Branches in Git – parallele Entwicklungslinien verstehen □□

Stell dir vor, du arbeitest an deinem Webprojekt und alles läuft stabil. Plötzlich hast du eine Idee für ein neues Feature – aber du bist dir nicht sicher, ob es funktioniert. Oder ein Kunde meldet einen dringenden Bug, während du gerade mitten in einer größeren Umbauaktion steckst. Genau für solche Situationen gibt es **Branches**.

Was ist ein Branch?

Ein **Branch** (englisch für „Zweig") ist im Grunde eine **eigenständige Entwicklungslinie** in deinem Git-Repository. Du kannst dir das wie parallele Universen deines Projekts vorstellen: In jedem Branch existiert eine eigene Version deines Codes, und Änderungen in einem Branch beeinflussen die anderen nicht – bis du sie bewusst zusammenführst.

Wenn du ein neues Repository erstellst, startest du automatisch auf einem Branch namens `main` (früher oft `master`). Dieser Branch enthält üblicherweise die **stabile, funktionierende Version** deines Projekts. Von hier aus kannst du beliebig viele weitere Branches abzweigen.

```
gitGraph
   commit id: "Projekt Start"
   commit id: "Erste Seiten"
   branch feature/kontaktformular
   checkout feature/kontaktformular
   commit id: "Formular HTML"
   commit id: "Validierung"
   checkout main
   commit id: "Bugfix Header"
   merge feature/kontaktformular id: "Merge Feature"
   commit id: "Weiterentwicklung"
```

Warum sind Branches so nützlich?

Branches lösen mehrere Probleme gleichzeitig, die ohne sie schnell zum Chaos führen würden:

- **Experimentieren ohne Risiko:** Du kannst neue Ideen ausprobieren, ohne deinen funktionierenden Code zu gefährden. Wenn das Experiment schiefgeht, löschst du einfach den Branch – dein `main`-Branch bleibt unberührt.
- **Paralleles Arbeiten:** Du kannst an mehreren Features oder Bugfixes gleichzeitig arbeiten, ohne dass sie sich gegenseitig in die Quere kommen. Jedes Feature lebt in seinem eigenen Branch.
- **Saubere Historie:** Statt einer endlosen Kette von Commits wie „Feature angefangen", „Feature weiter", „Bugfix zwischendurch", „Feature fertig" hast du klar getrennte Entwicklungsstränge, die du am Ende sauber zusammenführst.
- **Stabiler Hauptzweig:** Dein `main`-Branch bleibt immer in einem funktionierenden Zustand. Nur getestete, fertige Features werden dorthin überführt.

Wann sollte ich einen neuen Branch erstellen?

Als Faustregel gilt: **Erstelle einen neuen Branch, wenn du etwas Neues ausprobierst oder eine abgeschlossene Einheit entwickelst.** Hier sind typische Situationen:

1. **Neues Feature entwickeln**
   Du möchtest ein Kontaktformular, einen Login-Bereich oder eine neue Seite hinzufügen. Erstelle einen Branch wie `feature/kontaktformular` oder `feature/user-login`.
2. **Bugfix durchführen**
   Ein Fehler muss behoben werden, während du gerade an etwas anderem arbeitest. Erstelle einen Branch wie `bugfix/header-navigation` oder `fix/mobile-layout`.
3. **Experimentieren**
   Du willst ein neues CSS-Framework testen oder eine Funktion komplett umbauen, bist dir aber nicht sicher, ob es klappt. Ein Branch wie `experiment/tailwind-migration` gibt dir die Freiheit, ohne Konsequenzen zu testen.
4. **Größere Refactorings**
   Du willst deinen Code grundlegend umstrukturieren. Das dauert mehrere Commits und zwischendurch ist der Code möglicherweise kaputt. Ein eigener Branch wie `refactor/datenbank-struktur` hält deinen `main`-Branch sauber.

Praktisches Beispiel: Ein kleines Webprojekt

Nehmen wir an, du entwickelst eine einfache Portfolio-Website mit PHP. Dein `main`-Branch enthält bereits die Startseite und eine Über-mich-Seite, und alles funktioniert. Jetzt stehen drei Aufgaben an:

Ausgangssituation

```
main
 └── index.php (Startseite)
 └── about.php (Über mich)
 └── style.css (Styling)
```

Aufgabe 1: Kontaktformular entwickeln

Du erstellst einen neuen Branch:

```
git checkout -b feature/kontaktformular
```

Jetzt arbeitest du in diesem Branch an `contact.php`, fügst Formularvalidierung hinzu und passt das CSS an. Du machst mehrere Commits:

- „Kontaktformular HTML-Grundstruktur erstellt"
- „PHP-Validierung für Formularfelder hinzugefügt"
- „Erfolgsmeldung nach Absenden implementiert"

Währenddessen bleibt `main` unverändert und stabil.

Aufgabe 2: Dringender Bugfix

Ein Besucher meldet, dass die Navigation auf dem Handy nicht funktioniert. Du wechselst zurück zu `main` und erstellst von dort einen Bugfix-Branch:

```
git checkout main
git checkout -b bugfix/mobile-navigation
```

Du behebst den Fehler und committest:

- „Mobile Navigation: Hamburger-Menü funktioniert jetzt korrekt"

Dieser Fix ist dringend, also mergst du ihn direkt zurück in `main`:

```
git checkout main
git merge bugfix/mobile-navigation
```

Dein `main`-Branch hat jetzt den Bugfix, aber das Kontaktformular ist noch nicht drin – es wartet noch in seinem eigenen Branch.

Aufgabe 3: Feature fertigstellen und mergen

Du wechselst zurück zu deinem Feature-Branch und arbeitest weiter:

```
git checkout feature/kontaktformular
```

Sobald das Kontaktformular fertig und getestet ist, führst du es mit `main` zusammen:

```
git checkout main
git merge feature/kontaktformular
```

Endergebnis

Dein `main`-Branch enthält jetzt sowohl den Bugfix als auch das neue Feature – sauber getrennt entwickelt und zusammengeführt.

```
gitGraph
   commit id: "Startseite"
   commit id: "Über-mich-Seite"
   branch feature/kontaktformular
   checkout feature/kontaktformular
   commit id: "Formular HTML"
   checkout main
   branch bugfix/mobile-navigation
   checkout bugfix/mobile-navigation
   commit id: "Navigation gefixt"
```

```
    checkout main
    merge bugfix/mobile-navigation id: "Bugfix live"
    checkout feature/kontaktformular
    commit id: "Validierung"
    commit id: "Erfolgsmeldung"
    checkout main
    merge feature/kontaktformular id: "Feature live"
```

Branch-Namenskonventionen

Um Ordnung zu halten, haben sich bestimmte Namens-Präfixe etabliert:

Präfix	Verwendung	Beispiel
`feature/`	Neue Funktionen	`feature/newsletter-anmeldung`
`bugfix/` oder `fix/`	Fehlerbehebungen	`bugfix/login-fehler`
`hotfix/`	Dringende Fixes für Produktion	`hotfix/sicherheitsluecke`
`experiment/`	Experimente ohne Garantie	`experiment/neues-framework`
`refactor/`	Code-Umstrukturierungen	`refactor/datenbankzugriff`

Diese Präfixe sind keine Git-Vorgabe, sondern eine bewährte Konvention. Sie helfen dir (und anderen), auf einen Blick zu erkennen, worum es in einem Branch geht.

Zusammenfassung

Branches sind eines der mächtigsten Konzepte in Git. Sie erlauben dir, **gefahrlos zu experimentieren**, **parallel an mehreren Dingen zu arbeiten** und deinen **Hauptzweig stabil zu halten**. Für ein kleines Webprojekt reicht oft ein einfaches Muster: Arbeite auf `main`, aber erstelle für jedes neue Feature oder jeden Bugfix einen eigenen Branch – und merge ihn erst, wenn er fertig ist. □□

Revision #1
Created 2026-06-17 01:12:00 UTC by art10m
Updated 2026-06-17 01:13:24 UTC by art10m