feature-Branch-Workflow** erhält jede eigenständige Aufgabe einen eigenen Branch. Der Hauptbranch bleibt dabei möglichst stabil und jederzeit integrierbar.

Typische Branchnamen sind:

```
feature/add-invoice-export
fix/login-validation
docs/update-installation-guide
chore/update-dependencies
```

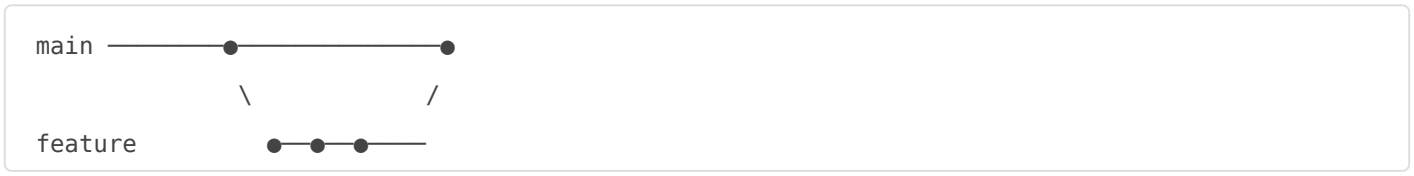
Ein Feature-Branch entsteht aus dem aktuellen Hauptbranch:

```
main
  └── feature/add-invoice-export
```

Während der Entwicklung enthält der Feature-Branch mehrere Commits:

```
main
  └── feature/add-invoice-export
        ├── Add export service
        ├── Add CSV formatter
        └── Add export tests
```

Nach Review und erfolgreichen Prüfungen wird die Arbeit zurück in `main` integriert:



Typischer Ablauf

1. `main` aktualisieren.
2. Einen Branch für eine klar abgegrenzte Aufgabe erstellen.
3. Die Änderung in kleinen, nachvollziehbaren Commits umsetzen.
4. Tests lokal ausführen.
5. Den Branch veröffentlichen.
6. Einen Pull Request nach `main` erstellen.
7. Review und CI abwarten.
8. Die Änderung integrieren.
9. Den Feature-Branch löschen.

Vorteile

- Unfertige Arbeit bleibt vom Hauptbranch getrennt.
- Mehrere Aufgaben können parallel entwickelt werden.
- Pull Requests schaffen einen klaren Ort für Diskussion und Review.
- Automatisierte Tests können vor dem Merge verpflichtend sein.
- Änderungen lassen sich gezielt zurückstellen oder verwerfen.

Wichtige Regel: Branches kurzlebig halten

Ein Feature-Branch sollte nicht wochenlang vom Hauptbranch getrennt bleiben. Je länger er existiert, desto größer wird das Risiko für:

- komplexe Merge-Konflikte,
- überholte Annahmen,
- schwer überprüfbare Pull Requests,
- doppelte oder widersprüchliche Arbeit.

Besser sind kleine, regelmäßig integrierte Änderungen. Ein Branch für „neue Rechnungsfunktion“ ist sinnvoller als ein monatelanger Branch für „Version 3 komplett neu schreiben“.

Pull-Request-Workflow

Ein **Pull Request** ist keine Git-Funktion im engeren Sinn, sondern eine Funktion von Plattformen wie GitHub. Er bittet darum, Änderungen aus einem Quellbranch in einen Zielbranch zu übernehmen.

Beispielsweise:

```
feature/add-invoice-export → main
```

Der Pull Request bündelt wichtige Informationen:

- die betroffenen Commits,
- den vollständigen Diff,
- die Beschreibung der Änderung,
- verknüpfte Issues,
- Kommentare und Review-Entscheidungen,
- Ergebnisse automatisierter Prüfungen.

Ein sinnvoller Pull Request beantwortet

- *Was* wurde verändert?
- *Warum* ist die Änderung nötig?
- *Wie* wurde sie getestet?
- Welche Risiken, Einschränkungen oder offenen Punkte gibt es?
- Welches Issue wird dadurch gelöst?

Eine kurze, gute Beschreibung könnte lauten:

> „Dieser Pull Request ergänzt einen CSV-Export für Rechnungen. Der Export berücksichtigt Rechnungsnummer, Datum, Kundschaft und Gesamtbetrag. PHPUnit-Tests decken leere Listen sowie mehrere Rechnungen ab.“

Pull Requests sind mehr als eine Freigabe

Ein professioneller Pull Request dient nicht nur dazu, Fehler zu finden. Er unterstützt auch:

- Wissenstransfer im Team,
- gemeinsame Architekturentscheidungen,
- Dokumentation technischer Gründe,

- Sicherheitsprüfungen,
- konsistente Codequalität.

Ein Review bedeutet dabei nicht: „Die Verantwortung liegt jetzt bei der prüfenden Person.“ Die Autorin oder der Autor bleibt für die Änderung verantwortlich.

GitHub Flow

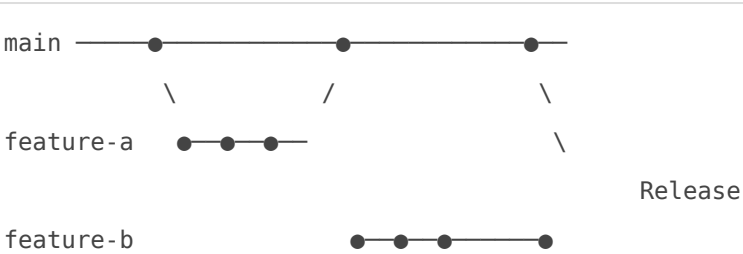
GitHub Flow ist ein schlanker, weit verbreiteter Workflow. Er basiert auf einem dauerhaft stabilen Hauptbranch und kurzen Arbeitsbranches.

Die Grundidee lautet:

> Alles auf `main` ist grundsätzlich bereit für eine Veröffentlichung.

Ablauf von GitHub Flow

1. Der Branch `main` enthält einen funktionierenden, freigegebenen Stand.
2. Für jede Aufgabe wird ein neuer Branch von `main` erstellt.
3. Die Änderung wird in kleinen Commits entwickelt.
4. Der Branch wird frühzeitig nach GitHub gepusht.
5. Ein Pull Request eröffnet Review und automatisierte Prüfungen.
6. Nach erfolgreicher Prüfung wird die Änderung nach `main` integriert.
7. Die Änderung wird bei Bedarf direkt aus `main` ausgeliefert.



Wann GitHub Flow gut passt

GitHub Flow eignet sich besonders für:

- Webanwendungen mit häufigen Deployments,
- Teams mit guter Testautomatisierung,

- SaaS-Produkte,
- Projekte mit kontinuierlicher Auslieferung,
- kleine bis mittlere Teams mit kurzen Änderungszyklen.

Voraussetzung: `main` muss geschützt sein

Damit GitHub Flow zuverlässig funktioniert, sollte der Hauptbranch nicht beliebig beschreibbar sein. Typische Schutzregeln sind:

- Pull Request vor dem Merge verpflichtend,
- mindestens eine Review-Freigabe,
- erfolgreiche CI-Prüfungen,
- keine direkten Pushes nach `main`,
- aktuelle Zielbranch-Basis vor dem Merge,
- optional eine Merge Queue bei hoher Teamaktivität.

Diese Regeln werden später als **Branch-Schutz** oder **Rulesets** genauer behandelt.

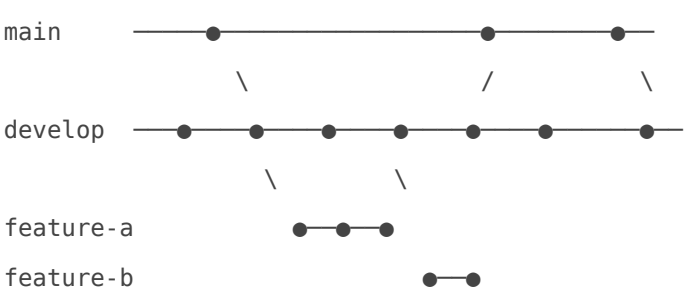
Git Flow

Git Flow verwendet mehr dauerhafte Branches und folgt stärker einem klassischen Release-Zyklus. Die zwei zentralen Branches sind üblicherweise:

- `main` für veröffentlichte, produktive Versionen,
- `develop` für die nächste geplante Entwicklungsversion.

Zusätzlich entstehen zeitlich begrenzte Branches:

- `feature/...` für neue Funktionen,
- `release/...` zur Vorbereitung einer Version,
- `hotfix/...` für dringende Fehlerkorrekturen in Produktion.



Typischer Ablauf

1. Neue Funktionen starten von `develop`.
2. Fertige Features werden nach `develop` integriert.
3. Für eine bevorstehende Veröffentlichung entsteht ein `release/...`-Branch.
4. Nach finaler Prüfung wird der Release-Branch nach `main` übernommen und getaggt.
5. Wichtige Produktionsfehler erhalten einen `hotfix/...`-Branch von `main`.
6. Ein Hotfix wird sowohl nach `main` als auch nach `develop` integriert.

Vorteile

- Klare Trennung zwischen produktivem Stand und laufender Entwicklung.
- Gut nachvollziehbare Release-Vorbereitung.
- Sinnvoll bei versionierter Software mit geplanten Auslieferungen.

Nachteile

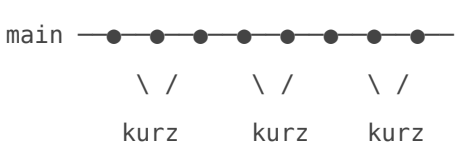
- Mehr Branches und mehr Merge-Vorgänge.
- Höherer organisatorischer Aufwand.
- Lang lebende Branches können Integrationen erschweren.
- Für kontinuierlich ausgelieferte Webanwendungen oft unnötig schwergewichtig.

Git Flow ist nicht „professioneller“ als GitHub Flow. Es passt lediglich zu anderen Rahmenbedingungen, etwa zu Produkten mit festen Release-Terminen, langen Testphasen oder unterstützten Wartungsversionen.

Trunk-Based Development

Bei **Trunk-Based Development** integriert das Team sehr häufig in einen gemeinsamen Hauptentwicklungszweig, den *Trunk*. In Git heißt dieser Branch oft `main`.

Der entscheidende Unterschied zum klassischen Feature-Branch-Workflow: Branches sind extrem kurzlebig oder werden nur lokal verwendet. Änderungen werden klein gehalten und häufig integriert.



Voraussetzungen

Trunk-Based Development funktioniert nur zuverlässig, wenn ein Team technische Disziplin mitbringt:

- umfassende automatisierte Tests,
- schnelle CI-Pipelines,
- kleine und häufige Commits,
- starke Code-Review-Kultur,
- Mechanismen wie Feature Flags für unvollständige Funktionen,
- konsequente Pflege eines funktionsfähigen Hauptbranches.

Geeignete Einsatzbereiche

- Teams mit kontinuierlicher Integration und Auslieferung,
- Produkte mit hoher Entwicklungsgeschwindigkeit,
- große Teams, die Integrationsstau vermeiden müssen,
- Systeme mit guter Testabdeckung.

Ohne zuverlässige Tests kann dieser Workflow riskant sein. Häufiges Integrieren ersetzt keine Qualitätsprüfung — es macht sie zwingender.

Forking-Workflow

Der **Forking-Workflow** ist besonders in Open-Source-Projekten üblich. Beitragende erhalten nicht direkt Schreibrechte auf das Original-Repository, sondern erstellen eine eigene Serverkopie, einen **Fork**.

```
Original-Repository
        ↓ Fork
Persönliches GitHub-Repository
        ↓ Clone
Lokales Repository
        ↓ Push
Persönlicher Fork
        ↓ Pull Request
Original-Repository
```

Dabei gibt es meist zwei Remotes:

- `origin`: der persönliche Fork,
- `upstream`: das ursprüngliche Projekt.

Typischer Ablauf

1. Ein Projekt auf GitHub forken.
2. Den Fork lokal klonen.
3. Das Original als `upstream` hinterlegen.
4. Einen Feature-Branch erstellen.
5. Änderungen entwickeln und zum eigenen Fork pushen.
6. Einen Pull Request vom Fork zum Original-Repository erstellen.
7. Den eigenen Fork regelmäßig mit `upstream` synchronisieren.

Vorteile

- Maintainer behalten die Kontrolle über Schreibrechte.
- Externe Beiträge sind ohne direkten Repository-Zugriff möglich.
- Änderungen bleiben zunächst im eigenen Bereich.
- Gut skalierbar für große Communities.

Nachteile

- Ein zusätzlicher Remote und Synchronisationsschritte erhöhen die Komplexität.
- Neue Mitwirkende müssen Fork, `origin` und `upstream` verstehen.
- Pull Requests können aus veralteten Forks entstehen, wenn nicht regelmäßig synchronisiert wird.

Release- und Hotfix-Workflows

Viele Teams benötigen neben normaler Feature-Entwicklung einen Ablauf für Veröffentlichungen und dringende Produktionsfehler.

Release-Branch

Ein **Release-Branch** friert den Funktionsumfang einer bevorstehenden Version ein. Während neue Features weiterentwickelt werden, konzentriert sich der Release-Branch auf:

- Tests,

- Fehlerkorrekturen,
- Dokumentation,
- Versionsnummern,
- Release Notes,
- Freigaben.

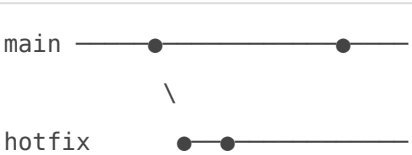
Beispiel:

```
main
  ├── feature/new-reporting
  └── release/2.4.0
```

Nach erfolgreicher Veröffentlichung wird der Stand typischerweise getaggt, etwa als `v2.4.0`.

Hotfix-Branch

Ein **Hotfix-Branch** korrigiert einen kritischen Fehler in einer bereits veröffentlichten Version. Er startet von dem Branch oder Tag, der den produktiven Stand repräsentiert.



Wichtig ist, dass ein Hotfix nicht nur in den produktiven Branch gelangt. Die Korrektur muss auch in die laufende Entwicklung übernommen werden. Sonst taucht derselbe Fehler in der nächsten Version erneut auf.

Welcher Workflow passt zu welchem Projekt?

Situation	Geeigneter Einstieg	Warum
Persönliches Lernprojekt	Direkter Workflow auf `main` oder kurze Feature-Branches	Minimaler Aufwand, schneller Lernfortschritt
Kleines Team mit wenigen Änderungen	Feature-Branch-Workflow mit Pull Requests	Gute Balance aus Sicherheit und Einfachheit
Webanwendung mit häufigen Deployments	GitHub Flow	Stabile Hauptlinie, kurze Branches, schnelle Integration

Situation	Geeigneter Einstieg	Warum
Versionierte Software mit festen Releases	Git Flow oder ein vereinfachter Release-Branch-Workflow	Geplante Release-Phasen und Hotfixes klar abbildbar
Team mit starker Testautomatisierung	Trunk-Based Development	Häufige Integration reduziert langfristige Abweichungen
Open-Source-Projekt	Forking-Workflow	Beiträge ohne direkte Schreibrechte ermöglichen
Großes Unternehmen mit Compliance-Anforderungen	Feature-Branches, Pull Requests, Schutzregeln und Release-Prozess	Nachvollziehbarkeit, Freigaben und kontrollierte Berechtigungen

Ein Team muss nicht ein Modell unverändert übernehmen. Häufig ist ein bewusst vereinfachter, dokumentierter Mischansatz sinnvoll.

Beispiel:

- Entwicklung nach GitHub Flow,
- Releases über Tags,
- Hotfixes über kurze `hotfix/...`-Branches,
- verpflichtende Tests und mindestens ein Review vor jedem Merge.

Merge, Squash und Rebase im Workflow

Wenn ein Pull Request fertig ist, muss entschieden werden, *wie* seine Änderungen in den Zielbranch gelangen. Die drei häufigsten Methoden sind:

Methode	Ergebnis in der Historie	Typischer Einsatz
Merge Commit	Verbindet zwei Entwicklungslinien mit einem Merge-Commit	Sichtbare Branch-Historie erwünscht
Squash Merge	Fasst alle Branch-Commits zu einem Commit zusammen	Kleine, saubere Hauptbranch-Historie
Rebase and Merge	Spielt Branch-Commits linear auf den Zielbranch	Lineare Historie, einzelne Commits sollen erhalten bleiben

Diese Entscheidung ist kein rein ästhetisches Detail. Sie beeinflusst:

- die Lesbarkeit der Historie,
- die Rückverfolgbarkeit einzelner Arbeitsschritte,
- die Größe eines späteren Reverts,

- den Umgang mit automatisierten Release Notes.

Für Einsteigerteams ist **Squash Merge** oft ein guter Standard: Der Feature-Branch darf mehrere Arbeitscommits enthalten, während `main` pro Pull Request einen klaren Commit erhält. Später werden Merge- und Rebase-Strategien ausführlich behandelt.

Ein praxistauglicher Standardworkflow für viele Teams

Für die meisten PHP-Projekte mit GitHub und PhpStorm ist dieser Ablauf ein solides Fundament:

1. Ein Issue beschreibt die Aufgabe.
2. Ein kurzer Feature- oder Fix-Branch entsteht von `main`.
3. Die Umsetzung erfolgt in kleinen, testbaren Commits.
4. PHPUnit, statische Analyse und Formatierungsprüfungen laufen lokal.
5. Der Branch wird zu GitHub gepusht.
6. Ein Pull Request verknüpft die Änderung mit dem Issue.
7. GitHub Actions führt die vorgesehenen Prüfungen aus.
8. Mindestens eine qualifizierte Person führt ein Review durch.
9. Nach erfolgreicher CI und Freigabe wird gemergt oder gesquasht.
10. Der Branch wird gelöscht.
11. `main` bleibt jederzeit in einem integrierbaren, idealerweise auslieferbaren Zustand.

Dieser Ablauf verbindet Geschwindigkeit mit Kontrolle, ohne unnötig kompliziert zu werden.

Grundsätze für jeden Workflow

Ein guter Workflow folgt einigen Regeln, unabhängig von Tool, Teamgröße oder Branch-Modell.

Änderungen klein halten

Kleine Änderungen sind leichter:

- zu verstehen,
- zu testen,
- zu prüfen,
- zu integrieren,

- zurückzunehmen.

Ein Pull Request mit einer klaren Aufgabe ist wertvoller als ein Sammelpaket aus Feature, Refactoring, Formatierung und Abhängigkeitsupdate.

Den Hauptbranch schützen

Direkte Pushes nach `main` sollten in Teamprojekten normalerweise vermieden werden. Pull Requests, Reviews und CI schaffen eine kontrollierte Integrationsschleuse.

Früh und regelmäßig integrieren

Lange getrennte Branches führen zu Konflikten und Überraschungen. Regelmäßige Synchronisierung mit dem Hauptbranch reduziert dieses Risiko.

Automatisierung als Sicherheitsnetz nutzen

Tests, Linter und statische Analyse ersetzen kein Review. Sie prüfen jedoch wiederholbare Regeln zuverlässig und entlasten Menschen von Routineaufgaben.

Konventionen dokumentieren

Ein Team sollte wichtige Entscheidungen schriftlich festhalten, zum Beispiel:

- Branch-Namensschema,
- Commit-Nachrichtenformat,
- erforderliche Prüfungen,
- Anzahl notwendiger Reviews,
- Merge-Strategie,
- Umgang mit Hotfixes,
- Regeln für Force Pushes.

Eine kurze `CONTRIBUTING.md` oder ein Abschnitt im Projekt-README verhindert viele Missverständnisse.

Häufige Fehlannahmen

„Ein Branch ist nur für große Features nötig.“

Nein. Gerade kleine, klar abgegrenzte Änderungen profitieren von einem Branch und einem kleinen Pull Request. Der Aufwand ist gering, der Nutzen für Review und Rückverfolgbarkeit hoch.

„Pull Requests sind nur Bürokratie.“

Ein schlecht gepflegter Pull Request kann bürokratisch wirken. Ein guter Pull Request ist dagegen ein Ort für Qualitätsprüfung, Kommunikation und Wissensweitergabe.

„Git Flow ist immer der professionelle Standard.“

Git Flow ist ein mögliches Modell, aber nicht universell passend. Für viele moderne Webprojekte ist GitHub Flow einfacher und effektiver.

„Der Hauptbranch darf nie kaputtgehen.“

Das ist ein wichtiges Ziel. Realistisch ist jedoch: Fehler können passieren. Deshalb braucht ein Team Tests, Reviews, Monitoring, schnelle Reverts und einen klaren Hotfix-Prozess.

„Viele lange Branches bedeuten gute Organisation.“

Oft ist das Gegenteil der Fall. Lang lebende Branches erhöhen die Distanz zum Hauptbranch und erschweren Integration. Gute Organisation bedeutet vor allem: kleine, nachvollziehbare und häufig integrierte Arbeit.

Merksätze

- **Ein Workflow ist eine Teamvereinbarung, keine Git-Pflicht.**
- **Branches isolieren Arbeit; Commits dokumentieren sie.**
- **Pull Requests machen Änderungen überprüfbar und diskutierbar.**
- **Kurze Branches und kleine Pull Requests reduzieren Integrationsrisiken.**
- **Der Hauptbranch sollte geschützt, geprüft und möglichst jederzeit verwendbar sein.**
- **Der beste Workflow ist der einfachste Ablauf, der die Anforderungen des Teams zuverlässig erfüllt.**

Im weiteren Kurs werden die Bausteine dieser Workflows — Branches, Merges, Rebases, Remotes, Pull Requests, Reviews, Releases und Automatisierung — Schritt für Schritt praktisch vertieft.

Revision #1
Created 2026-07-25 11:13:41 UTC by art10m
Updated 2026-07-25 11:13:41 UTC by art10m