Kapitel 5: Mit Branches arbeiten

Einleitung: Branches (Verzweigungen) sind eines der mächtigsten Features von Git. Stell dir deinen Code wie einen Baum vor: Der Hauptstamm ist dein funktionierender Code, und jeder Branch ist ein Ast, auf dem du experimentieren kannst, ohne den Stamm zu gefährden. Wenn du ein neues Feature entwickelst oder einen Bug behebst, erstellst du einen separaten Branch, arbeitest dort in Ruhe, und führst die Änderungen erst zusammen (Merge), wenn alles funktioniert. So bleibt dein Hauptcode immer stabil. Allerdings kann es beim Zusammenführen zu Konflikten kommen, wenn dieselbe Stelle unterschiedlich bearbeitet wurde. Dieses Kapitel zeigt dir, wie du Branches sicher handhabst.

- Branches in Git – parallele Entwicklungslinien verstehen □□
- Branches in PhpStorm erstellen und wechseln □□
- Feature-Branch in den Main-Branch mergen – Schritt für Schritt in PhpStorm □□
- Merge-Konflikte verstehen und in PhpStorm lösen □□

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. □□

Branches in PhpStorm erstellen und wechseln □□

Das Arbeiten mit Branches gehört zu den wichtigsten Git-Funktionen, und PhpStorm macht es dir besonders leicht, neue Branches zu erstellen und zwischen ihnen zu wechseln – ganz ohne Terminal.

Einen neuen Branch erstellen

Es gibt mehrere Wege, um in PhpStorm einen neuen Branch anzulegen:

Weg 1: Über die Branch-Anzeige in der Statusleiste

1. Schau in die **rechte untere Ecke** von PhpStorm – dort siehst du den Namen deines aktuellen Branches (z. B. „main" oder „master").
2. **Klicke auf den Branch-Namen** – es öffnet sich ein Popup-Menü mit allen verfügbaren Branches.
3. Wähle **New Branch** (oder „Neuer Branch" in der deutschen Version).
4. Gib einen **aussagekräftigen Namen** für deinen Branch ein, z. B.:
   - `feature/kontaktformular`
   - `bugfix/login-fehler`
   - `experiment/neues-design`
5. Aktiviere die Option **Checkout branch** (standardmäßig aktiv), damit du direkt in den neuen Branch wechselst.
6. Bestätige mit **Create**.

Weg 2: Über das Git-Menü

1. Gehe zu **Git → New Branch…** in der Menüleiste.
2. Der weitere Ablauf ist identisch: Name eingeben, Checkout-Option wählen, bestätigen.

Weg 3: Über das Git-Tool-Fenster

1. Öffne das **Git-Fenster** am unteren Rand von PhpStorm (Reiter „Git").
2. Wechsle zum Tab **Log** und klicke mit der rechten Maustaste auf einen beliebigen Commit.
3. Wähle **New Branch…** – der neue Branch startet dann an genau diesem Commit.

> **Tipp:** Die dritte Variante ist besonders nützlich, wenn du einen Branch nicht vom aktuellen Stand, sondern von einem älteren Commit aus erstellen möchtest.

Zwischen Branches wechseln

Das Wechseln zwischen Branches nennt man in Git **Checkout**. In PhpStorm geht das schnell:

Über die Statusleiste (empfohlen)

1. Klicke wieder auf den **Branch-Namen** in der rechten unteren Ecke.
2. Du siehst eine Liste aller lokalen Branches (unter „Local") und – falls vorhanden – auch der Remote-Branches (unter „Remote").
3. **Klicke auf den gewünschten Branch** und wähle **Checkout**.
4. PhpStorm wechselt sofort in den ausgewählten Branch, und dein Arbeitsverzeichnis zeigt nun den Stand dieses Branches.

Über das Git-Menü

Alternativ kannst du **Git → Branches…** wählen – das öffnet dasselbe Popup wie der Klick auf die Statusleiste.

Was passiert beim Branch-Wechsel im Hintergrund?

Wenn du den Branch wechselst, passieren mehrere Dinge:

```
flowchart TD
    A["Du waehlst Branch 'feature/login'"] --> B["Git prueft auf ungespeicherte Aenderungen"]
    B -->|Keine Konflikte| C["HEAD zeigt jetzt auf 'feature/login'"]
    C --> D["Dateien im Arbeitsverzeichnis\nwerden angepasst"]
    D --> E["PhpStorm aktualisiert\nalle offenen Editoren"]
    B -->|Konflikte moeglich| F["PhpStorm fragt:\nStashen, Committen oder Abbrechen?"]
```

Wichtig zu verstehen:

- Der **HEAD-Zeiger** (das ist Git's Markierung für „wo bin ich gerade?") wird auf den neuen Branch gesetzt.
- Deine **Dateien im Arbeitsverzeichnis** werden automatisch auf den Stand des Ziel-Branches gebracht.
- Alle **offenen Dateien in PhpStorm** aktualisieren sich entsprechend – du siehst sofort den Code des neuen Branches.

Umgang mit ungespeicherten Änderungen

Was passiert, wenn du Änderungen hast, die noch nicht committet sind, und trotzdem den Branch wechseln willst?

PhpStorm zeigt dir in diesem Fall einen Dialog mit mehreren Optionen:

Option	Bedeutung
Smart Checkout	PhpStorm versucht, deine Änderungen in den neuen Branch „mitzunehmen". Funktioniert oft, aber nicht immer konfliktfrei.
Force Checkout	Deine lokalen Änderungen werden verworfen – **Vorsicht, Datenverlust möglich!**
Don't Checkout	Der Wechsel wird abgebrochen, du bleibst im aktuellen Branch.
Stash & Checkout	Deine Änderungen werden temporär „versteckt" (gestasht), der Branch wird gewechselt, und du kannst den Stash später wieder anwenden.

> ◫◫ **Empfehlung für Anfänger:** Am sichersten ist es, vor einem Branch-Wechsel entweder zu **committen** oder die Änderungen mit **Stash** zwischenzuspeichern. So gehst du kein Risiko ein.

Praktisches Beispiel: Feature-Entwicklung

Angenommen, du arbeitest an einem Blog-Projekt und möchtest ein Kontaktformular hinzufügen, ohne deinen stabilen Code zu gefährden:

1. **Branch erstellen:**
   Klicke auf „main" → **New Branch** → Name: `feature/kontaktformular` → **Create**
2. **Entwickeln:**
   Du bist jetzt im neuen Branch und kannst arbeiten. Alle Commits landen nur in diesem Branch.
3. **Zwischendurch zum Hauptbranch wechseln:**
   Ein dringender Bug wird gemeldet. Klicke auf „feature/kontaktformular" → wähle „main" → **Checkout**. Du bist zurück im stabilen Code.
4. **Bug fixen, committen, zurückwechseln:**
   Nach dem Fix wechselst du wieder zu „feature/kontaktformular" und arbeitest weiter am Formular.

Branches löschen

Wenn ein Branch nicht mehr gebraucht wird (z. B. nach einem Merge), kannst du ihn löschen:

1. Klicke auf den Branch-Namen in der Statusleiste.
2. Finde den zu löschenden Branch in der Liste.
3. Klicke mit der **rechten Maustaste** darauf (oder hover und klicke auf das Drei-Punkte-Menü).
4. Wähle **Delete**.

> ⚠ **Hinweis:** Den Branch, in dem du dich gerade befindest, kannst du nicht löschen. Wechsle vorher in einen anderen Branch.

Zusammenfassung der wichtigsten Aktionen

Aktion	Weg in PhpStorm
Neuen Branch erstellen	Statusleiste → Branch-Name → *New Branch*
Branch wechseln	Statusleiste → Branch-Name → gewünschten Branch → *Checkout*
Aktuellen Branch sehen	Rechte untere Ecke der Statusleiste
Branch löschen	Statusleiste → Rechtsklick auf Branch → *Delete*
Alle Branches anzeigen	Git-Fenster → Tab *Log* → linke Spalte „Branches"

Mit diesen Grundlagen kannst du effektiv mit Branches arbeiten und verschiedene Features oder Experimente sauber voneinander trennen – ohne Angst haben zu müssen, deinen funktionierenden Code zu zerstören. □□

Feature-Branch in den Main-Branch mergen – Schritt für Schritt in PhpStorm □□

Du hast in deinem Feature-Branch (z. B. `login-formular`) alle Änderungen abgeschlossen und getestet. Jetzt ist es Zeit, diese Arbeit in deinen Hauptbranch (meist `main` oder `master`) zu integrieren. Diesen Vorgang nennt man **Merge** – und PhpStorm macht ihn dir besonders komfortabel.

Das Grundprinzip verstehen

Beim Mergen passiert Folgendes: Git nimmt alle Änderungen aus deinem Feature-Branch und **fügt sie in einen anderen Branch ein** (den Zielbranch). Wichtig dabei ist die Reihenfolge:

```
flowchart LR
    A["1. Wechsle zum Zielbranch\n(z.B. main)"] --> B["2. Merge den Feature-Branch\nin den aktuellen Branch"]
    B --> C["3. Aenderungen sind\njetzt in main"]
```

> □□ **Merke:** Du wechselst immer **zuerst** in den Branch, der die Änderungen **empfangen** soll, und holst dann den anderen Branch hinein.

Schritt-für-Schritt-Anleitung

1. Offene Änderungen committen

Bevor du mergst, stelle sicher, dass in **beiden Branches** keine uncommitteten Änderungen herumliegen. Öffne das Commit-Fenster (`Ctrl + K` / `Cmd + K`) und prüfe, ob alles sauber committet ist. Falls du noch Änderungen im Feature-Branch hast, committe diese zuerst.

2. Zum Main-Branch wechseln

Da du die Änderungen **in den Main-Branch** integrieren willst, musst du dorthin wechseln:

1. Schau in die **rechte untere Ecke** von PhpStorm – dort siehst du den aktuellen Branch-Namen.
2. **Klicke darauf** – es öffnet sich das Branch-Popup.
3. Suche deinen Main-Branch (z. B. `main` oder `master`) unter **Local Branches**.
4. Klicke auf den Branch-Namen und wähle **Checkout**.

PhpStorm wechselt nun zum Main-Branch. Du siehst, dass sich der Name in der Statusleiste ändert, und dein Working Directory zeigt jetzt den Stand des Main-Branches.

3. Den Feature-Branch mergen

Jetzt kommt der eigentliche Merge:

1. Klicke erneut auf den **Branch-Namen** in der Statusleiste (jetzt steht dort „main").
2. Im Popup siehst du unter **Local Branches** deinen Feature-Branch (z. B. `login-formular`).
3. **Klicke auf den Feature-Branch** und wähle im Untermenü **Merge into Current**.

PhpStorm führt nun den Merge durch. Wenn alles glatt läuft (keine Konflikte), siehst du eine kurze Erfolgsmeldung, und die Änderungen aus deinem Feature-Branch sind jetzt Teil des Main-Branches.

4. Das Ergebnis überprüfen

Nach dem Merge lohnt sich ein kurzer Blick ins Git-Log (`Alt + 9` oder **Git → Show Git Log**):

- Du siehst einen neuen **Merge-Commit**, der die beiden Entwicklungslinien zusammenführt.
- Die Commit-Historie zeigt, dass die Commits aus dem Feature-Branch nun auch im Main-Branch enthalten sind.

Was passiert im Hintergrund?

Wenn du in PhpStorm „Merge into Current" auswählst, führt Git im Hintergrund folgenden Befehl aus:

```
git merge login-formular
```

Dabei erstellt Git (je nach Situation) entweder:

- Einen **Merge-Commit**, der beide Branches zusammenführt und als „Knotenpunkt" in der Historie sichtbar ist.
- Einen **Fast-Forward-Merge**, wenn der Main-Branch seit der Erstellung des Feature-Branches keine eigenen Änderungen hatte. In diesem Fall wird kein extra Merge-Commit erstellt – die Commits werden einfach „vorgeschoben".

Was tun bei Merge-Konflikten? ⚠

Manchmal hat Git ein Problem: Wenn **dieselbe Stelle** in einer Datei in beiden Branches unterschiedlich geändert wurde, kann Git nicht automatisch entscheiden, welche Version gelten soll. Das nennt man einen **Merge-Konflikt**.

In diesem Fall öffnet PhpStorm automatisch ein **Konfliktlösungs-Fenster**:

1. Du siehst **drei Spalten**: Links dein aktueller Branch (Main), rechts der Feature-Branch, in der Mitte das Ergebnis.
2. Für jede Konfliktstelle kannst du entscheiden:
   - **Accept Left** – die Version aus Main übernehmen
   - **Accept Right** – die Version aus dem Feature-Branch übernehmen
   - **Manuell bearbeiten** – in der Mitte selbst eine Kombination schreiben
3. Wenn alle Konflikte gelöst sind, klicke auf **Apply**.
4. PhpStorm erstellt dann automatisch den Merge-Commit.

> □□ **Tipp:** Konflikte klingen schlimmer als sie sind. In den meisten Fällen ist schnell klar, welche Version die richtige ist – und das visuelle Tool von PhpStorm macht die Lösung sehr intuitiv.

Nach dem Merge: Feature-Branch löschen?

Wenn der Merge erfolgreich war und du den Feature-Branch nicht mehr brauchst, kannst du ihn aufräumen:

1. Klicke wieder auf den **Branch-Namen** in der Statusleiste.
2. Finde deinen Feature-Branch unter **Local Branches**.
3. Klicke darauf und wähle **Delete**.

Das hält deine Branch-Liste übersichtlich. Die Commits gehen dabei **nicht verloren** – sie sind jetzt Teil des Main-Branches und bleiben in der Historie erhalten.

Zusammenfassung

Schritt	Aktion in PhpStorm
1. Alles committen	`Ctrl/Cmd + K` → Commit
2. Zum Zielbranch wechseln	Branch-Popup → Main → **Checkout**
3. Feature-Branch mergen	Branch-Popup → Feature-Branch → **Merge into Current**
4. Bei Konflikten	Konflikt-Dialog nutzen → **Apply**
5. Optional aufräumen	Branch-Popup → Feature-Branch → **Delete**

Mit dieser Routine kannst du sicher neue Features entwickeln, testen und dann sauber in deinen Hauptbranch integrieren – genau so, wie es in professionellen Projekten üblich ist. □□

Merge-Konflikte verstehen und in PhpStorm lösen □□

Ein **Merge-Konflikt** ist einer der Momente, vor denen Git-Anfänger oft Respekt haben – aber wenn du einmal verstanden hast, was passiert und wie du damit umgehst, verliert er schnell seinen Schrecken. PhpStorm bietet dir dafür ein hervorragendes visuelles Werkzeug, das die Auflösung deutlich erleichtert.

Was ist ein Merge-Konflikt?

Ein Merge-Konflikt entsteht, wenn Git **nicht automatisch entscheiden kann**, wie zwei unterschiedliche Änderungen zusammengeführt werden sollen. Das passiert, wenn **dieselbe Stelle in einer Datei** in beiden Branches verändert wurde – Git weiß dann nicht, welche Version die „richtige" ist und bittet dich um Hilfe.

Wann tritt ein Konflikt auf?

Konflikte entstehen typischerweise in diesen Situationen:

- **Zwei Branches bearbeiten dieselbe Codezeile:** Du änderst in deinem Feature-Branch eine Funktion, während jemand (oder du selbst) im Main-Branch dieselbe Zeile ebenfalls angepasst hat.
- **Eine Datei wird in einem Branch gelöscht, im anderen bearbeitet:** Git weiß nicht, ob die Datei weg soll oder die Bearbeitung wichtig ist.
- **Parallele Arbeit an derselben Datei:** Selbst wenn du alleine arbeitest, kann das passieren, wenn du z. B. einen Hotfix direkt im Main-Branch machst, während du parallel an einem Feature arbeitest.

Wann gibt es *keinen* Konflikt?

Wenn die Änderungen in **unterschiedlichen Dateien** oder in **unterschiedlichen Bereichen derselben Datei** stattfinden, kann Git diese problemlos automatisch zusammenführen – kein Konflikt.

Ein konkretes Beispiel

Stell dir vor, du hast eine Datei `config.php` mit folgender Zeile:

```
$siteTitle = "Meine Website";
```

Jetzt passiert Folgendes:

1. Im **Main-Branch** änderst du die Zeile zu:
   ```
   $siteTitle = "Meine tolle Website";
   ```
2. Im **Feature-Branch** (den du vorher vom Main-Branch abgezweigt hast) änderst du dieselbe Zeile zu:
   ```
   $siteTitle = "Mein Webprojekt";
   ```

Wenn du jetzt versuchst, den Feature-Branch in den Main-Branch zu mergen, hat Git ein Problem: Beide Branches haben **dieselbe Zeile unterschiedlich verändert**. Git kann nicht wissen, ob du „Meine tolle Website" oder „Mein Webprojekt" haben möchtest – oder vielleicht sogar etwas ganz anderes.

```
flowchart TD
    A["Ursprung<br>$siteTitle = Meine Website"] --> B["Main-Branch<br>$siteTitle = Meine tolle Website"]
    A --> C["Feature-Branch<br>$siteTitle = Mein Webprojekt"]
    B --> D["Merge-Versuch"]
    C --> D
    D --> E["Konflikt!<br>Git weiss nicht,<br>welche Version gilt"]
```

Merge-Konflikt in PhpStorm lösen – Schritt für Schritt

Angenommen, du bist im **Main-Branch** und willst deinen Feature-Branch mergen:

1. **Merge starten**
   Klicke unten rechts auf den Branch-Namen (z. B. „main"), wähle deinen Feature-Branch aus der Liste und klicke auf **Merge into Current**. Alternativ: **Git → Merge…** im Menü.

2. **Konflikt-Meldung erscheint**
   Wenn ein Konflikt existiert, zeigt PhpStorm dir ein Dialogfenster mit dem Titel **Merge Conflicts**. Hier siehst du eine Liste aller Dateien, die Konflikte haben – in unserem Beispiel `config.php`.
3. **Konflikt-Editor öffnen**
   Wähle die konfliktbehaftete Datei aus und klicke auf **Merge…** (oder doppelklicke auf die Datei). Es öffnet sich der **3-Wege-Merge-Editor** – das Herzstück der Konfliktlösung in PhpStorm.
4. **Den 3-Wege-Editor verstehen**
   Du siehst drei Spalten:

Linke Spalte	Mittlere Spalte	Rechte Spalte
Dein aktueller Branch (Main)	**Das Ergebnis** (was am Ende gespeichert wird)	**Der eingehende Branch** (Feature)

   Die konfliktbehafteten Stellen sind farblich hervorgehoben. In unserem Beispiel siehst du links „Meine tolle Website" und rechts „Mein Webprojekt".
5. **Änderungen übernehmen oder kombinieren**
   Für jede konfliktbehaftete Stelle hast du mehrere Möglichkeiten:
   - **Accept Left** (Pfeil-Button neben der linken Version): Übernimmt die Version aus dem Main-Branch.
   - **Accept Right** (Pfeil-Button neben der rechten Version): Übernimmt die Version aus dem Feature-Branch.
   - **Accept Both**: Übernimmt beide Versionen nacheinander (selten sinnvoll bei derselben Zeile, aber nützlich bei Codeblöcken).
   - **Manuell bearbeiten**: Du kannst in der mittleren Spalte direkt den Text anpassen und etwas völlig Neues schreiben, z. B.:

     ```
     $siteTitle = "Mein tolles Webprojekt";
     ```

6. **Alle Konflikte lösen**
   Gehe jeden markierten Konflikt durch und triff eine Entscheidung. Die mittlere Spalte zeigt dir immer das aktuelle Ergebnis. Sobald alle Konflikte gelöst sind (keine roten Markierungen mehr), wird der **Apply**-Button aktiv.
7. **Änderungen übernehmen**
   Klicke auf **Apply**. Der Konflikt für diese Datei ist gelöst. Wenn es mehrere Dateien mit Konflikten gibt, wiederhole den Prozess für jede.
8. **Merge abschließen**
   Nachdem alle Konflikte gelöst sind, erstellt PhpStorm automatisch einen **Merge-Commit**. Du kannst die vorgeschlagene Commit-Nachricht anpassen (z. B. „Merge branch ‚feature-xyz' into main") und den Commit bestätigen.

Tipps für den Umgang mit Konflikten

- **Keine Panik!** Ein Konflikt bedeutet nicht, dass etwas kaputt ist – Git bittet dich nur um eine Entscheidung, die es selbst nicht treffen kann.
- **Verstehe beide Versionen:** Bevor du eine Version wählst, schau dir an, *warum* die Änderung gemacht wurde. Manchmal ist die beste Lösung eine Kombination aus beiden.
- **Teste nach dem Merge:** Nachdem du Konflikte gelöst hast, teste dein Projekt gründlich. Syntaktisch kann alles korrekt aussehen, aber logisch kann etwas nicht zusammenpassen.
- **Konflikte vermeiden:** Je öfter du deine Branches synchronisierst (regelmäßig `main` in deinen Feature-Branch mergen), desto kleiner und seltener werden Konflikte.
- **Im Zweifel: Abbrechen:** Wenn du dir unsicher bist, kannst du den Merge jederzeit mit **Git → Abort Merge** abbrechen. Alles bleibt beim Alten, und du kannst dich in Ruhe vorbereiten.

Zusammenfassung

Ein Merge-Konflikt entsteht, wenn Git zwei unterschiedliche Änderungen an derselben Stelle nicht automatisch zusammenführen kann. Das ist kein Fehler, sondern eine **Aufforderung zur Entscheidung**. PhpStorm bietet dir mit dem 3-Wege-Merge-Editor ein mächtiges Werkzeug, um diese Konflikte visuell und komfortabel zu lösen – du siehst beide Versionen nebeneinander, kannst gezielt auswählen oder kombinieren und behältst immer die volle Kontrolle über das Ergebnis.