Versionskontrolle und Git verstehen

Dieses Kapitel erklärt, warum Versionskontrolle unverzichtbar ist und welche Probleme Git löst. Du lernst Snapshots, Repositories, Commits, Arbeitsstände und verteilte Systeme kennen und grenzt Git von GitHub sowie von zentralen Werkzeugen wie Subversion verständlich ab.

Warum Software versioniert wird

Lernziele

Nach dieser Lektion kannst du erklären,

- warum Softwareprojekte eine nachvollziehbare Historie benötigen,
- welche Risiken ohne Versionskontrolle entstehen,
- warum Kopien wie `projekt_final_neu_wirklichfinal` keine verlässliche Lösung sind,
- wie Versionierung Zusammenarbeit, Qualität und Sicherheit unterstützt,
- weshalb Git nicht nur für große Teams, sondern auch für Einzelentwickler wertvoll ist.

Software verändert sich ständig

Software ist kein statisches Dokument. Schon ein kleines Projekt entwickelt sich fortlaufend weiter:

- Funktionen kommen hinzu.
- Fehler werden korrigiert.
- Abhängigkeiten werden aktualisiert.
- Konfigurationen ändern sich.
- Dateien werden umbenannt, verschoben oder gelöscht.
- Anforderungen ändern sich nach Gesprächen mit Kunden, dem Team oder dem Fachbereich.

Eine einzelne Datei kann dabei dutzende oder hunderte Änderungen durchlaufen. Bei einem PHP-Projekt betrifft das häufig nicht nur den Anwendungscode, sondern auch Tests, Composer-Konfiguration, Datenbankmigrationen, Dokumentation und CI-Konfiguration.

Die entscheidende Frage lautet daher nicht, *ob* sich ein Projekt verändert, sondern:

> **Wie lässt sich nachvollziehen, was sich wann, warum und durch wen verändert hat?**

Genau dafür wird Software versioniert.

Das Problem ohne Versionskontrolle

Stell dir vor, du entwickelst eine kleine PHP-Anwendung für eine Terminverwaltung. Zu Beginn existiert nur eine Datei:

```
terminverwaltung/
└── index.php
```

Im Laufe der Zeit ergänzt du Validierungen, einen Datenbankzugriff und ein Login. Vor einer riskanten Änderung erstellst du vorsichtshalber eine Kopie des gesamten Ordners:

```
terminverwaltung/
terminverwaltung_backup/
terminverwaltung_backup_neu/
terminverwaltung_final/
terminverwaltung_final2/
terminverwaltung_final_wirklich/
```

Dieses Vorgehen wirkt zunächst sicher, führt aber schnell zu Problemen.

Unklare Bedeutung von Kopien

An den Ordnernamen ist kaum zu erkennen:

- Welche Version läuft produktiv?
- Welche Kopie enthält den letzten funktionierenden Stand?
- Welche Änderung wurde zwischen `final` und `final2` vorgenommen?
- Welche Version enthält den Fehler?
- Darf ein alter Ordner gelöscht werden?

Der Dateiname oder Ordnername wird zur improvisierten Versionshistorie. Diese Historie ist jedoch unvollständig, fehleranfällig und schwer durchsuchbar.

Änderungen gehen verloren

Wenn du eine Funktion umbaust und später feststellst, dass die frühere Variante besser war, brauchst du den alten Zustand. Ohne Versionskontrolle musst du hoffen, dass eine passende Sicherheitskopie existiert.

Fehlt sie, bleibt nur:

- Änderungen mühsam zurückbauen,
- Code aus Erinnerungen rekonstruieren,
- Dateien vergleichen,
- oder Arbeit erneut erledigen.

Zusammenarbeit überschreibt Arbeit

Arbeiten zwei Personen gleichzeitig an derselben Datei, entsteht ein besonders gefährliches Szenario:

1. Anna bearbeitet `UserService.php`.
2. Ben bearbeitet ebenfalls `UserService.php`.
3. Anna speichert ihre Datei.
4. Ben speichert später seine Datei über Annas Version.

Je nach Arbeitsweise kann Annas Änderung vollständig verschwinden. Selbst wenn beide vorher Kopien angelegt haben, ist später oft unklar, welche Teile aus welcher Datei übernommen werden müssen.

Fehler sind schwer einzugrenzen

Ein Fehler wird am Montag entdeckt. Am Freitag funktionierte noch alles. Inzwischen gab es viele Änderungen.

Ohne Historie ist die Fehlersuche mühsam:

- Welche Dateien wurden seit Freitag verändert?
- Welche konkrete Zeile war die Ursache?
- Wer kennt die fachliche Entscheidung hinter der Änderung?
- Lässt sich der funktionierende Zustand kurzfristig wiederherstellen?

Ohne belastbare Antworten steigt der Zeitaufwand – und das Risiko, beim Reparieren neue Fehler einzubauen.

Versionierung schafft eine überprüfbare Projektgeschichte

Versionskontrolle speichert nicht einfach nur Dateikopien. Sie hält eine **Geschichte bewusst festgehaltener Projektzustände** fest.

Ein solcher gespeicherter Zustand wird in Git später *Commit* genannt. Ein Commit dokumentiert typischerweise:

- welche Dateien geändert wurden,
- welche Inhalte sich geändert haben,
- wann die Änderung gespeichert wurde,
- wer sie erstellt hat,
- und warum sie vorgenommen wurde.

Statt vieler unklarer Projektordner entsteht eine nachvollziehbare Abfolge:

```
Projektbeginn
    ↓
Grundstruktur angelegt
    ↓
Benutzerregistrierung ergänzt
    ↓
Passwortvalidierung korrigiert
    ↓
Tests für Login hinzugefügt
    ↓
Sicherheitslücke geschlossen
```

Die Historie wird damit zu einem technischen Gedächtnis des Projekts.

Die zentralen Gründe für Versionskontrolle

Änderungen nachvollziehen

Eine gute Projektgeschichte beantwortet wichtige Fragen schnell:

- Was wurde verändert?
- Wann wurde es verändert?
- Wer hat es verändert?
- Warum wurde es verändert?
- Zu welchem Ticket, Issue oder Fehlerbericht gehört die Änderung?

Ein Commit kann beispielsweise die Nachricht tragen:

```
Fix: Passwort-Hash vor dem Speichern erzeugen
```

Diese Nachricht ist wesentlich hilfreicher als eine Ordnerkopie namens `backup_17`.

> **Grundsatz:** Eine Änderung ohne nachvollziehbaren Zweck wird später teuer – spätestens bei Wartung, Fehlersuche oder Review.

Frühere Zustände wiederherstellen

Nicht jede Änderung ist erfolgreich. Ein neues Feature kann Fehler verursachen, eine Konfigurationsänderung kann den Build beschädigen oder ein Refactoring kann unerwartete Seiteneffekte haben.

Versionskontrolle ermöglicht es, gezielt zu einem früheren Zustand zurückzukehren oder einzelne ältere Dateiversionen wiederherzustellen.

Das bedeutet nicht, dass man sorglos arbeiten sollte. Es bedeutet aber, dass experimentelles Arbeiten kontrollierbar wird:

- Du kannst eine Idee ausprobieren.
- Du kannst Änderungen vergleichen.
- Du kannst einen Fehler isolieren.
- Du kannst einen funktionierenden Stand wiederherstellen.
- Du kannst eine falsche Entscheidung transparent korrigieren.

Damit wird das Projekt widerstandsfähiger gegenüber menschlichen Fehlern.

Parallel arbeiten

In professionellen Projekten arbeiten mehrere Personen gleichzeitig:

- Entwicklerinnen und Entwickler implementieren Features.
- Tester prüfen Fehlerkorrekturen.
- DevOps-Teams ändern Deployment-Konfigurationen.
- Technische Redaktionen aktualisieren Dokumentation.
- Sicherheitsverantwortliche prüfen Abhängigkeiten und Richtlinien.

Versionskontrolle koordiniert diese parallele Arbeit. Sie erkennt, wenn Änderungen unabhängig voneinander kombiniert werden können, und macht sichtbar, wenn zwei Personen denselben Bereich unterschiedlich verändert haben.

Solche Überschneidungen heißen später **Konflikte**. Ein Konflikt ist keine Beschädigung des Projekts, sondern ein Hinweis:

> Git kann nicht zuverlässig entscheiden, welche von zwei konkurrierenden Änderungen fachlich richtig ist.

Menschen treffen diese Entscheidung bewusst. Das ist deutlich sicherer, als Änderungen unbemerkt zu überschreiben.

Fehler schneller finden

Eine Projektgeschichte hilft nicht nur beim Zurücksetzen. Sie ist auch ein Diagnosewerkzeug.

Angenommen, eine API-Anfrage liefert seit kurzem falsche Daten. Mit einer Historie kannst du untersuchen:

1. Wann trat der Fehler erstmals auf?
2. Welche Änderungen wurden kurz davor integriert?
3. Welche Datei oder Funktion wurde verändert?
4. Welche Person kennt den fachlichen Hintergrund?
5. Welcher Commit führte die fehlerhafte Änderung ein?

Git bietet dafür später Funktionen wie Log-Ansichten, Diffs, Dateihistorien und `git bisect`. Sie helfen dabei, Fehler systematisch einzugrenzen, statt nur Vermutungen anzustellen.

Code-Reviews ermöglichen

Bevor eine Änderung in den Hauptzweig eines Projekts gelangt, sollte sie oft von einer anderen Person geprüft werden. Dieser Prozess heißt **Code Review**.

Ein Review funktioniert nur, wenn klar erkennbar ist:

- Welche Änderung wird vorgeschlagen?
- Welche Dateien sind betroffen?
- Was war das Ziel?
- Welche Tests wurden ergänzt oder ausgeführt?
- Welche Auswirkungen sind zu erwarten?

Versionskontrolle liefert dafür eine präzise Änderungsmenge. Statt ein gesamtes Projekt manuell zu vergleichen, prüfen Reviewer gezielt die Unterschiede zwischen zwei definierten Ständen.

Das verbessert nicht nur die Codequalität. Reviews fördern auch Wissenstransfer, gemeinsame Standards und frühzeitige Sicherheitsprüfungen.

Releases reproduzierbar machen

Software wird häufig in klaren Versionen veröffentlicht, etwa als `1.0.0`, `1.1.0` oder `2.0.0`.

Ohne Versionskontrolle ist es schwierig, sicher zu beantworten:

- Welcher Quellcode wurde für Version `1.4.2` ausgeliefert?
- Welche Konfiguration gehörte zu diesem Release?
- Welche Fehlerkorrektungen sind darin enthalten?
- Kann derselbe Stand erneut gebaut werden?
- Lässt sich ein Hotfix auf genau dieser Basis entwickeln?

Mit Versionierung kann ein Release eindeutig an einen bestimmten Projektzustand gebunden werden. Später werden dafür Git-Tags verwendet.

Das ist besonders wichtig, wenn eine ältere Produktversion weiterhin gewartet werden muss.

Wissen dauerhaft im Projekt festhalten

Teams verändern sich. Personen wechseln Aufgaben, verlassen ein Unternehmen oder erinnern sich nach Monaten nicht mehr an jede Entscheidung.

Eine gute Historie ergänzt Dokumentation. Sie kann beispielsweise zeigen:

- warum eine Validierung absichtlich streng ist,
- weshalb eine Abhängigkeit aktualisiert wurde,
- warum eine scheinbar unnötige Sonderbehandlung existiert,
- welches Issue einen Fehler beschrieben hat,
- welche Sicherheitslücke geschlossen wurde.

Natürlich ersetzt eine Commit-Nachricht keine vollständige Architektur- oder Fachkonzeptdokumentation. Sie hält aber den Zusammenhang zwischen einer konkreten Codeänderung und ihrer Entscheidung fest.

Versionskontrolle ist mehr als ein Backup

Backups und Versionskontrolle haben unterschiedliche Aufgaben.

Aspekt	Backup	Versionskontrolle
Hauptziel	Datenverlust verhindern	Änderungen nachvollziehbar verwalten
Typischer Umfang	Gesamter Datenbestand	Projektdateien und ihre Historie
Zeitpunkt	Meist automatisch oder periodisch	Bewusst bei fachlich sinnvollen Änderungen
Vergleich einzelner Änderungen	Oft umständlich	Zentrale Funktion
Zusammenarbeit	Nicht dafür ausgelegt	Zentrale Funktion
Rückkehr zu einer bestimmten Änderung	Häufig schwierig	Gezielt möglich
Dokumentation des Zwecks	Meist nicht vorhanden	Commit-Nachrichten und Referenzen

Ein Backup schützt beispielsweise vor einem defekten Datenträger, Ransomware oder versehentlich gelöschten Daten. Git schützt nicht zuverlässig vor allen diesen Risiken, insbesondere nicht allein auf einem einzelnen Computer.

Umgekehrt beantwortet ein Backup normalerweise nicht die Frage:

> „Welche Zeilen wurden geändert, als die Passwortvalidierung angepasst wurde?“

Die professionelle Praxis lautet deshalb:

> **Versionskontrolle und Backups ergänzen einander – keines ersetzt das andere.**

Ein Beispiel aus dem PHP-Alltag

Nehmen wir an, eine Methode zur E-Mail-Prüfung wird geändert.

Vorher:

```
function isValidEmail(string $email): bool
{
    return str_contains($email, '@');
}
```

Nachher:

```
function isValidEmail(string $email): bool
{
    return filter_var($email, FILTER_VALIDATE_EMAIL) !== false;
}
```

Ohne Versionskontrolle ist später nur sichtbar, wie die Datei *jetzt* aussieht. Der frühere Ansatz, der Zeitpunkt der Änderung und die Motivation gehen verloren.

Mit Versionskontrolle kann die Änderung als eigener Commit festgehalten werden:

```
Fix: E-Mail-Adressen mit PHP-Filter validieren
```

Zusätzlich können Tests im selben Commit dokumentieren, welche Fälle berücksichtigt wurden. Später lässt sich eindeutig nachvollziehen:

- welche Implementierung ersetzt wurde,
- warum die Änderung erfolgte,
- welche Dateien dazugehören,
- und ob die Änderung mit einem Bug-Report verknüpft war.

Versionierung unterstützt kleine und große Projekte

Ein verbreiteter Irrtum lautet:

> „Git brauche ich erst, wenn mehrere Personen am Projekt arbeiten.“

Tatsächlich profitieren Einzelentwickler besonders früh von Versionierung.

Für Einzelentwickler

Git hilft dir,

- Experimente sicher auszuprobieren,
- Änderungen in kleinen Schritten festzuhalten,
- Fehler auf frühere Änderungen zurückzuführen,
- zwischen mehreren Aufgaben zu wechseln,
- ältere Zustände wiederzufinden,
- und ein Portfolio mit nachvollziehbarer Entwicklung zu pflegen.

Für Teams

Zusätzlich unterstützt Git,

- parallele Feature-Entwicklung,
- Pull Requests und Code Reviews,
- geregelte Freigaben,
- Release-Management,
- CI-Prüfungen,
- Branch-Schutz,
- und eine gemeinsame, transparente Projektgeschichte.

Der Einstieg lohnt sich daher bereits beim ersten ernsthaften Projekt.

Was genau sollte versioniert werden?

Grundsätzlich gehören Dateien in die Versionskontrolle, wenn sie nötig sind, um das Projekt zu verstehen, zu entwickeln, zu testen oder reproduzierbar zu bauen.

Für ein PHP-Projekt sind das häufig:

```
src/
tests/
config/
public/
composer.json
composer.lock
phpunit.xml
README.md
.github/
```

Nicht versioniert werden sollten typischerweise Dateien, die lokal erzeugt werden, vertrauliche Daten enthalten oder sich jederzeit reproduzieren lassen, beispielsweise:

```
vendor/
.idea/workspace.xml
.env
var/cache/
var/log/
```

Welche Dateien ignoriert werden sollen, hängt vom Projekt ab. Die Regeln dafür werden später mit `.gitignore` systematisch behandelt.

Wichtig ist zunächst das Prinzip:

> **Versioniere den gemeinsamen, nachvollziehbaren Projektzustand – nicht persönliche Arbeitsreste oder geheime Zugangsdaten.**

Gute Versionierung beginnt mit kleinen, sinnvollen Schritten

Versionskontrolle entfaltet ihren größten Nutzen, wenn Änderungen klar abgegrenzt werden.

Ungünstig wäre ein riesiger Sammel-Commit wie:

```
Diverse Änderungen
```

Darin könnten sich gleichzeitig befinden:

- ein neues Feature,
- eine Fehlerkorrektur,
- Formatierungsänderungen,
- aktualisierte Abhängigkeiten,
- und eine versehentlich eingecheckte Konfigurationsdatei.

Besser sind kleine, logisch zusammenhängende Schritte:

```
Feat: Terminvalidierung ergänzen
Test: Ungültige Endzeiten abdecken
Fix: Zeitzone beim Speichern berücksichtigen
Docs: API-Beispiel für Terminanlage ergänzen
```

Dadurch wird die Historie lesbar, überprüfbar und leichter rückgängig zu machen.

Git als Werkzeug für diese Aufgabe

Git ist ein verteiltes Versionskontrollsystem. Es speichert die Projektgeschichte lokal auf deinem Computer und kann sie mit anderen Repositories austauschen, etwa über GitHub.

Dabei ist Git nicht einfach ein Cloud-Speicher und auch nicht nur ein Werkzeug für Befehlszeilenprofis. Es bildet die Grundlage für viele alltägliche Entwicklungsabläufe:

- lokale Änderungen sicher festhalten,
- neue Entwicklungszweige erstellen,
- Änderungen anderer abrufen,
- Pull Requests prüfen,
- Releases markieren,

- und Fehler bis zu ihrer Ursache zurückverfolgen.

PhpStorm stellt viele dieser Funktionen grafisch bereit. Dennoch ist es wichtig, die zugrunde liegenden Konzepte zu verstehen. Dann kannst du sicher entscheiden, welche Aktion sinnvoll ist – unabhängig davon, ob du die IDE oder die Kommandozeile verwendest.

Merksätze

- **Softwareentwicklung ist Veränderung.** Ohne Historie werden Veränderungen schnell unübersichtlich.
- **Versionskontrolle beantwortet Fragen.** Was, wann, wer und warum sind zentrale Informationen.
- **Git ist kein Ersatz für Backups.** Beide Schutzmechanismen werden benötigt.
- **Auch Einzelentwickler profitieren.** Git macht Experimente, Korrekturen und Fehlersuche sicherer.
- **Kleine, zusammenhängende Änderungen erzeugen eine wertvolle Historie.**
- **Versionskontrolle macht Zusammenarbeit kontrollierbar, nicht automatisch konfliktfrei.**
- **Ein Commit ist ein bewusst dokumentierter Projektzustand**, nicht bloß ein beliebiger Speichervorgang.

Selbstcheck

Prüfe dein Verständnis mit diesen Fragen:

1. Warum sind Ordnerkopien mit Namen wie `projekt_final2` keine zuverlässige Versionsstrategie?
2. Welche Fragen sollte eine gute Projektgeschichte beantworten können?
3. Worin unterscheidet sich Versionskontrolle von einem Backup?
4. Warum ist Git auch bei einem Solo-Projekt sinnvoll?
5. Weshalb sollten unabhängige Änderungen nicht in einem einzigen großen Commit zusammengefasst werden?
6. Welche Arten von Dateien gehören in einem PHP-Projekt normalerweise *nicht* ins Repository?

Im nächsten Abschnitt wird betrachtet, wie Änderungen, Versionen und Projektgeschichte konkret zusammenhängen.

Änderungen, Versionen und Projektgeschichte

Änderungen sind die Rohstoffe der Entwicklung

Software entsteht nicht in einem einzigen Schritt. Sie entwickelt sich fortlaufend:

- Eine Funktion wird ergänzt.
- Ein Fehler wird korrigiert.
- Eine Abhängigkeit wird aktualisiert.
- Eine Konfiguration wird angepasst.
- Dokumentation wird verbessert.
- Code wird umstrukturiert, ohne sein Verhalten zu ändern.

Jede dieser Anpassungen ist eine **Änderung**. Für sich genommen kann sie klein sein; über Wochen, Monate und Jahre bilden Änderungen jedoch die Geschichte eines Projekts.

Ohne Versionskontrolle bleibt diese Geschichte meist unsichtbar. Dateien werden überschrieben, Kopien angelegt oder Änderungen nur mündlich kommuniziert. Das funktioniert kurzzeitig, wird aber schnell unsicher.

Stell dir vor, eine Datei `Calculator.php` wird geändert:

```
public function add(int $a, int $b): int
{
    return $a + $b;
}
```

Später ergänzt jemand eine Eingabeprüfung:

```
public function add(int $a, int $b): int
{
    if ($a < 0 || $b < 0) {
        throw new InvalidArgumentException('Negative Werte sind nicht erlaubt.');
```

```
    }

    return $a + $b;
}
```

Die zweite Variante ist nicht einfach „die Datei“. Sie ist eine *neue Fassung* derselben Datei. Für ein einzelnes Beispiel ist der Unterschied offensichtlich. In einem Projekt mit hunderten Dateien, mehreren Entwicklerinnen und Entwicklern und vielen parallelen Aufgaben ist er ohne Werkzeug kaum zuverlässig nachzuvollziehen.

Was eine Version bedeutet

Eine **Version** ist ein eindeutig nachvollziehbarer Zustand eines Projekts zu einem bestimmten Zeitpunkt.

Dabei geht es nicht nur um eine einzelne Datei. Eine Version kann den gemeinsamen Zustand vieler Bestandteile beschreiben:

- PHP-Quellcode,
- Tests,
- Composer-Konfiguration,
- Dokumentation,
- Konfigurationsdateien,
- CI-Workflows,
- Datenbankschemata,
- Build-Skripte.

Eine Version beantwortet Fragen wie:

- Welcher Code war letzte Woche produktiv?
- Wann wurde dieser Fehler eingeführt?
- Welche Dateien gehörten zu Version `2.1.0`?
- Welche Änderung hat ein Verhalten verändert?
- Wer hat diese Anpassung vorgenommen und warum?
- Wie kann ein früherer funktionierender Stand wiederhergestellt werden?

Eine Versionsnummer wie `1.4.2` ist dabei eine **fachliche Kennzeichnung**, etwa für ein Release. Git selbst verwaltet dagegen technische Zustände über Commits und eindeutige Kennungen. Auf diesen Unterschied wird der Kurs später noch detailliert eingehen.

> **Wichtig:** Eine Version ist nicht zwingend eine veröffentlichte Produktversion. Auch jeder kleine, lokale Entwicklungsschritt kann eine nachvollziehbare Version sein.

Von Dateikopien zur echten Projektgeschichte

Ohne Versionskontrolle sieht die Entwicklung eines Projekts oft ungefähr so aus:

```
projekt/
├── index.php
├── index_alt.php
├── index_neu.php
├── index_final.php
├── index_final_final.php
└── index_final_wirklich_final.php
```

Oder es entstehen Ordner wie:

```
shopprojekt/
├── backup/
├── backup_neu/
├── backup_vor_update/
├── stand_vom_freitag/
└── produktiv_final/
```

Diese Methode hat erkennbare Nachteile:

1. **Unklare Herkunft**
   Es ist nicht mehr klar, welche Kopie die richtige ist.
2. **Keine Begründung**
   Dateinamen erklären selten, *warum* etwas geändert wurde.
3. **Schwierige Vergleiche**
   Unterschiede zwischen zwei Ordnern oder Dateien müssen mühsam gesucht werden.
4. **Keine sichere Zusammenarbeit**
   Mehrere Personen überschreiben leicht gegenseitig ihre Änderungen.

5. **Unvollständige Sicherung**
   Häufig werden nur einzelne Dateien kopiert. Zusammenhängende Änderungen fehlen dann.
6. **Keine saubere Rückkehr**
   Ein alter Stand kann vielleicht gefunden werden, aber nicht zuverlässig wiederhergestellt werden.

Git ersetzt diese unstrukturierten Kopien durch eine präzise Projektgeschichte. Statt Dateien mit Zusätzen wie `_alt` oder `_neu` zu speichern, hält Git fest:

- *welche* Dateien geändert wurden,
- *was* sich darin geändert hat,
- *wann* die Änderung gespeichert wurde,
- *wer* sie erstellt hat,
- *warum* sie vorgenommen wurde,
- und *auf welchem vorherigen Stand* sie aufbaut.

Änderungen als nachvollziehbare Einheiten

In professionellen Projekten wird nicht jede Tastatureingabe sofort als dauerhafte Version gespeichert. Stattdessen fasst man logisch zusammenhängende Änderungen zu einer Einheit zusammen.

Ein Beispiel: Für eine neue Passwortprüfung könnten diese Änderungen zusammengehören:

- `PasswordValidator.php` wird ergänzt,
- passende Unit-Tests werden erstellt,
- eine Fehlermeldung wird dokumentiert,
- eine Konfiguration wird angepasst.

Diese gemeinsame Änderung sollte später als *eine nachvollziehbare Einheit* sichtbar sein. In Git heißt eine solche gespeicherte Einheit **Commit**.

Eine passende Beschreibung könnte lauten:

```
feat(auth): Passwortlänge validieren
```

Nicht hilfreich wären dagegen Nachrichten wie:

```
Update
```

```
Änderungen
```

```
Fix
```

Die Qualität einer Projektgeschichte hängt wesentlich davon ab, ob Änderungen sinnvoll zusammengefasst und verständlich beschrieben werden.

Gute Änderungseinheit

Eine gute Änderungseinheit beantwortet idealerweise eine konkrete Frage:

> „Was wurde mit diesem Schritt erreicht?“

Beispiele:

- „E-Mail-Adresse bei der Registrierung validieren“
- „Fehlerhafte Berechnung der Mehrwertsteuer korrigieren“
- „PHPUnit auf Version 11 aktualisieren“
- „README um lokale Installationsschritte ergänzen“
- „Nicht verwendete Legacy-Klasse entfernen“

Zu große Änderungseinheit

Folgende Mischung ist problematisch:

```
Neue Login-Funktion, CSS-Anpassungen, Composer-Update,
Tippfehlerkorrekturen und Datenbank-Migration
```

Diese Änderung enthält mehrere unabhängige Themen. Wenn später ein Fehler in der Login-Funktion auftritt, möchte man nicht gleichzeitig CSS-Anpassungen oder ein Abhängigkeitsupdate zurücknehmen müssen.

Zu kleine Änderungseinheit

Auch das Gegenteil kann unpraktisch sein:

```
Leerzeile in Datei A ergänzt
```

```
Leerzeile in Datei B ergänzt
```

```
Leerzeile in Datei C ergänzt
```

Nicht jede minimale Änderung braucht einen eigenen Commit. Entscheidend ist, dass die Historie fachlich verständlich bleibt.

Projektgeschichte als Kette von Entscheidungen

Eine Projektgeschichte ist mehr als eine Liste technischer Dateiunterschiede. Sie dokumentiert Entscheidungen.

Angenommen, ein Projekt entwickelt sich in diesen Schritten:

```
A  Projektgrundstruktur anlegen
B  Composer-Abhängigkeiten hinzufügen
C  Benutzerregistrierung implementieren
D  E-Mail-Validierung ergänzen
E  Validierungsfehler korrigieren
F  Version 1.0.0 veröffentlichen
```

Diese Abfolge erklärt nicht nur, *dass* sich Dateien verändert haben. Sie zeigt auch, wie das Projekt zu seinem aktuellen Zustand gelangt ist.

Vereinfacht kann man die Entwicklung als Kette darstellen:

```
A — B — C — D — E — F
```

Jeder Schritt baut auf einem vorherigen Stand auf. Die aktuelle Version enthält also die Ergebnisse aller vorherigen Schritte.

Das ist ein wichtiges Denkmodell:

> Die Projektgeschichte ist keine Ansammlung zufälliger Sicherungskopien, sondern eine Folge bewusst gespeicherter Zustände und Entscheidungen.

Später können sich solche Entwicklungslinien aufteilen und wieder zusammengeführt werden. Git verwendet dafür **Branches** und **Merges**. Zunächst genügt die Vorstellung einer linearen Geschichte.

Momentaufnahme und Änderung unterscheiden

Im Alltag sprechen Teams oft davon, „eine Änderung zu speichern“. Git arbeitet intern jedoch vor allem mit **Momentaufnahmen** des Projektzustands.

Bei einem Commit hält Git fest, wie die verfolgten Dateien zu diesem Zeitpunkt aussehen. Der Commit verweist außerdem auf seinen Vorgänger. Dadurch kann Git Unterschiede zwischen Zuständen berechnen.

Vereinfacht:

```
Commit A: Grundstruktur
Commit B: Grundstruktur + Composer-Dateien
Commit C: Grundstruktur + Composer-Dateien + Registrierung
```

Git muss nicht für jede Version eine vollständige Ordnerkopie neben die vorherige legen. Es speichert Inhalte effizient und organisiert sie so, dass frühere Zustände weiterhin erreichbar bleiben.

Für die praktische Arbeit ist zunächst diese Formulierung hilfreich:

- Eine **Änderung** beschreibt, was du bearbeitet hast.
- Ein **Commit** speichert einen nachvollziehbaren Projektzustand.
- Die **Historie** verbindet diese Zustände miteinander.

Warum die Geschichte im Alltag wichtig ist

Eine gute Historie hilft nicht nur bei großen Problemen. Sie ist ein tägliches Arbeitswerkzeug.

Fehlerursachen finden

Ein Test schlägt plötzlich fehl. Nun stellt sich die Frage:

> „Seit wann ist dieses Verhalten vorhanden?“

Mit der Historie kannst du prüfen:

- Welche Änderungen wurden zuletzt vorgenommen?
- Welche Datei wurde verändert?
- In welchem Commit wurde die betreffende Zeile eingeführt?
- Welche Begründung stand in der Commit-Nachricht?

Statt zu raten, untersuchst du konkrete Informationen.

Frühere Zustände vergleichen

Vielleicht funktioniert ein Formular in der aktuellen Version nicht mehr, in der Vorversion aber schon. Dann vergleichst du beide Stände.

Beispielhaft:

```
Version vorher: Formular sendet Daten korrekt
Version aktuell: Validierung verhindert das Absenden
```

Ein Vergleich zeigt, welche Codezeilen zwischen diesen Zuständen verändert wurden.

Änderungen gezielt zurücknehmen

Nicht jede Änderung soll dauerhaft bleiben. Vielleicht wurde eine Funktion falsch umgesetzt oder ein Update verursacht Probleme.

Ohne Versionskontrolle müsstest du alte Dateien suchen und manuell zurückkopieren. Mit Git kannst du gezielt feststellen:

- welcher Commit problematisch ist,
- welche Dateien er verändert hat,
- und wie die Änderung kontrolliert zurückgenommen werden kann.

Wissen im Team erhalten

Menschen wechseln Projekte, vergessen Details oder sind im Urlaub. Eine verständliche Historie bewahrt Kontext.

Statt jemanden fragen zu müssen, warum ein ungewöhnlicher Codeabschnitt existiert, kann die Historie Hinweise liefern:

```
fix(api): Timeout für langsame Zahlungsanbieter erhöhen
```

Vielleicht wurde diese Anpassung wegen eines konkreten Produktionsproblems vorgenommen. Eine gute Commit-Nachricht, ein verknüpftes Issue oder ein Pull Request machen diese Entscheidung später nachvollziehbar.

Releases reproduzieren

Wenn ein Kunde einen Fehler in Version `1.8.0` meldet, muss das Team genau diesen Stand untersuchen können — nicht nur die heutige, bereits weiterentwickelte Variante.

Versionskontrolle ermöglicht es, ältere Stände gezielt zu prüfen, Tests auszuführen und bei Bedarf eine Korrektur auf dieser Basis zu entwickeln.

Die Zeitachse eines kleinen Projekts

Ein einfaches PHP-Projekt könnte sich so entwickeln:

```
1. Projektstruktur erstellen
2. Composer konfigurieren
3. Erste Klasse schreiben
4. Tests hinzufügen
5. Fehler korrigieren
6. Dokumentation ergänzen
7. Release veröffentlichen
```

Als Historie könnte das aussehen:

```
a1b2c3d  Projektstruktur anlegen
b2c3d4e  Composer und Autoloading konfigurieren
c3d4e5f  Preisberechnung implementieren
d4e5f6a  Tests für Preisberechnung ergänzen
e5f6a7b  Rundungsfehler bei Mehrwertsteuer korrigieren
f6a7b8c  README um Installationsanleitung erweitern
g7b8c9d  Release 1.0.0 vorbereiten
```

Die Zeichenfolgen am Anfang stehen beispielhaft für Commit-IDs. Git verwendet sie, um jeden gespeicherten Zustand eindeutig zu identifizieren.

An dieser Liste lässt sich bereits viel ablesen:

- Die Preisberechnung wurde *vor* den Tests implementiert.
- Danach wurde ein Rundungsfehler entdeckt und behoben.
- Die Dokumentation wurde erst später ergänzt.
- Das Release baut auf allen vorherigen Änderungen auf.

Eine solche Historie ist wesentlich wertvoller als ein Ordner mit dem Namen `projekt_neu_final`.

Änderungen haben Kontext

Eine Zeile Code allein erklärt selten genug. Der Kontext einer Änderung ist oft mindestens genauso wichtig wie der technische Unterschied.

Betrachte diese Änderung:

```
private const TAX_RATE = 0.19;
```

Wird daraus später:

```
private const TAX_RATE = 0.20;
```

ist der technische Unterschied klein. Die entscheidende Frage lautet aber:

> Warum wurde der Wert geändert?

Mögliche Antworten:

- Eine gesetzliche Änderung tritt zu einem bestimmten Datum in Kraft.
- Der Wert war bisher fehlerhaft.
- Die Konstante wird nur in einer Testumgebung verwendet.
- Eine neue Konfiguration ersetzt den festen Wert.

Versionskontrolle kann diesen Kontext über Commit-Nachrichten, Issues, Pull Requests und Reviews festhalten. Git speichert primär die technische Historie; Plattformen wie GitHub ergänzen sie um Diskussionen, Aufgaben und Freigaben.

Die Projektgeschichte ist kein Ersatz für Kommunikation

Eine Commit-Historie ist wertvoll, aber sie ersetzt nicht alles.

Sie kann dokumentieren:

- was geändert wurde,
- wann es geschah,
- wer die Änderung gespeichert hat,
- welche Nachricht dazu hinterlegt wurde.

Sie kann jedoch nicht automatisch alle fachlichen Hintergründe erfassen. Wichtige Architekturentscheidungen, Sicherheitsannahmen oder Produktentscheidungen gehören häufig zusätzlich in:

- Issues,
- Pull-Request-Beschreibungen,
- Architekturentscheidungsprotokolle,
- technische Dokumentation,
- Ticketsysteme,
- das Projekt-README.

Eine gute Praxis ist daher:

1. **Commits klein und verständlich halten.**
2. **Commit-Nachrichten konkret formulieren.**
3. **Größere Entscheidungen dokumentieren.**
4. **Zusammenhänge zwischen Code, Issue und Pull Request verknüpfen.**

Eine Geschichte darf nicht beliebig sein

Nicht jede Projektgeschichte ist automatisch gut, nur weil Git verwendet wird. Auch ein Git-Repository kann unübersichtlich werden.

Typische Warnzeichen sind:

- Commit-Nachrichten wie „WIP“, „Test“, „Änderung“ oder „asdf“,
- riesige Commits mit vielen unabhängigen Themen,
- fehlende Tests bei funktionalen Änderungen,
- Zugangsdaten oder generierte Dateien in der Historie,
- unklare oder widersprüchliche Branch-Namen,
- häufiges Überschreiben bereits veröffentlichter Historie.

Ein Ziel professioneller Versionskontrolle lautet deshalb:

> Die Historie soll nicht nur vollständig, sondern auch verständlich, überprüfbar und sicher sein.

Das bedeutet nicht, dass jeder Commit perfekt sein muss. Gerade beim Lernen oder bei lokaler Experimentierarbeit entstehen unfertige Zwischenstände. Vor einer gemeinsamen Veröffentlichung können diese jedoch oft sinnvoll geordnet, zusammengefasst oder dokumentiert werden.

Praktisches Denkmodell: Fragen an jede Änderung

Bevor du eine Änderung dauerhaft speicherst, helfen diese Fragen:

1. **Was habe ich geändert?**
   Beschreibe die technische oder fachliche Wirkung klar.
2. **Warum war die Änderung nötig?**
   Gibt es einen Fehler, ein Ticket, eine Anforderung oder eine Entscheidung?
3. **Gehört alles in einen gemeinsamen Schritt?**
   Oder lassen sich unabhängige Änderungen trennen?

4. **Ist der neue Zustand funktionsfähig?**
   Wurden Tests ausgeführt oder zumindest die Auswirkungen geprüft?
5. **Kann eine andere Person die Änderung später verstehen?**
   Reicht die Commit-Nachricht aus, oder braucht es zusätzliche Dokumentation?
6. **Kann die Änderung bei Bedarf isoliert rückgängig gemacht werden?**
   Kleine, klar abgegrenzte Commits erleichtern das erheblich.

Diese Fragen machen aus „Dateien speichern“ eine professionelle, nachvollziehbare Arbeitsweise.

Git und die unveränderliche Vergangenheit

Ein zentraler Vorteil von Git ist, dass frühere Commits normalerweise nicht einfach überschrieben werden. Neue Arbeit baut auf bestehenden Zuständen auf.

Wenn du heute einen Commit erstellst, bleibt der vorherige Zustand weiterhin Teil der Historie:

```
Vorheriger Stand ── Neue Änderung ── Aktueller Stand
```

Dadurch kannst du:

- frühere Versionen untersuchen,
- Änderungen vergleichen,
- ältere Stände wiederherstellen,
- Fehler zeitlich eingrenzen,
- Releases dauerhaft markieren.

Später wirst du lernen, dass Git-Historien unter bestimmten Umständen umgeschrieben werden können, etwa mit `git rebase` oder speziellen Bereinigungswerkzeugen. Das sind jedoch bewusste, teilweise riskante Operationen — insbesondere bei bereits mit anderen geteilten Commits.

Für den Einstieg gilt:

> Neue Arbeit wird als neuer nachvollziehbarer Schritt ergänzt, nicht durch das unkontrollierte Überschreiben alter Arbeit ersetzt.

Zusammenfassung

Änderungen bilden die Grundlage jeder Softwareentwicklung. Versionskontrolle macht aus diesen Änderungen eine zuverlässige Projektgeschichte.

Die wichtigsten Begriffe:

- **Änderung:** Eine Anpassung an Dateien oder Projektinhalten.
- **Version:** Ein nachvollziehbarer Zustand des gesamten Projekts zu einem bestimmten Zeitpunkt.
- **Commit:** Eine in Git gespeicherte, beschriebene Momentaufnahme eines Projektzustands.
- **Historie:** Die geordnete Verbindung aller Commits und damit die Entwicklungsgeschichte des Projekts.

Eine gute Projektgeschichte hilft dir dabei,

- Fehler schneller zu verstehen,
- ältere Stände sicher wiederzufinden,
- Änderungen gezielt zurückzunehmen,
- Releases reproduzierbar zu machen,
- und Wissen im Team langfristig zu bewahren.

Im nächsten Schritt wird deutlich, wie unterschiedliche Systeme diese Geschichte organisieren: zentral auf einem Server oder verteilt auf die Rechner aller Beteiligten.

Zentrale und verteilte Versionskontrolle

Lernziele

Nach diesem Abschnitt kannst du:

- zentrale und verteilte Versionskontrollsysteme unterscheiden,
- den grundlegenden Ablauf beider Modelle erklären,
- Vorteile, Grenzen und Risiken beider Ansätze bewerten,
- einordnen, warum Git als *verteiltes* Versionskontrollsystem konzipiert ist,
- verstehen, weshalb GitHub trotz seiner zentralen Rolle nicht aus Git ein zentrales System macht.

Zwei grundlegende Modelle

Versionskontrollsysteme organisieren Änderungen an Dateien und machen die Entwicklungsgeschichte nachvollziehbar. Dabei haben sich zwei grundlegende Architekturmodelle etabliert:

1. **Zentrale Versionskontrolle**: Es gibt ein maßgebliches Repository auf einem Server.
2. **Verteilte Versionskontrolle**: Jede lokale Kopie enthält grundsätzlich die vollständige Repository-Historie.

Der entscheidende Unterschied betrifft nicht nur den Speicherort der Daten. Er verändert auch, **wann** Teams miteinander kommunizieren müssen, wie sie offline arbeiten können, wie sicher ihre Historie ist und welche Arbeitsabläufe sinnvoll sind.

Zentrale Versionskontrolle

Bei einem zentralen Versionskontrollsystem liegt die vollständige Projektgeschichte auf einem zentralen Server. Entwicklerinnen und Entwickler laden Dateien daraus herunter, bearbeiten sie

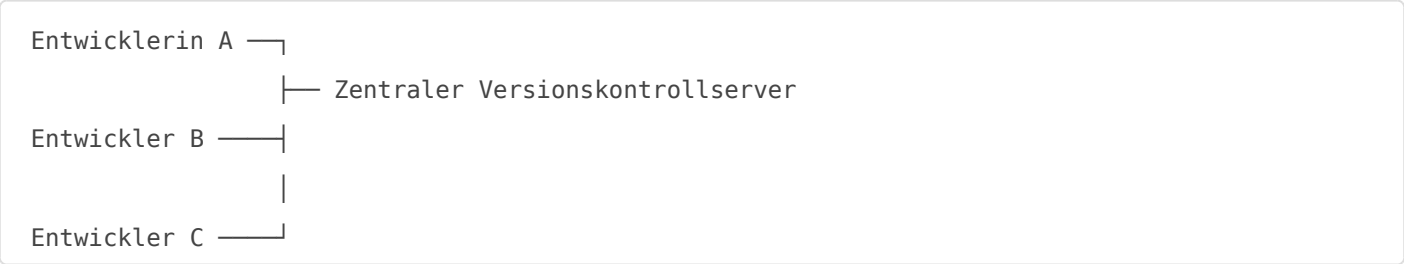
lokal und übertragen ihre Änderungen wieder zurück auf den Server.

Typische zentrale Systeme sind:

- Subversion, meist über den Befehl `svn`
- Perforce
- Team Foundation Version Control, kurz TFVC
- ältere Systeme wie CVS

Grundprinzip

Die Arbeitskopie auf dem lokalen Computer enthält in der Regel die aktuelle Version der Dateien, aber nicht zwingend die vollständige Historie aller Versionen. Der Server ist die zentrale Wahrheit über den Projektzustand.



Ein typischer Ablauf sieht so aus:

1. Änderungen vom Server abrufen.
2. Dateien lokal bearbeiten.
3. Vor dem Hochladen prüfen, ob andere Personen Änderungen veröffentlicht haben.
4. Eigene Änderungen an den zentralen Server senden.
5. Der Server speichert die neue offizielle Revision.

Typischer zentraler Workflow

In Subversion wären die zentralen Aktionen beispielsweise:

```
svn update
svn commit -m "Validierung für E-Mail-Adresse ergänzt"
```

Die exakte Syntax ist hier weniger wichtig als das Denkmodell: Ohne Verbindung zum Server sind viele zentrale Aktionen nicht möglich.

Vorteile zentraler Systeme

Zentrale Versionskontrolle kann gerade in klar geregelten Umgebungen sinnvoll sein.

Ein eindeutig zentraler Projektstand

Es gibt einen klaren Ort, an dem sich die offizielle Historie befindet. Teams müssen nicht entscheiden, welches Repository maßgeblich ist: Der zentrale Server ist es per Definition.

Das vereinfacht manche organisatorischen Prozesse:

- Berechtigungen werden zentral verwaltet.
- Backups konzentrieren sich auf einen Server.
- Regeln für Änderungen lassen sich zentral erzwingen.
- Der aktuelle Projektstand ist an einer Stelle sichtbar.

Einfaches mentales Modell

Für Einsteigerinnen und Einsteiger wirkt das Modell häufig intuitiv:

> „Ich lade die aktuelle Version herunter, ändere etwas und speichere es wieder auf dem Server.“

Die Abläufe ähneln der Arbeit mit einer gemeinsam genutzten Dateiablage, auch wenn Versionskontrolle deutlich leistungsfähiger ist.

Serverseitige Kontrolle

Unternehmen mit strengen Vorgaben können zentrale Serverregeln nutzen, etwa:

- verpflichtende Prüfungen vor einem Commit,

- zentrale Zugriffsrechte,
- Audit-Protokolle,
- verbindliche Dateisperren für bestimmte Dateitypen.

Besonders bei großen Binärdateien oder Dateien, die sich kaum sinnvoll zusammenführen lassen, können Sperrmechanismen nützlich sein. Beispielsweise soll nicht gleichzeitig an derselben Photoshop-Datei gearbeitet werden.

Grenzen zentraler Systeme

Die Einfachheit des Modells bringt Abhängigkeiten mit sich.

Abhängigkeit vom Server

Ist der zentrale Server nicht erreichbar, sind zentrale Funktionen eingeschränkt oder unmöglich. Das kann passieren durch:

- Netzwerkprobleme,
- Wartungsarbeiten,
- VPN-Ausfälle,
- einen Serverausfall,
- fehlende Internetverbindung auf Reisen.

Lokale Änderungen sind zwar häufig weiterhin möglich, aber das Übertragen, Abrufen und oft auch das komfortable Anzeigen der Historie hängt vom Server ab.

Eingeschränkte Offline-Arbeit

In einem zentralen System kann ein Commit direkt eine Serveroperation sein. Ohne Netzwerk lässt sich die Änderung daher nicht als vollwertiger, gemeinsamer Versionsstand speichern.

Das führt oft dazu, dass Entwicklerinnen und Entwickler größere Änderungspakete sammeln und später gesammelt übertragen. Große Pakete sind aber schwieriger zu prüfen, zu testen und bei Problemen zurückzunehmen.

Zentraler Ausfallpunkt

Der Server ist ein sogenannter *Single Point of Failure*: Fällt er aus und existiert kein funktionierendes Backup, ist die Projektgeschichte gefährdet.

Ein zentraler Server kann selbstverständlich professionell gesichert werden. Dennoch bleibt seine Verfügbarkeit für den täglichen Arbeitsablauf entscheidend.

Branches können teuer sein

Bei einigen älteren zentralen Systemen sind Branches technisch oder organisatorisch schwergewichtiger. Teams vermeiden sie dann häufig und arbeiten lange auf gemeinsamen Entwicklungszweigen.

Das erhöht die Wahrscheinlichkeit, dass parallele Änderungen miteinander kollidieren.

> **Merksatz:** In zentralen Systemen ist Zusammenarbeit oft stärker an den gemeinsamen Server und dessen Verfügbarkeit gebunden.

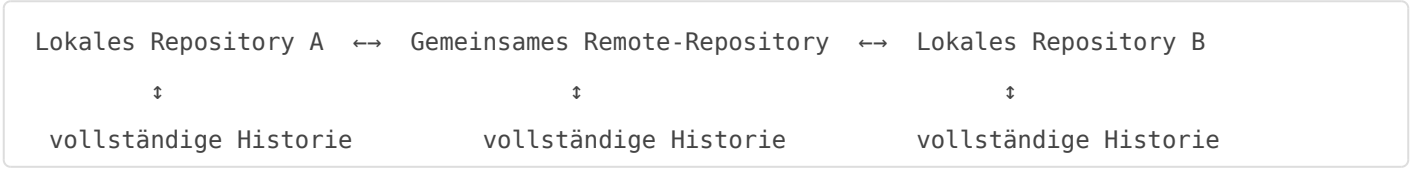
Verteilte Versionskontrolle

Ein verteiltes Versionskontrollsystem speichert die vollständige Projektgeschichte nicht nur auf einem Server, sondern auch lokal bei den Beteiligten.

Git ist das bekannteste verteilte Versionskontrollsystem. Weitere Beispiele sind Mercurial und Fossil.

Wenn du ein Git-Repository klonst, erhältst du nicht bloß die aktuell sichtbaren Dateien. Du erhältst normalerweise auch:

- die Commit-Historie,
- Branches und Tags,
- die Git-Objekte,
- lokale Referenzen,
- die Informationen, die Git für Vergleiche, Branches und viele Wiederherstellungen benötigt.



Das Wort *verteilt* bedeutet nicht, dass es keinen zentralen Server geben darf. Es bedeutet vielmehr:

>

> Jedes vollständige Git-Repository ist grundsätzlich eine eigenständige Kopie der Projektgeschichte.

Das lokale Repository als vollwertige Kopie

Ein Git-Repository besteht aus mehr als dem Projektordner mit den sichtbaren Dateien. Im versteckten Verzeichnis `.git` verwaltet Git unter anderem:

- Commits,
- Branch-Referenzen,
- Tags,
- die Staging Area,
- Konfiguration,
- lokale Bewegungsprotokolle, die Reflogs.

Beim Klonen eines Repositorys entsteht daher eine lokale, funktionsfähige Datenbasis.

```
Projektordner
├── src
├── tests
├── composer.json
└── .git
    ├── objects
    ├── refs
    ├── HEAD
    ├── index
    └── config
```

Die Details dieser Dateien und Verzeichnisse werden in späteren Kapiteln vertieft. Für den Moment genügt dieses Verständnis:

- Die sichtbaren Projektdateien bilden dein **Arbeitsverzeichnis**.
- Das `.git`-Verzeichnis enthält die **lokale Versionsdatenbank**.

Lokale Commits ohne Netzwerk

Ein entscheidender Vorteil von Git ist, dass ein Commit lokal erstellt wird. Dafür ist weder GitHub noch ein Unternehmensserver erforderlich.

```
git add src/EmailValidator.php
git commit -m "Validierung für E-Mail-Adresse ergänzen"
```

Dieser Commit wird zunächst nur im lokalen Repository gespeichert.

Erst mit einem Push überträgst du ihn in ein anderes Repository, beispielsweise zu GitHub:

```
git push origin main
```

Die Begriffe beschreiben unterschiedliche Vorgänge:

Aktion	Bedeutung
`git commit`	Speichert einen neuen Versionsstand **lokal**
`git push`	Überträgt lokale Commits in ein entferntes Repository
`git fetch`	Lädt Informationen und Commits aus einem entfernten Repository herunter
`git pull`	Ruft Änderungen ab und integriert sie in den aktuellen lokalen Branch

Diese Trennung ist ein Kernmerkmal verteilter Versionskontrolle.

Typischer Git-Workflow

Ein einfacher Git-Workflow kann so aussehen:

```
Dateien lokal ändern
        ↓
Änderungen auswählen und vormerken
        ↓
Lokalen Commit erstellen
        ↓
Tests lokal ausführen
        ↓
```

```
Commit zu GitHub oder einem anderen Remote pushen
```

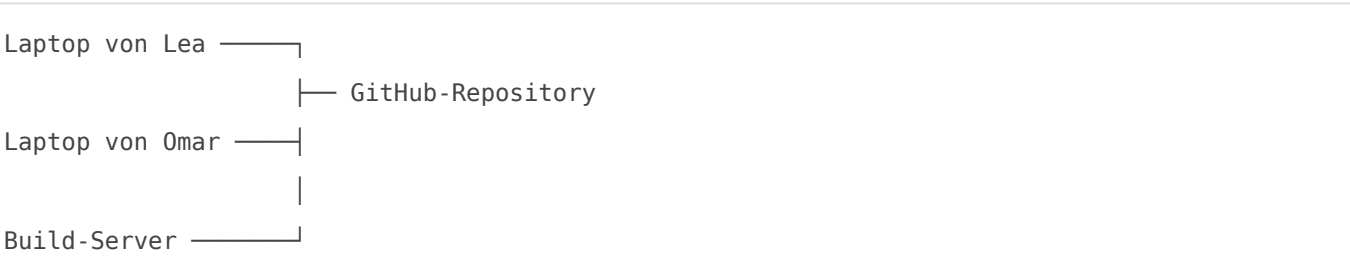
Als Befehlsfolge:

```
git status
git add src/EmailValidator.php
git commit -m "Validierung für E-Mail-Adresse ergänzen"
git push
```

Zwischen `git commit` und `git push` können Sekunden, Stunden oder auch Tage liegen. Ob das sinnvoll ist, hängt vom Teamworkflow ab. Technisch ist es möglich, weil der Commit bereits lokal existiert.

Remote-Repositorys sind nicht „der Git-Server“

In der Praxis verwenden Teams oft ein gemeinsames Remote-Repository als Koordinationspunkt. Das kann bei GitHub, GitLab, Bitbucket oder auf einem eigenen Server liegen.



Dieses GitHub-Repository ist für das Team häufig die **gemeinsame Referenz**. Es ist aber technisch nicht das einzige Repository, das die Historie enthält.

Auch Leas und Omars lokale Repositories enthalten eine vollständige Historie. GitHub ergänzt Git um Funktionen für Zusammenarbeit und Plattformbetrieb, zum Beispiel:

- Pull Requests,
- Code Reviews,
- Issues,
- Rechteverwaltung,
- GitHub Actions,
- Releases,
- Sicherheitsprüfungen.

GitHub ist damit für viele Teams organisatorisch zentral, Git selbst bleibt jedoch ein verteiltes System.

> **Wichtig:** „Wir arbeiten mit GitHub“ bedeutet nicht, dass Git zentralisiert wäre. GitHub ist ein Remote und eine Kollaborationsplattform für Git-Repositories.

Mehrere Remotes verwenden

Ein Git-Repository kann mit mehreren entfernten Repositories verbunden sein. Das ist ein deutlicher Ausdruck der verteilten Architektur.

Ein Open-Source-Beitrag kann beispielsweise so organisiert sein:

```
Lokales Repository
       │
       ├── origin   → eigener Fork auf GitHub
       │
       └── upstream → ursprüngliches Projekt auf GitHub
```

Die zugehörigen Remote-Namen sind frei wählbar. `origin` ist nur eine Konvention, die Git beim Klonen automatisch verwendet.

```
git remote -v
```

Eine mögliche Ausgabe:

```
origin    git@github.com:beispielkonto/mein-fork.git (fetch)
origin    git@github.com:beispielkonto/mein-fork.git (push)
upstream  git@github.com:organisation/originalprojekt.git (fetch)
upstream  git@github.com:organisation/originalprojekt.git (push)
```

Du kannst also Änderungen aus einem Repository abrufen und in ein anderes veröffentlichen. Git schreibt keine einzige zentrale Instanz vor.

Vergleich der Modelle

Aspekt	Zentrale Versionskontrolle	Verteilte Versionskontrolle mit Git
Maßgebliche Historie	Liegt primär auf einem zentralen Server	Liegt in vollständigen lokalen Repository-Kopien und optionalen Remotes
Commit	Häufig direkt auf dem Server	Zunächst lokal
Offline-Arbeit	Oft eingeschränkt	Für Commits, Branches, Logs und Diffs weitgehend möglich
Serverausfall	Kann den gesamten Workflow blockieren	Lokale Arbeit und Historienanalyse bleiben möglich
Branches	Je nach System eher aufwendig	Sehr leichtgewichtig und schnell
Zusammenarbeit	Stark auf zentralen Server ausgerichtet	Über Push, Fetch und frei wählbare Remotes
Datensicherung	Zentraler Server ist besonders kritisch	Mehrere vollständige Kopien können zusätzliche Redundanz schaffen
Zugriffssteuerung	Typischerweise zentral	Meist über Remotes, Plattformen und Serverregeln organisiert

Die Tabelle beschreibt Tendenzen. Moderne zentrale Systeme können Offline-Funktionen anbieten, und ein Git-Team kann seinen Workflow stark um ein einziges GitHub-Repository herum organisieren. Die grundlegende Architektur bleibt dennoch verschieden.

Verteilung bedeutet nicht automatisch Sicherheit

Es wäre falsch zu behaupten, dass Git automatisch ein Backup-Konzept ersetzt. Mehrere lokale Kopien können zwar helfen, doch sie sind nicht automatisch zuverlässig gesichert.

Beispiele für Risiken:

- Niemand außer einer Person hat die neuesten lokalen Commits.
- Ein Laptop wird beschädigt oder geht verloren.
- Ein lokaler Branch wird gelöscht.
- Ein Commit wird nie zu einem Remote gepusht.
- Ein zentraler GitHub-Account wird falsch konfiguriert oder kompromittiert.

Ein professionelles Team sorgt deshalb dafür, dass wichtige Arbeit regelmäßig auf ein geeignetes Remote übertragen wird und dass dieses Remote zusätzlich abgesichert ist.

> **Praxisregel:** Ein lokaler Commit schützt vor vielen Fehlern im Arbeitsverzeichnis. Ein gepushter Commit schützt zusätzlich vor dem Verlust eines einzelnen Computers. Ein getestetes Backup schützt vor weitergehenden Ausfällen.

Warum Git trotzdem oft „zentral“ wirkt

Ein GitHub-Workflow fühlt sich für viele Teams zentral an:

1. Ein Repository auf GitHub wird erstellt.
2. Alle Teammitglieder klonen dieses Repository.
3. Feature-Branches werden dorthin gepusht.
4. Pull Requests werden dort geprüft.
5. Der Hauptbranch wird dort geschützt.
6. CI-Prüfungen laufen dort.

Das ist keine Schwäche oder ein Widerspruch. Git erlaubt eine zentrale *Zusammenarbeitsstruktur*, ohne seine verteilte Natur aufzugeben.

Man kann es so unterscheiden:

Ebene	Zentral oder verteilt?
Git-Datenmodell	Verteilt
Lokale Git-Repositories	Eigenständig und vollständig
GitHub als Teamplattform	Häufig zentraler Koordinationspunkt
Teamregeln und Branch-Schutz	Zentral organisiert
Commits, Branches und Diffs auf dem Laptop	Lokal verfügbar

Diese Kombination ist einer der Gründe für Gits Erfolg: Teams erhalten lokale Geschwindigkeit und Flexibilität, können ihre Zusammenarbeit aber trotzdem klar über GitHub organisieren.

Ein konkretes Alltagsszenario

Stell dir vor, du arbeitest im Zug ohne Internetverbindung an einem PHP-Projekt.

Mit Git kannst du weiterhin:

```
git status
git diff
git switch feature/email-validation
git add src/EmailValidator.php
git commit -m "Fehlermeldung für ungültige Domain verbessern"
git log --oneline
```

Du kannst also Änderungen prüfen, Branches wechseln, Commits erstellen und die Historie untersuchen.

Sobald wieder eine Verbindung besteht, veröffentlichst du die bereits vorhandenen Commits:

```
git push origin feature/email-validation
```

Danach kann GitHub einen Pull Request anbieten oder du erstellst ihn selbst über die Website beziehungsweise PhpStorm.

In einem strikt zentralen System wäre der lokale Arbeitsschritt vergleichbar möglich, aber die vollständige Versionsverwaltung und insbesondere der Commit könnten stärker von der Erreichbarkeit des Servers abhängen.

Auswirkungen auf Branching und Experimente

Weil lokale Branches in Git sehr günstig erzeugt werden können, eignet sich Git gut für kurze, isolierte Experimente.

```
git switch -c experiment/neue-validierungsregel
```

Du kannst dort eine Idee umsetzen, testen und später entscheiden:

- Die Änderungen sind sinnvoll: Branch weiterentwickeln oder zusammenführen.
- Die Änderungen sind nicht sinnvoll: Branch löschen.
- Die Idee ist noch offen: Branch lokal behalten.

```
git switch main
git branch -D experiment/neue-validierungsregel
```

Solange du keinen Push ausführst, bleibt dieser Branch vollständig lokal. Er beeinflusst weder GitHub noch andere Teammitglieder.

Dieses Verhalten fördert kleine, risikoarme Experimente und klar abgegrenzte Änderungen.

Häufige Missverständnisse

„GitHub speichert meine Commits automatisch“

Nein. Ein lokaler Commit wird nicht automatisch zu GitHub übertragen.

Nach diesem Befehl:

```
git commit -m "Formularvalidierung verbessern"
```

existiert der Commit zunächst nur lokal. Erst ein Push veröffentlicht ihn im Remote:

```
git push
```

„Ein Git-Repository braucht immer GitHub“

Nein. Git funktioniert vollständig ohne GitHub.

Du kannst ein lokales Repository anlegen und verwenden:

```
git init
```

GitHub wird erst relevant, wenn du ein Remote für Zusammenarbeit, Sicherung, Veröffentlichung oder Automatisierung verwenden möchtest.

„Verteilt heißt, dass jede Person alles veröffentlichen darf“

Nein. Die technische Architektur und die Berechtigungen sind verschiedene Dinge.

Git erlaubt lokale Commits und lokale Branches ohne zentrale Freigabe. Ein GitHub-Repository kann aber genau festlegen:

- Wer pushen darf,
- welche Branches geschützt sind,
- ob Pull-Request-Reviews nötig sind,
- welche Prüfungen erfolgreich sein müssen,
- wer Releases erstellen darf.

„Lokale Commits sind unwichtig, weil nur GitHub zählt“

Lokale Commits sind sehr wichtig. Sie strukturieren deine Arbeit, ermöglichen sichere Experimente und bilden die Grundlage für jeden späteren Push. GitHub erhält keine magischen Änderungen, sondern die Commits, die du lokal erstellt hast.

Entscheidungshilfe: Welches Modell passt wann?

In der Praxis entscheidet man selten ausschließlich aufgrund der Architektur. Anforderungen an Teams, Infrastruktur, Sicherheit und bestehende Werkzeuge spielen ebenfalls eine Rolle.

Zentrale Versionskontrolle kann passend sein, wenn:

- ein bestehendes Unternehmen stark auf ein zentrales System ausgerichtet ist,
- sehr spezielle Sperr- und Dateiverwaltungsprozesse benötigt werden,
- große Binärdateien und exklusive Bearbeitung im Vordergrund stehen,
- ein vorhandenes System nicht kurzfristig migriert werden kann.

Verteilte Versionskontrolle mit Git ist besonders geeignet, wenn:

- Teams häufig mit Branches arbeiten,
- Entwicklerinnen und Entwickler auch offline produktiv sein sollen,
- kleine, lokale Commits erwünscht sind,
- Pull Requests und automatisierte Prüfungen etabliert werden sollen,
- Open-Source-Zusammenarbeit oder Forks relevant sind,
- moderne Plattformen wie GitHub, GitLab oder Bitbucket genutzt werden.

Für die meisten Softwareprojekte ist Git heute der verbreitete Standard. Das liegt nicht nur an GitHub, sondern vor allem an Gits leistungsfähigem lokalen Datenmodell, seinen günstigen Branches und seiner flexiblen Zusammenarbeit über Remotes.

Merkpunkte

- Ein **zentrales** Versionskontrollsystem speichert die maßgebliche Historie primär auf einem Server.
- Ein **verteiltes** System wie Git gibt jeder vollständigen lokalen Repository-Kopie die Projektgeschichte mit.
- In Git ist ein Commit zunächst **lokal**; `git push` veröffentlicht ihn in einem Remote.
- GitHub ist ein wichtiger zentraler Koordinationspunkt, macht Git aber nicht zu einem zentralen Versionskontrollsystem.
- Verteilte Repositories bieten Redundanz, ersetzen aber kein bewusstes Backup- und Push-Konzept.
- Die Fähigkeit, lokal zu committen, zu vergleichen, zu verzweigen und Historie zu lesen, ist ein zentraler Vorteil von Git.

Selbsttest

1. Worin besteht der wichtigste architektonische Unterschied zwischen zentraler und verteilter Versionskontrolle?
2. Warum kannst du mit Git ohne Internet einen Commit erstellen?
3. Was ist der Unterschied zwischen `git commit` und `git push`?
4. Weshalb bleibt Git ein verteiltes System, obwohl ein Team GitHub als gemeinsamen Mittelpunkt verwendet?
5. Warum ist ein lokaler Commit allein noch keine ausreichende Datensicherung?
6. Nenne eine Situation, in der lokale Branches besonders hilfreich sind.

Git als Snapshot-System

Die zentrale Idee: Git speichert Zustände, nicht nur Änderungen

Viele Versionskontrollsysteme werden zunächst als eine Folge von Änderungen verstanden:

> „In Version 2 wurden drei Zeilen ergänzt, in Version 3 eine Datei umbenannt.“

Git kann Änderungen selbstverständlich anzeigen und übertragen. Intern basiert sein Modell aber auf einer anderen, äußerst wichtigen Idee:

> **Ein Commit repräsentiert einen vollständigen Snapshot des Projektzustands zu einem bestimmten Zeitpunkt.**

Ein Snapshot ist wie eine Momentaufnahme des gesamten Repository-Inhalts. Er beschreibt, welche Dateien und Verzeichnisse zu diesem Zeitpunkt existieren und welche Inhalte sie haben.

Statt gedanklich nur zu fragen:

> „Welche Zeilen haben sich seit gestern geändert?“

kannst du bei Git fragen:

> „Wie sah das gesamte Projekt bei diesem Commit aus?“

Dieses Denkmodell erleichtert später das Verständnis von Branches, Merges, Rebase, Wiederherstellung und Konflikten erheblich.

Was ein Snapshot enthält

Ein Git-Commit verweist auf einen bestimmten Zustand des Projektverzeichnisses. Dieser Zustand umfasst insbesondere:

- Dateien und ihre Inhalte
- Verzeichnisse und ihre Struktur
- Dateinamen und Pfade
- Informationen über ausführbare Dateien
- den oder die Vorgänger-Commits
- Autor, Committer, Zeitstempel und Commit-Nachricht

Angenommen, ein kleines PHP-Projekt enthält zunächst diese Dateien:

```
projekt/
├── composer.json
├── src/
│   └── Greeting.php
└── tests/
    └── GreetingTest.php
```

Nach dem ersten Commit speichert Git einen Snapshot genau dieses Zustands.

Später ergänzt du eine README-Datei:

```
projekt/
├── README.md
├── composer.json
├── src/
│   └── Greeting.php
└── tests/
    └── GreetingTest.php
```

Der nächste Commit beschreibt diesen *neuen vollständigen Zustand*. Er sagt vereinfacht:

> „Zu diesem Zeitpunkt bestand das Projekt aus README.md, composer.json, src/Greeting.php und tests/GreetingTest.php — jeweils mit genau diesen Inhalten.“

Git speichert also nicht bloß die Aussage „README.md wurde hinzugefügt“. Diese Änderung lässt sich aus dem Vergleich zweier Snapshots ableiten.

Git zeigt Änderungen durch den Vergleich von Snapshots

Wenn du diesen Befehl ausführst:

```
git diff
```

zeigt Git dir Unterschiede zwischen Zuständen an. Das bedeutet jedoch nicht, dass Git einen Commit zwingend als klassische Liste einzelner Textänderungen speichert.

Git vergleicht beispielsweise:

- den Zustand im Arbeitsverzeichnis mit dem Index,
- den Index mit dem letzten Commit,
- zwei beliebige Commits,
- zwei Branches,
- oder zwei Versionstags.

Beispiel:

```
git diff HEAD~1 HEAD
```

Dieser Befehl vergleicht den vorletzten Commit mit dem aktuellen Commit. Git berechnet daraus, welche Dateien hinzugefügt, entfernt oder verändert wurden.

Die sichtbare Änderung könnte so aussehen:

```
+ # Mein PHP-Projekt
+
+ Ein kleines Beispielprojekt für Git.
```

Das ist eine *Darstellung des Unterschieds*. Das zugrunde liegende Modell bleibt: Git kennt zwei vollständige Zustände und vergleicht sie.

Ein Commit ist kein vollständiges Dateikopie-Archiv

Die Formulierung „vollständiger Snapshot“ kann einen falschen Eindruck erzeugen: Git kopiert nicht bei jedem Commit blind jede Datei erneut auf die Festplatte.

Das wäre bei größeren Projekten ineffizient.

Stattdessen speichert Git Inhalte objektbasiert:

- Unveränderte Dateiinhalte werden wiederverwendet.
- Nur neue oder geänderte Inhalte benötigen neue Objekte.
- Verzeichnisstrukturen werden ebenfalls als Objekte gespeichert.
- Commits verweisen auf diese unveränderlichen Objekte.

Wenn sich nur `README.md` ändert, muss Git nicht erneut eine Kopie von `src/Greeting.php` speichern. Der neue Snapshot verweist einfach wieder auf den bereits bekannten Inhalt dieser Datei.

Denkmodell: Jeder Commit beschreibt einen vollständigen Zustand, aber Git speichert gleiche Inhalte nur einmal.

Das verbindet Verständlichkeit mit hoher Effizienz.

Beispiel: Drei Zustände eines Projekts

Stell dir diese Entwicklung vor.

Commit A: Projekt anlegen

```
composer.json
src/Greeting.php
```

`src/Greeting.php`:

```
<?php

declare(strict_types=1);

final class Greeting
{
    public function message(): string
    {
        return 'Hallo';
    }
}
```

Commit B: Test ergänzen

```
composer.json
src/Greeting.php
tests/GreetingTest.php
```

Der Snapshot von Commit B enthält nun zusätzlich die Testdatei. Die Datei `src/Greeting.php` ist unverändert und kann daher auf denselben gespeicherten Inhalt wie in Commit A verweisen.

Commit C: Begrüßung ändern

```
composer.json
src/Greeting.php
tests/GreetingTest.php
```

Nun wird der Inhalt verändert:

```
public function message(): string
{
    return 'Hallo, Git!';
}
```

Commit C enthält wieder den vollständigen Projektzustand. Für `composer.json` und `tests/GreetingTest.php` kann Git bereits bekannte Inhalte verwenden. Für die geänderte Datei `src/Greeting.php` speichert Git einen neuen Inhalt.

Vereinfacht lässt sich die Historie so lesen:

```
Commit A
└── composer.json
└── src/Greeting.php [Version 1]

Commit B
└── composer.json
└── src/Greeting.php [Version 1]
└── tests/GreetingTest.php

Commit C
└── composer.json
└── src/Greeting.php [Version 2]
└── tests/GreetingTest.php
```

Jeder Commit ist ein klar definierter Projektzustand. Git erkennt dabei automatisch, welche Inhalte bereits existieren.

Warum dieses Modell so nützlich ist

Das Snapshot-Modell erklärt viele alltägliche Git-Funktionen besonders gut.

Alte Projektzustände ansehen

Du kannst jederzeit einen früheren Commit untersuchen:

```
git show <commit-hash>
```

Oder du wechselst gezielt zu einem historischen Zustand:

```
git switch --detach <commit-hash>
```

Du siehst dann das Projekt so, wie es zu diesem Commit gespeichert wurde.

> ⚠ In diesem Zustand arbeitest du meist in einem *Detached HEAD*. Änderungen sind möglich, aber nicht automatisch einem normalen Branch zugeordnet.

Einen Branch erstellen

Ein Branch ist im Kern ein beweglicher Name für einen Commit und damit für einen Snapshot der Historie.

Wenn du einen Feature-Branch erstellst, beginnt dieser beim aktuellen Snapshot. Danach entwickelt sich der Branch mit eigenen neuen Snapshots weiter.

```
A --- B --- C  main
           \
            D --- E  feature/login
```

- `C` ist der gemeinsame Ausgangszustand.
- `D` und `E` sind neue Snapshots auf dem Feature-Branch.
- `main` verweist weiterhin auf `C`, bis dort weitere Commits entstehen.

Einen Merge durchführen

Beim Mergen versucht Git, verschiedene Entwicklungsstände zu einem neuen gemeinsamen Snapshot zusammenzuführen.

Git betrachtet dafür typischerweise:

1. den gemeinsamen Ausgangszustand,
2. den Zustand des aktuellen Branches,
3. den Zustand des einzufügenden Branches.

Das Ergebnis ist ein neuer Snapshot, häufig mit einem Merge-Commit.

Änderungen zurückholen

Wenn ein früherer Commit einen funktionierenden Zustand enthält, kannst du daraus gezielt Dateien oder Inhalte wiederherstellen. Du musst nicht mühsam manuell rekonstruieren, welche Änderungen irgendwann vorgenommen wurden.

Beispiel:

```
git restore --source=<commit-hash> -- src/Greeting.php
```

Damit übernimmst du die Version der Datei aus einem bestimmten historischen Snapshot in dein Arbeitsverzeichnis.

Snapshots und die drei Bereiche von Git

Das Snapshot-Modell hängt direkt mit den drei zentralen Bereichen von Git zusammen:

Bereich	Bedeutung im Snapshot-Modell
Arbeitsverzeichnis	Der aktuell ausgecheckte Projektzustand auf deiner Festplatte, den du bearbeitest
Index oder *Staging Area*	Der vorbereitete Zustand für den nächsten Snapshot
Lokales Repository	Die dauerhaft gespeicherten Snapshots und ihre Historie

Ein typischer Ablauf sieht so aus:

```
Arbeitsverzeichnis
      │
      │ git add
      ▼
Index
      │
      │ git commit
      ▼
Repository mit neuem Snapshot
```

Wenn du eine Datei änderst, betrifft das zunächst nur dein Arbeitsverzeichnis. Mit `git add` legst du fest, welche Version dieser Datei in den nächsten Snapshot aufgenommen wird. Erst `git commit` erzeugt den neuen gespeicherten Zustand.

Das ist ein entscheidender Unterschied zu einem bloßen automatischen Speichern.

Der Index: Entwurf des nächsten Snapshots

Die Staging Area ist besonders sinnvoll, wenn du mehrere Änderungen gleichzeitig bearbeitest, aber nicht alles gemeinsam committen möchtest.

Angenommen, du hast parallel:

- einen Fehler in `src/Greeting.php` behoben,
- eine neue Funktion begonnen,
- Tippfehler in `README.md` korrigiert.

Mit dem Index kannst du gezielt einen sauberen Snapshot vorbereiten:

```
git add src/Greeting.php
git commit -m "fix: Begrüßung bei leerem Namen korrigieren"
```

Die noch unvollständige Funktion und die Dokumentationsänderung bleiben zunächst außerhalb dieses Commits.

Der Commit beschreibt dadurch einen klaren, überprüfbaren Zustand:

> „Der Fehler ist behoben, ohne andere unfertige Arbeiten einzuschließen.“

Dateien werden nicht als „umbenannt“ gespeichert

Eine häufig überraschende Konsequenz des Snapshot-Modells betrifft Umbenennungen.

Wenn du ausführst:

```
git mv src/Greeting.php src/Greeter.php
git commit -m "refactor: Greeting in Greeter umbenennen"
```

speichert Git grundsätzlich einen neuen Snapshot mit:

- einer nicht mehr vorhandenen Datei unter `src/Greeting.php`,
- einer vorhandenen Datei unter `src/Greeter.php`.

Git kann beim Vergleich erkennen, dass die Inhalte sehr ähnlich oder identisch sind. Dann zeigt es die Änderung als Umbenennung an.

```
git log --follow -- src/Greeter.php
```

Die Umbenennung ist also häufig eine **Erkennung beim Vergleich**, keine zwingend separat gespeicherte Operation.

Das gilt ähnlich für verschobene Dateien und für viele Arten von Dateiumstrukturierungen.

Snapshot-Modell und Speicherbedarf

Git speichert Inhalte anhand ihres Hashwerts. Gleicher Inhalt erzeugt denselben identifizierbaren Objektinhalt und kann wiederverwendet werden.

Praktisch bedeutet das:

- Eine unveränderte Datei wird nicht bei jedem Commit vollständig dupliziert.
- Identische Inhalte können mehrfach referenziert werden.
- Git komprimiert und packt ältere Objekte zusätzlich effizient.
- Ein Repository mit vielen Commits ist nicht automatisch so groß wie viele vollständige ZIP-Archive des Projekts.

Trotzdem solltest du große Binärdateien wie Videos, Datenbank-Dumps oder erzeugte Build-Artefakte nicht unbedacht versionieren. Schon kleine Änderungen in solchen Dateien können zu großen zusätzlichen Objekten führen.

Für geeignete große Binärdateien gibt es später Git LFS. Generierte Dateien, Caches, Zugangsdaten und lokale IDE-Dateien gehören häufig in `.gitignore`.

Typische Missverständnisse vermeiden

„Git speichert nur Unterschiede“

Nicht als grundlegendes Denkmodell. Git speichert **Zustände als verknüpfte Objekte** und kann Unterschiede zwischen diesen Zuständen berechnen.

„Jeder Commit kopiert das gesamte Projekt“

Ein Commit beschreibt den vollständigen Zustand. Identische Inhalte werden aber effizient wiederverwendet.

„Ein Commit sichert automatisch alles“

Nein. Ein Commit enthält nur das, was du zuvor in die Staging Area aufgenommen hast.

Prüfe daher vor jedem Commit:

```
git status
git diff --staged
```

„Git erkennt jede Umbenennung dauerhaft“

Git erkennt Umbenennungen meist beim Vergleich ähnlicher Inhalte. Bei umfangreichen Änderungen kann diese Erkennung ausbleiben oder anders ausfallen.

„Ein Snapshot ist ein Backup ohne Grenzen“

Git ist sehr hilfreich zur Wiederherstellung versionierter Dateien. Nicht versionierte Dateien, ignorierte Dateien und nie committed Änderungen sind jedoch nicht automatisch geschützt. Ein echtes Backup bleibt wichtig.

Praktische Kontrolle mit Git

Mit diesen Befehlen untersuchst du Snapshots und ihre Unterschiede im Alltag:

```
# Aktuellen Zustand und vorgemerkte Änderungen prüfen
git status

# Nicht vorgemerkte Änderungen ansehen
git diff

# Vorgemerkte Änderungen zum nächsten Snapshot ansehen
git diff --staged

# Aktuellen Commit mit seinem Vorgänger vergleichen
git diff HEAD~1 HEAD

# Inhalt eines bestimmten Commits anzeigen
git show <commit-hash>

# Historische Snapshots in kompakter Form auflisten
git log --oneline --decorate --graph
```

In PhpStorm findest du dieselbe Idee im **Git Log**:

- Jeder Eintrag steht für einen Commit und damit für einen gespeicherten Projektzustand.
- Ein Klick auf einen Commit zeigt die aus dem Vergleich abgeleiteten Änderungen.
- Über **Compare with Current** oder ähnliche Vergleichsfunktionen vergleichst du historische Snapshots mit deinem aktuellen Stand.
- Die Diff-Ansicht zeigt Unterschiede; sie ersetzt aber nicht das grundlegende Verständnis, dass Git Zustände gegenüberstellt.

Merksätze

- **Ein Commit ist ein Snapshot eines vorbereiteten Projektzustands.**
- **Git zeigt Diffs, indem es Snapshots miteinander vergleicht.**
- **Unveränderte Inhalte werden effizient wiederverwendet.**
- **Der Index ist der Entwurf für den nächsten Snapshot.**
- **Branches zeigen auf Commits und damit auf bestimmte Zustände der Historie.**

- **Saubere Commits erzeugen verständliche, überprüfbare Zustände.**

Wenn du Git als Sammlung miteinander verbundener Projektzustände verstehst, wirken spätere Befehle deutlich weniger wie Magie: `branch`, `merge`, `rebase`, `restore` und `revert` werden dann zu unterschiedlichen Wegen, Snapshots zu erstellen, zu vergleichen oder wiederherzustellen.

Repository, Commit und Branch

Drei Grundbegriffe für die tägliche Git-Arbeit

Fast jede Git-Aktion dreht sich um drei Begriffe:

- **Repository**: der verwaltete Projektbestand inklusive Geschichte
- **Commit**: ein dauerhaft gespeicherter, beschrifteter Projektzustand
- **Branch**: eine bewegliche Entwicklungslinie innerhalb eines Repositorys

Wer diese Begriffe sauber voneinander trennt, versteht viele Git-Befehle nicht mehr als einzelne Kommandos, sondern als nachvollziehbare Zustandsänderungen.

Das Repository: Projekt und Geschichte an einem Ort

Ein **Repository** ist der Bereich, den Git verwaltet. Es enthält nicht nur die aktuellen Dateien eines Projekts, sondern auch deren vollständige nachvollziehbare Entwicklung.

In einem Repository speichert Git unter anderem:

- Dateien des Projekts,
- frühere Versionen dieser Dateien,
- Commits mit Autor, Zeitpunkt und Nachricht,
- Branches und Tags,
- Einstellungen für dieses konkrete Projekt,
- Informationen über verbundene Server, etwa GitHub.

Ein lokales Git-Repository entsteht beispielsweise mit:

```
git init
```

Danach legt Git im Projektordner einen versteckten Unterordner namens `.git` an. Dieser Ordner ist das technische Herz des Repositorys.

```
mein-projekt/
├── .git/
├── src/
├── tests/
├── README.md
└── composer.json
```

> **Wichtig:** Der Ordner `.git` enthält die Git-Datenbank mit der Historie. Wird er gelöscht, bleiben zwar die Projektdateien erhalten, aber Git „vergisst“ Commits, Branches, Einstellungen und die gesamte Versionsgeschichte.

Lokale und entfernte Repositories

Ein Repository kann ausschließlich auf dem eigenen Computer liegen. Das ist ein **lokales Repository**.

Später kann es mit einem Server verbunden werden, zum Beispiel mit GitHub. Das dortige Repository wird als **entferntes Repository** oder *Remote Repository* bezeichnet.

```
Lokaler Computer                    GitHub
────────────────                    ──────

Arbeitskopie                        Entferntes Repository
Git-Historie        ←── Sync ──→    Gemeinsame Git-Historie
Eigene Branches                     Veröffentlichte Branches
```

Git funktioniert aber bereits vollständig ohne GitHub und ohne Internetverbindung. GitHub erweitert Git um Zusammenarbeit, Hosting, Reviews, Issues und Automatisierung. Es ersetzt Git nicht.

Der Commit: ein nachvollziehbarer Projektzustand

Ein **Commit** ist ein gespeicherter Zustand des Projekts zu einem bestimmten Zeitpunkt. Er hält fest, welche Dateien und Inhalte zu diesem Zeitpunkt zum Projekt gehören.

Ein Commit ist daher **kein bloßer Speichern-Knopf**. Er ist eine bewusste Aussage über die Entwicklung des Projekts.

Beispiele für sinnvolle Commit-Aussagen:

- „Registrierung mit E-Mail-Validierung hinzugefügt“
- „Fehlerhafte Berechnung der Mehrwertsteuer korrigiert“
- „API-Dokumentation für Benutzerendpunkt ergänzt“
- „PHPUnit-Tests für Passwort-Reset hinzugefügt“

Weniger hilfreich sind Nachrichten wie:

- „Änderungen“
- „Update“
- „Fix“
- „WIP“
- „asdf“

Eine gute Commit-Nachricht beantwortet knapp die Frage:

> *Was wurde in diesem Schritt fachlich oder technisch verändert?*

Was Git in einem Commit speichert

Ein Commit enthält mehr als den Inhalt der Dateien. Vereinfacht gehören dazu:

- ein eindeutiger Bezeichner, meist als Hash dargestellt,
- der gespeicherte Projektzustand,
- eine Commit-Nachricht,
- Autor und Zeitpunkt,
- Committer und Zeitpunkt der tatsächlichen Speicherung,
- ein oder mehrere Vorgänger-Commits.

Ein Commit könnte in einer Kurzansicht so aussehen:

```
Commit: 8f3c1a2
Autor:  Maria Beispiel
Datum:  2025-03-08

Validierung für Registrierungsformular hinzugefügt
```

Der lange technische Bezeichner ist ein Hash, zum Beispiel:

```
8f3c1a2d7e4b0c4af95e3b8d1e6f9a2c7b4d5e6f
```

Im Alltag genügt meist ein eindeutiger Anfang dieses Hashes, etwa `8f3c1a2`.

Commits bilden eine Geschichte

Jeder neue Commit verweist normalerweise auf seinen Vorgänger. Dadurch entsteht eine chronologische, technisch präzise Projektgeschichte.

```
A ── B ── C
```

Dabei könnte gelten:

```
A  Projektgrundstruktur erstellt
B  Benutzeranmeldung implementiert
C  Anmeldung mit Tests abgesichert
```

Der aktuelle Projektstand ist dann Commit `C`. Git kann aber jederzeit die Zustände von `A` oder `B` untersuchen, vergleichen oder bei Bedarf wiederherstellen.

Commits sind lokale Sicherungspunkte

Ein Commit wird zunächst nur im lokalen Repository gespeichert. Er ist also noch nicht automatisch auf GitHub sichtbar und auch nicht für Kolleginnen und Kollegen verfügbar.

```
Dateien ändern
      ↓
Commit lokal erstellen
      ↓
Commit zu GitHub übertragen
```

Das Übertragen zu einem Remote erfolgt später mit `git push`.

Diese Trennung ist bewusst: Du kannst lokal sauber arbeiten, Commits vorbereiten, korrigieren und testen, bevor du Änderungen mit anderen teilst.

Ein Commit ist kein Dateiarchiv

Ein verbreitetes Denkmodell lautet: Git speichere bei jedem Commit einfach eine vollständige ZIP-Datei des Projekts. Das ist als erste Vorstellung nützlich, aber technisch nicht ganz richtig.

Git behandelt jeden Commit als vollständigen **Snapshot** des Projektzustands. Intern speichert Git identische Inhalte jedoch effizient nur einmal und verknüpft sie wieder.

Das entscheidende mentale Modell lautet:

> Ein Commit beschreibt, wie das gesamte Projekt zu einem bestimmten Zeitpunkt aussieht.

Git speichert also nicht primär die Anweisung „Zeile 12 wurde geändert“, sondern einen neuen Projektzustand, der auf bereits bekannten Inhalten aufbaut.

Das erklärt später unter anderem, warum Git:

- Änderungen sehr zuverlässig vergleichen kann,
- Branches schnell erstellt,
- alte Zustände einfach wiederherstellen kann,
- mehrere Entwicklungswege effizient verwaltet.

Der Branch: eine Entwicklungslinie

Ein **Branch** ist eine benannte Entwicklungslinie. Er zeigt auf einen bestimmten Commit und bewegt sich weiter, wenn auf diesem Branch neue Commits entstehen.

Ein frisch initialisiertes Repository besitzt meist einen Standard-Branch namens `main`.

```
main
  ↓
A ── B ── C
```

Der Name `main` zeigt hier auf den neuesten Commit `C`.

Wenn ein weiterer Commit entsteht, verschiebt Git den Branch-Zeiger automatisch:

```
A ── B ── C ── D
                ↑
              main
```

Der Branch enthält dabei nicht selbst alle Dateien und auch keine Kopie der Commits. Er ist vielmehr ein **beweglicher Verweis** auf den neuesten Commit einer Entwicklungslinie.

> **Merksatz:** Ein Branch ist kein Projektordner und keine vollständige Projektkopie. Er ist ein Name, der auf einen Commit zeigt.

Warum Branches wichtig sind

Branches ermöglichen es, mehrere Arbeiten parallel durchzuführen, ohne den stabilen Hauptstand direkt zu gefährden.

Angenommen, die produktive Anwendung befindet sich auf `main`:

```
A ── B ── C
          ↑
        main
```

Nun soll eine neue Passwort-Reset-Funktion entwickelt werden. Dafür wird ein Feature-Branch angelegt:

```
A ── B ── C
          ↑
        main
          ↑
   feature/password-reset
```

Zu Beginn zeigen beide Branches auf denselben Commit `C`. Danach entstehen die neuen Commits nur auf dem Feature-Branch:

```
A ── B ── C ── D ── E
           ↑         ↑
         main   feature/password-reset
```

Die neue Funktion kann entwickelt, getestet und geprüft werden, ohne den Stand auf `main` zu verändern.

Erst wenn die Arbeit fertig ist, wird der Feature-Branch wieder mit `main` zusammengeführt. Dieser Vorgang heißt **Merge**.

```
A ── B ── C ──────────── F
          │             ↑
          D ── E      main
              ↑
      feature/password-reset
```

Die genaue Form der Historie hängt von der gewählten Merge-Strategie ab. Das Prinzip bleibt jedoch gleich: Branches erlauben isolierte Entwicklung und kontrollierte Integration.

HEAD: Wo arbeite ich gerade?

Git muss wissen, auf welchem Branch du gerade arbeitest. Dafür verwendet Git die Referenz **HEAD**.

HEAD zeigt normalerweise auf den aktuell ausgecheckten Branch:

```
A ── B ── C
           ↑
         main
           ↑
          HEAD
```

Wenn du den Branch wechselst, folgt HEAD diesem Branch:

```
A ── B ── C ── D
           ↑         ↑
         main   feature/login
                     ↑
                   HEAD
```

Neue Commits werden dann auf `feature/login` erstellt. Der Branch `main` bleibt unverändert auf Commit `C`.

In der Kommandozeile zeigt dieser Befehl die vorhandenen lokalen Branches an:

```
git branch
```

Der Stern markiert den aktuell aktiven Branch:

```
* main
  feature/login
  feature/password-reset
```

Mit modernen Git-Versionen wechselst du einen Branch meist so:

```
git switch feature/login
```

Die Arbeit mit Branches wird in einem späteren Kapitel ausführlich behandelt. Für den Einstieg genügt dieses Modell:

- **HEAD** markiert deine aktuelle Arbeitsposition.
- Der aktive **Branch** bestimmt, wohin neue Commits geschrieben werden.
- Ein neuer Commit bewegt den aktiven Branch weiter.

Repository, Commit und Branch im Zusammenhang

Die drei Begriffe gehören zusammen, erfüllen aber unterschiedliche Aufgaben:

Begriff	Aufgabe	Beispiel
Repository	Verwaltet Projektdateien, Historie und Git-Metadaten	Das gesamte PHP-Projekt mit allen Commits
Commit	Speichert einen nachvollziehbaren Projektzustand	„Login-Validierung ergänzt“
Branch	Kennzeichnet eine Entwicklungslinie und zeigt auf ihren neuesten Commit	`main`, `feature/login`

Eine vereinfachte Darstellung:

```
Repository
│
├── Branch main
│   └── A ── B ── C
│
└── Branch feature/login
    └── A ── B ── C ── D ── E
```

Die Commits `A`, `B` und `C` werden von beiden Branches geteilt. Erst ab `D` entwickelt sich der Feature-Branch unabhängig weiter.

Praxisbeispiel: Entwicklung einer kleinen Funktion

Stell dir ein PHP-Projekt für eine Aufgabenverwaltung vor. Das Repository enthält zunächst drei Commits:

```
A  Projekt mit Composer eingerichtet
B  Aufgabenliste implementiert
C  PHPUnit-Grundkonfiguration ergänzt
```

Der Standard-Branch `main` zeigt auf den letzten Commit:

```
A ── B ── C
          ↑
        main
```

Nun soll eine Funktion zum Markieren erledigter Aufgaben entstehen.

1. Feature-Branch erstellen

```
git switch -c feature/complete-task
```

Git erstellt den Branch und wechselt sofort dorthin:

```
A ── B ── C
           ↑
         main
           ↑
feature/complete-task
           ↑
          HEAD
```

2. Funktion implementieren und committen

Nach der Implementierung und passenden Tests wird ein Commit erstellt:

```
git add src/Task.php tests/TaskTest.php
git commit -m "Erledigte Aufgaben markieren"
```

Die Historie sieht danach so aus:

```
A ── B ── C ── D
           ↑      ↑
         main  feature/complete-task
```

3. Weiterarbeiten, ohne `main` zu verändern

Ein weiterer Commit ergänzt die Darstellung im Benutzerinterface:

```
git add templates/tasks.php
git commit -m "Erledigte Aufgaben visuell hervorheben"
```

```
A ── B ── C ── D ── E
           ↑              ↑
         main  feature/complete-task
```

Währenddessen bleibt `main` auf dem getesteten Stand `C`. Andere Teammitglieder können dort arbeiten, ohne unvollständige Änderungen an der neuen Funktion zu übernehmen.

Häufige Missverständnisse

„Ein Branch ist eine Kopie des Projekts“

Nicht im üblichen Sinn. Git kopiert nicht bei jedem neuen Branch das gesamte Projektverzeichnis. Ein Branch ist hauptsächlich ein zusätzlicher Zeiger auf einen Commit. Deshalb sind Branches in Git schnell und günstig.

„Ein Commit wird automatisch auf GitHub gespeichert“

Nein. Ein Commit entsteht zunächst lokal. Erst ein Push überträgt ihn an ein entferntes Repository.

```
git push
```

„Ich brauche für jede Änderung einen neuen Branch“

Nicht zwingend. Für kleine persönliche Experimente kann Arbeit direkt auf `main` sinnvoll sein. In Teamprojekten und bei größeren Änderungen sind separate Feature-Branches jedoch meist der sichere Standard.

„Ein Branch muss nach dem Zusammenführen bleiben“

Nein. Ist eine Funktion integriert, wird ihr Feature-Branch häufig gelöscht. Die enthaltenen Commits bleiben erhalten, sofern sie in die Zielhistorie übernommen wurden.

„Ein Commit muss möglichst viele Änderungen enthalten“

Das Gegenteil ist meist besser: Ein Commit sollte eine **abgeschlossene, logisch zusammengehörige Änderung** darstellen. Kleine, verständliche Commits erleichtern Reviews, Fehlersuche und spätere Korrekturen.

Praktische Leitlinien

Für den Einstieg helfen diese Regeln:

1. **Ein Repository pro klar abgegrenztem Projekt**
   Ein Repository sollte eine zusammenhängende Anwendung, Bibliothek oder Konfiguration verwalten.
2. **Commits klein und verständlich halten**
   Ein Commit sollte idealerweise eine Aufgabe, Fehlerbehebung oder fachliche Änderung beschreiben.
3. **Aussagekräftige Commit-Nachrichten schreiben**
   Die Nachricht soll auch Wochen später ohne erneutes Lesen des Codes verständlich sein.
4. **Für neue Funktionen eigene Branches verwenden**
   Namen wie `feature/login`, `fix/tax-calculation` oder `docs/setup-guide` machen den Zweck sichtbar.
5. **Den stabilen Hauptbranch schützen**
   `main` sollte in Teamprojekten möglichst nur getestete und überprüfte Änderungen enthalten.
6. **Vor einer Änderung den aktuellen Branch prüfen**
   Viele Fehler entstehen, weil versehentlich auf dem falschen Branch gearbeitet wird.

```
git status
```

Git zeigt dabei unter anderem an, auf welchem Branch du dich befindest.

Zusammenfassung

Ein Git-Repository ist das verwaltete Projekt mitsamt seiner vollständigen Historie. Ein Commit speichert einen bewusst gewählten, beschrifteten Projektzustand. Ein Branch benennt eine Entwicklungslinie und zeigt auf ihren aktuellsten Commit.

Das zentrale Bild lautet:

```
Repository
├── enthält die gesamte Historie
```

```
├── besteht aus vielen Commits
└── organisiert parallele Entwicklung über Branches
```

Oder noch kürzer:

> **Das Repository ist der Speicherort, der Commit ist der Zustand, der Branch ist die Entwicklungslinie.**

Mit diesem Modell lassen sich die nächsten Git-Schritte — insbesondere Staging, Commit-Erstellung, Branch-Wechsel und Zusammenführen — systematisch verstehen.

Git und GitHub unterscheiden

Zwei Begriffe, zwei Aufgaben

Git und **GitHub** werden oft in einem Atemzug genannt, sind aber grundverschiedene Dinge:

- **Git** ist ein Versionskontrollsystem. Es läuft primär lokal auf deinem Computer und verwaltet die Geschichte deiner Dateien.
- **GitHub** ist eine Online-Plattform, die Git-Repositories hostet und Zusammenarbeit, Reviews, Automatisierung und Projektorganisation ergänzt.

Die Kurzform lautet:

> **Git verwaltet Versionen. GitHub verbindet Menschen und Repositories.**

Du kannst Git vollständig ohne GitHub einsetzen. Umgekehrt basiert ein GitHub-Repository in der Regel auf Git.

Git: das lokale Versionskontrollsystem

Git ist freie Software, die von Linus Torvalds entwickelt wurde. Es speichert die Entwicklung eines Projekts als Folge von **Commits**. Jeder Commit beschreibt einen nachvollziehbaren Stand des Projekts.

Git arbeitet in einem lokalen Repository auf deinem Rechner. Sobald du ein Repository mit `git init` anlegst oder mit `git clone` kopierst, enthält dein Projekt einen versteckten Ordner namens `.git`. Darin liegen unter anderem:

- die Commit-Historie,

- Branches und Tags,
- die Staging Area,
- lokale Konfigurationen,
- Referenzen auf entfernte Repositories,
- Informationen zur Wiederherstellung über Reflogs.

Ein Repository kann deshalb auch ohne Internetverbindung sinnvoll genutzt werden.

Typische Git-Aufgaben

Mit Git erledigst du beispielsweise diese Arbeitsschritte:

```
git status
git add src/Calculator.php
git commit -m "Füge Addition hinzu"
git switch -c feature/subtraction
git merge feature/subtraction
git log --oneline
```

Diese Befehle funktionieren **lokal**. Sie benötigen weder ein GitHub-Konto noch eine Netzwerkverbindung.

Git ist kein Server und keine Website

Git selbst stellt keine zentrale Website, keine Benutzerverwaltung und keine Pull-Request-Oberfläche bereit. Es kennt zwar entfernte Repositories, sogenannte **Remotes**, aber Git schreibt nicht vor, wo diese liegen.

Ein Remote kann zum Beispiel sein:

- ein Repository auf GitHub,
- ein Repository auf GitLab oder Bitbucket,
- ein Git-Server im Unternehmen,
- ein Repository auf einem eigenen Server,
- ein Verzeichnis in einem lokalen Netzwerk,
- theoretisch sogar ein anderer Ordner auf demselben Computer.

GitHub: eine Plattform rund um Git

GitHub ist ein kommerzieller Cloud-Dienst von GitHub, Inc., das zu Microsoft gehört. Die Plattform speichert Git-Repositories auf Servern und ergänzt Git um Funktionen für Zusammenarbeit und Projektmanagement.

Ein Repository auf GitHub ist also ein **entferntes Git-Repository**, das über das Internet erreichbar ist. Üblicherweise erhält es den Remote-Namen `origin`.

```
git remote -v
```

Eine typische Ausgabe sieht so aus:

```
origin  git@github.com:beispielkonto/mein-projekt.git (fetch)
origin  git@github.com:beispielkonto/mein-projekt.git (push)
```

Die URL zeigt: Das lokale Repository ist mit einem Repository auf GitHub verbunden.

Funktionen, die GitHub zusätzlich liefert

GitHub erweitert reines Git um eine umfangreiche Web-Plattform:

- **Repository-Hosting** für öffentliche und private Projekte
- **Pull Requests** für vorgeschlagene Änderungen und Code Reviews
- **Issues** für Fehler, Aufgaben und Feature-Wünsche
- **GitHub Actions** für Tests, Builds, Deployments und Releases
- **Projects** für Planung und Projektübersichten
- **Discussions** für langfristige technische oder Community-Diskussionen
- **Wiki** für ergänzende Dokumentation
- **Teams, Rollen und Berechtigungen** für Organisationen
- **Branch-Schutzregeln** und verpflichtende Prüfungen
- **Security-Funktionen**, etwa Secret Scanning, Dependabot und Code Scanning
- **Releases** mit Versionshinweisen und Dateien zum Herunterladen

Keine dieser Funktionen ist Bestandteil von Git selbst.

Der zentrale Vergleich

Aspekt	Git	GitHub
Art	Versionskontrollsystem	Hosting- und Kollaborationsplattform
Läuft primär	Lokal auf deinem Rechner	Auf Servern und im Webbrowser

Aspekt	Git	GitHub
Internet erforderlich	Nein, für lokale Arbeit nicht	Ja, normalerweise für die Nutzung
Kernaufgabe	Dateien, Commits, Branches und Historie verwalten	Repositories teilen und Zusammenarbeit organisieren
Wird installiert	Ja, beispielsweise über Git for Windows	Nein, Nutzung per Browser, API oder Client
Kennt Pull Requests	Nein	Ja
Kennt Issues und Projektboards	Nein	Ja
Kann allein verwendet werden	Ja	Nein, Git ist die technische Grundlage
Beispiele für Alternativen	Mercurial, Subversion	GitLab, Bitbucket, Azure Repos, Gitea

---

Ein praktisches Bild: Notizbuch und gemeinsamer Arbeitsraum

Eine hilfreiche Analogie ist ein persönliches Notizbuch:

- **Git** ist dein Notizbuch mit einer lückenlosen Versionsgeschichte. Du kannst Seiten ergänzen, Fehler korrigieren, Kapitel verzweigen und frühere Stände nachschlagen.
- **GitHub** ist ein gemeinsamer Arbeitsraum mit Kopien dieser Notizbücher. Dort können andere Personen Änderungen ansehen, kommentieren, prüfen, zusammenführen und automatisiert testen lassen.

Das Notizbuch funktioniert auch dann, wenn du allein arbeitest oder keine Internetverbindung hast. Der gemeinsame Arbeitsraum wird wichtig, sobald du dein Projekt sichern, veröffentlichen oder mit anderen entwickeln möchtest.

Diese Analogie hat allerdings eine Grenze: GitHub speichert nicht nur eine einfache Sicherungskopie. Es ist ein vollwertiges, ebenfalls Git-basiertes Repository mit zusätzlicher Plattformfunktionalität.

Ein typischer Ablauf mit Git und GitHub

In der Praxis greifen beide Werkzeuge ineinander.

1. Du erstellst oder klonst ein **lokales Git-Repository**.
2. Du änderst Dateien in deinem Arbeitsverzeichnis.
3. Du nimmst ausgewählte Änderungen mit `git add` in die Staging Area auf.
4. Du erzeugst lokal einen Commit mit `git commit`.
5. Du überträgst den Commit mit `git push` zu GitHub.
6. Auf GitHub erstellst du bei Bedarf einen Pull Request.
7. Teammitglieder prüfen die Änderungen, hinterlassen Kommentare und geben sie frei.
8. GitHub führt den Branch nach den Teamregeln zusammen oder löst automatisierte Prüfungen aus.
9. Du holst die neue gemeinsame Historie mit `git fetch` oder `git pull` zurück auf deinen Computer.

Wichtig ist die Reihenfolge: **Ein Commit entsteht zunächst lokal in Git.** Erst ein Push macht ihn für das Remote-Repository auf GitHub verfügbar.

Lokale und entfernte Historie unterscheiden

Ein häufiger Anfängerfehler besteht darin, die lokale und die entfernte Historie als identisch zu betrachten. Tatsächlich können sie voneinander abweichen.

Angenommen, dein aktueller lokaler Branch `main` enthält einen Commit, den du noch nicht veröffentlicht hast:

```
Lokal:    A — B — C
GitHub:   A — B
```

Commit `C` existiert bereits vollständig in deinem lokalen Git-Repository. Erst mit diesem Befehl wird er nach GitHub übertragen:

```
git push origin main
```

Danach entspricht die Historie wieder einander:

```
Lokal:    A — B — C
GitHub:   A — B — C
```

Andersherum können Teammitglieder Commits auf GitHub veröffentlicht haben, die du noch nicht abgerufen hast. Dann hilft:

```
git fetch origin
```

GitHub ist daher nicht „Git in der Cloud“, sondern ein **entferntes Repository mit einer Kollaborationsplattform darum herum**.

GitHub ist nicht die einzige Git-Plattform

Git ist ein offener Standard und nicht an GitHub gebunden. Ein lokales Git-Repository kann mit unterschiedlichen Anbietern verbunden werden.

Bekannte Alternativen sind:

Plattform	Typischer Einsatz
GitLab	Cloud-Angebot oder selbst betriebene Unternehmensinstanz
Bitbucket	Häufig in Teams mit Atlassian-Werkzeugen wie Jira
Azure Repos	Integration in Microsoft Azure DevOps
Gitea	Schlanke, selbst hostbare Open-Source-Lösung
Forgejo	Community-orientierte, selbst hostbare Git-Plattform

Die grundlegenden Git-Befehle bleiben dabei gleich:

```
git clone
git fetch
git pull
git push
```

Was sich ändert, sind vor allem die Weboberfläche, Rechteverwaltung, Automatisierungen und Begriffe der Plattform. GitHub verwendet beispielsweise den Begriff **Pull Request**. Bei GitLab heißt eine vergleichbare Funktion **Merge Request**.

GitHub ist auch nicht GitHub Desktop

Zusätzlich kann der Name **GitHub Desktop** verwirren.

GitHub Desktop ist eine grafische Anwendung für Git und GitHub. Sie vereinfacht häufige Git-Aktionen, etwa Commits, Branch-Wechsel und Pushes. Sie ist jedoch weder Git selbst noch die GitHub-Webplattform.

Die drei Begriffe lassen sich klar einordnen:

Begriff	Bedeutung
Git	Technisches Versionskontrollsystem
GitHub	Web-Plattform für Git-Hosting und Zusammenarbeit
GitHub Desktop	Grafische Anwendung zur Bedienung von Git und GitHub

Auch PhpStorm ist in diesem Sinn ein Client: Die IDE ruft im Hintergrund Git-Funktionen auf, zeigt ihre Ergebnisse grafisch an und kann zusätzlich mit GitHub verbunden werden.

Was PhpStorm dabei übernimmt

PhpStorm kann sowohl mit lokalem Git als auch mit GitHub arbeiten.

Lokale Git-Funktionen in PhpStorm

Ohne GitHub-Anmeldung kannst du in PhpStorm bereits:

- ein Repository initialisieren,
- Änderungen vergleichen,
- Dateien zur Staging Area hinzufügen,
- Commits erstellen,
- Branches anlegen und wechseln,
- Merges und Rebases durchführen,
- die lokale Historie untersuchen.

Dafür muss Git auf Windows installiert und in PhpStorm konfiguriert sein.

GitHub-Funktionen in PhpStorm

Nach der Verbindung mit einem GitHub-Konto kann PhpStorm zusätzlich:

- Projekte auf GitHub veröffentlichen,
- Repositories klonen,
- Pull Requests erstellen und prüfen,
- Pull-Request-Kommentare lesen und beantworten,
- Issues anzeigen,
- Remotes komfortabel verwalten.

Die IDE ersetzt dabei nicht das Verständnis der zugrunde liegenden Git-Abläufe. Sie bietet lediglich eine andere Bedienoberfläche für viele davon.

Häufige Missverständnisse

„Ich habe GitHub installiert.“

GitHub ist hauptsächlich ein Webdienst und wird nicht wie Git installiert. Möglich ist, dass du stattdessen eines dieser Werkzeuge installiert hast:

- Git for Windows,
- GitHub Desktop,
- die GitHub CLI `gh`,
- eine IDE-Erweiterung oder eine Anmeldung in PhpStorm.

Präziser wäre zum Beispiel: „Ich habe Git for Windows installiert und mein Repository mit GitHub verbunden.“

„Meine Dateien sind sicher auf GitHub, weil ich einen Commit erstellt habe.“

Ein lokaler Commit ist noch nicht auf GitHub gesichert. Erst `git push` überträgt ihn zu einem Remote.

Prüfe den Zustand mit:

```
git status
```

Git meldet häufig ausdrücklich, wenn dein lokaler Branch Commits enthält, die noch nicht übertragen wurden.

„GitHub erstellt meine Commits.“

GitHub kann über die Weboberfläche Commits erzeugen, etwa beim Bearbeiten einer Datei im Browser oder beim Zusammenführen eines Pull Requests. Die meisten Commits erstellst du jedoch lokal mit Git oder über PhpStorm.

„Ohne GitHub kann ich Git nicht nutzen.“

Git funktioniert vollständig ohne GitHub. Das ist nützlich für:

- private Entwürfe,
- Projekte ohne Internetzugang,
- lokale Experimente,
- interne Server,
- vertrauliche Quelltexte,
- Schulungs- und Übungsprojekte.

„Ein Branch auf GitHub ist dasselbe wie mein lokaler Branch.“

Sie sind verwandt, aber nicht identisch. Beispielsweise sind `main` und `origin/main` unterschiedliche Referenzen:

- `main` bezeichnet deinen lokalen Branch.
- `origin/main` repräsentiert den zuletzt bekannten Stand des Branches auf dem Remote `origin`.

Git aktualisiert `origin/main` erst durch Kommunikation mit dem Remote, etwa per `git fetch`.

Die richtige Begriffswahl im Alltag

Eine präzise Sprache verhindert Missverständnisse im Team:

Ungenau	Präziser

„Ich habe es in GitHub committed.“	„Ich habe lokal committed und nach GitHub gepusht.“
„GitHub hat einen Konflikt.“	„Der Pull Request kann nicht automatisch gemergt werden.“
„Ich ziehe GitHub.“	„Ich hole Änderungen vom Remote mit `git pull`.“
„Der Branch ist auf GitHub.“	„Der lokale Branch wurde zu `origin/feature/login` gepusht.“
„Git ist kaputt.“	„Mein lokaler Branch und der Remote-Branch sind auseinander gelaufen.“

Diese Unterscheidung wird besonders wichtig, sobald mehrere Personen gleichzeitig an einem Projekt arbeiten.

Merksatz

> **Git ist das Werkzeug für Versionsgeschichte. GitHub ist ein Ort und eine Plattform für gemeinsame Git-Projekte.**

Wenn du lokal Dateien änderst, stagen, committen, branchen oder zurücksetzen möchtest, arbeitest du primär mit **Git**. Wenn du Änderungen veröffentlichst, Pull Requests prüfst, Issues planst oder automatisierte Tests auf einem Server ausführst, nutzt du Funktionen von **GitHub**.

Im nächsten Schritt lernst du typische Git-Workflows kennen und kannst Git sowie GitHub gezielt für Einzelarbeit und Teamarbeit einordnen.

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
```

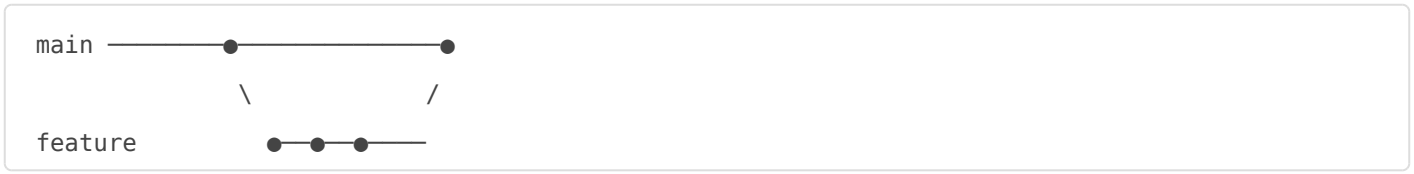
Ein Feature-Branch entsteht aus dem aktuellen Hauptbranch:

```
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.

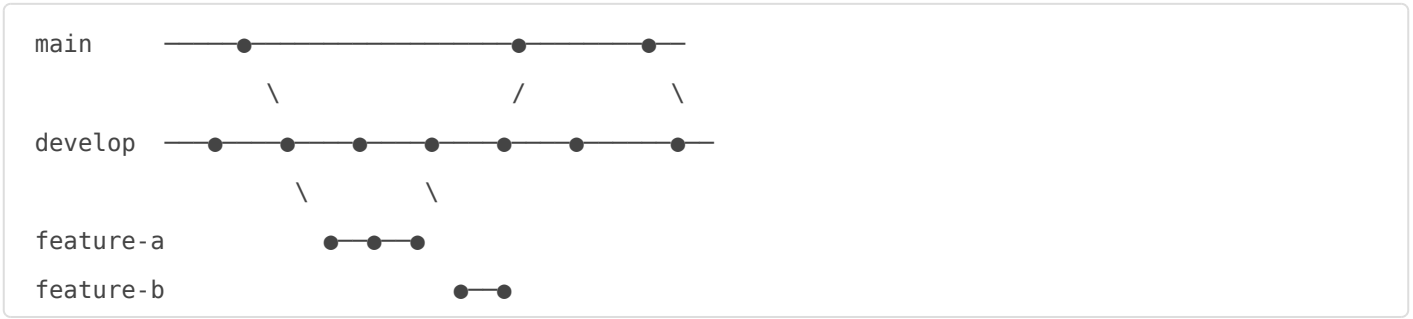
Git Flow

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

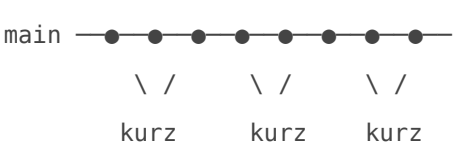
- Mehr Branches und mehr Merge-Vorgänge.
- Höherer organisatorischer Aufwand.
- Lang lebende Branches können Integrationen erschweren.
- Für kontinuierlich ausgelieferte Webanwendungen oft unnötig schwergewichtig.

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.

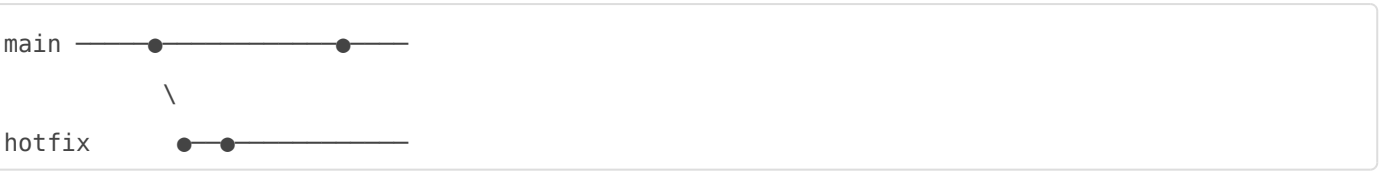
Beispiel:

```
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
Versionierte Software mit festen Releases	Git Flow oder ein vereinfachter Release-Branch-Workflow	Geplante Release-Phasen und Hotfixes klar abbildbar

Situation	Geeigneter Einstieg	Warum
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.

Häufige Anfängerfehler und Denkmodelle

Warum Denkmodelle wichtiger sind als Befehle

Viele Git-Probleme entstehen nicht, weil ein Befehl unbekannt ist, sondern weil das zugrunde liegende Modell fehlt. Wer Git nur als Sammlung von Kommandos lernt, merkt sich etwa:

```
git add .
git commit -m "Änderungen"
git push
```

Das funktioniert in einfachen Situationen. Sobald aber ein Konflikt, ein falscher Commit oder eine Änderung auf dem falschen Branch auftaucht, fehlt die Orientierung.

Ein belastbares Verständnis beginnt mit dieser Grundidee:

> Git verwaltet nicht einfach Dateien. Git verwaltet **Versionen von Projektzuständen** und die Beziehungen zwischen diesen Versionen.

Die folgenden Denkmodelle helfen dabei, Git-Situationen ruhig zu analysieren, statt Befehle auf Verdacht auszuprobieren.

Das wichtigste Modell: Git hat mehrere Zustände

Eine Datei kann in Git nicht nur „da“ oder „gespeichert“ sein. Für die tägliche Arbeit sind drei Bereiche entscheidend:

1. **Arbeitsverzeichnis**
   Die Dateien, die gerade im Editor liegen und bearbeitet werden.
2. **Staging Area** oder **Index**
   Die Auswahl der Änderungen, die im *nächsten* Commit landen soll.
3. **Lokales Repository**
   Die bereits erstellte, lokale Commit-Historie.

Später kommt häufig noch ein vierter Bezugspunkt hinzu:

4. **Entferntes Repository**
   Beispielsweise ein Repository auf GitHub.

Der typische Weg einer Änderung sieht so aus:

```
Datei bearbeiten
      ↓
Arbeitsverzeichnis
      ↓ git add
Staging Area
      ↓ git commit
lokale Historie
      ↓ git push
GitHub oder anderer Remote
```

Die zentrale Frage bei jedem Problem

Bevor du einen Git-Befehl eingibst, frage dich:

> **Wo befindet sich die Änderung gerade – und wohin soll sie?**

Diese Frage ist oft hilfreicher als die Suche nach einem vermeintlichen „Rückgängig-Befehl“.

Situation	Die Änderung befindet sich gerade	Typisches Ziel
Datei wurde bearbeitet, aber noch nicht vorgemerkt	Arbeitsverzeichnis	Staging Area oder verwerfen
Falsche Datei wurde für den Commit ausgewählt	Staging Area	Zurück ins Arbeitsverzeichnis

Situation	Die Änderung befindet sich gerade	Typisches Ziel
Commit wurde lokal erstellt, aber noch nicht veröffentlicht	Lokales Repository	Commit korrigieren oder Historie anpassen
Commit wurde bereits auf GitHub veröffentlicht	Remote und lokale Historie	Nachvollziehbar korrigieren, nicht unbedacht umschreiben

Der Befehl `git status` beantwortet genau diese Zustandsfrage und sollte daher zum Standard werden.

```
git status
```

Denkmodell: Ein Commit ist ein Snapshot, kein Änderungsprotokoll

Anfänger stellen sich einen Commit oft als Liste einzelner Änderungen vor, etwa:

> „In diesem Commit wurde Zeile 18 in Datei A angepasst.“

Das ist als Anzeige nicht falsch, aber als Modell unvollständig. Git speichert bei einem Commit grundsätzlich einen **vollständigen Snapshot des vorgemerkten Projektzustands**.

Ein Commit bedeutet eher:

> „So sah das gesamte Projekt zu diesem Zeitpunkt aus.“

Git kann daraus sehr effizient die Unterschiede zwischen zwei Snapshots berechnen. Deshalb wirken Commits in Tools und Diffs häufig wie Änderungslisten. Intern und gedanklich ist aber der Snapshot entscheidend.

Konsequenz für die Praxis

Ein Commit sollte einen **in sich stimmigen Zustand** festhalten:

- Die Anwendung lässt sich idealerweise ausführen.

- Tests sind möglichst grün.
- Die Änderung erfüllt genau einen nachvollziehbaren Zweck.
- Andere Teammitglieder können verstehen, *warum* der Snapshot entstanden ist.

Schlecht wäre ein Commit wie:

```
WIP
```

Oder:

```
Neue Funktion, CSS angepasst, Debug-Ausgaben, Composer aktualisiert
```

Besser wäre eine fachlich klare Trennung:

```
feat: E-Mail-Adresse bei der Registrierung validieren
```

```
test: Validierung ungültiger E-Mail-Adressen abdecken
```

```
style: Formularabstände auf der Registrierungsseite vereinheitlichen
```

Ein guter Commit ist keine Momentaufnahme der eigenen Arbeitszeit. Er ist eine verständliche Einheit in der gemeinsamen Projektgeschichte.

Denkmodell: Git speichert lokale Arbeit nicht automatisch auf GitHub

Ein besonders häufiger Irrtum lautet:

> „Ich habe einen Commit erstellt, also ist die Änderung jetzt auf GitHub.“

Das stimmt nicht. Ein Commit wird zunächst nur im **lokalen Repository** erstellt. Erst ein Push überträgt ihn an ein entferntes Repository.

```
git commit  →  lokale Historie
git push    →  entfernte Historie
```

Umgekehrt lädt `git pull` nicht einfach „alles Neue“ herunter. Es holt Änderungen vom Remote und integriert sie in den aktuellen lokalen Branch.

Vier Fragen zur Einordnung

Wenn du nicht sicher bist, ob etwas gesichert oder veröffentlicht ist, unterscheide präzise:

1. Wurde die Datei gespeichert?
2. Wurde die Änderung mit `git add` vorgemerkt?
3. Wurde ein lokaler Commit erstellt?
4. Wurde dieser Commit zu GitHub übertragen?

Diese vier Schritte sind unabhängig. Eine gespeicherte Datei ist noch kein Commit; ein lokaler Commit ist noch keine Veröffentlichung.

> **Merksatz:** Ein Commit sichert eine Version lokal. Ein Push teilt diese Version mit einem Remote.

Denkmodell: Git und GitHub sind verschiedene Systeme

Git und GitHub werden im Alltag oft zusammen genannt, erfüllen aber unterschiedliche Aufgaben.

- **Git** ist das lokale Versionskontrollsystem. Es verwaltet Commits, Branches, Historie und Änderungen auf dem eigenen Rechner.
- **GitHub** ist eine Plattform für gehostete Git-Repositories und Zusammenarbeit. Sie ergänzt Git unter anderem um Pull Requests, Issues, Reviews, Actions und Zugriffsverwaltung.

Du kannst Git vollständig ohne GitHub verwenden:

```
git init
git add .
git commit -m "Initialer Stand"
```

Und GitHub kann ohne ein lokales Repository kaum sinnvoll genutzt werden: GitHub speichert zwar Git-Repositories, ersetzt aber nicht die lokale Git-Arbeit.

Typischer Denkfehler

> „GitHub hat meinen Code verloren.“

Oft liegt die Änderung nur lokal vor, wurde auf einem anderen Branch erstellt oder noch nicht gepusht. Bevor du von Datenverlust ausgehst, prüfe deshalb:

```
git status
git log --oneline
git branch
```

Erst danach sollte nach Problemen bei GitHub, dem Netzwerk oder Berechtigungen gesucht werden.

Denkmodell: Ein Branch ist keine Projektkopie

Anfänger behandeln Branches häufig wie getrennte Projektordner:

> „Wenn ich einen Branch erstelle, wird das Projekt dupliziert.“

Ein Branch ist in Git vor allem ein **beweglicher Name für einen Commit**. Er markiert den aktuellen Endpunkt einer Entwicklungslinie.

```
main     → Commit A
feature  → Commit A
```

Sobald auf `feature` ein neuer Commit entsteht, bewegt sich nur dieser Branch weiter:

```
main     → Commit A
feature  → Commit B
```

Das Projekt wird nicht jedes Mal vollständig kopiert. Diese Leichtgewichtigkeit ist einer der Gründe, warum Branches in Git so günstig und alltäglich sind.

Praktische Konsequenz

Du darfst Branches häufig und gezielt einsetzen:

- für ein Feature,
- für einen Fehler,
- für ein Experiment,
- für eine Dokumentationsänderung,
- für einen Pull Request.

Ein Branch ist kein Zeichen dafür, dass eine Aufgabe riesig oder riskant sein muss. Im Gegenteil: Kleine, kurzlebige Branches machen Änderungen besser isolierbar und überprüfbar.

Denkmodell: HEAD zeigt, wo du gerade arbeitest

`HEAD` ist ein zentraler Begriff, der zunächst abstrakt wirken kann. Vereinfacht bedeutet er:

> **HEAD markiert den aktuell ausgecheckten Stand.**

Meist zeigt `HEAD` auf den aktuell aktiven Branch. Wenn du auf `main` arbeitest, zeigt `HEAD` auf `main`. Ein neuer Commit verschiebt dann den Branch und damit indirekt auch `HEAD`.

```
HEAD → main → letzter Commit
```

Wenn du den Branch wechselst, wechselt auch dein Arbeitskontext:

```
HEAD → feature/login → letzter Feature-Commit
```

Warum das wichtig ist

Bevor du commitest, mergest, zurücksetzt oder Dateien wiederherstellst, solltest du wissen:

- Auf welchem Branch befinde ich mich?
- Welcher Commit ist aktuell ausgecheckt?
- Soll die nächste Änderung wirklich hier entstehen?

Diese Informationen liefert:

```
git status
```

Auch PhpStorm zeigt den aktuellen Branch deutlich in der Statusleiste und im Branch-Menü an. Ein kurzer Blick vor einer größeren Aktion verhindert viele Fehler.

Häufiger Fehler: Direkt auf `main` arbeiten

Gerade bei Einzelprojekten wirkt es zunächst bequem, alle Änderungen direkt auf `main` vorzunehmen. In Teams und bei Pull-Request-Workflows führt das jedoch oft zu Problemen:

- Unfertige Arbeit vermischt sich mit stabilem Code.
- Mehrere Aufgaben werden schwer voneinander trennbar.
- Reviews werden unübersichtlich.
- Ein Fehler ist schwieriger isoliert zurückzunehmen.
- Synchronisierung mit anderen wird konfliktanfälliger.

Ein besseres Grundmuster ist:

```
main
  └── feature/registrierung-validieren
```

Die Arbeit findet auf dem Feature-Branch statt. Erst wenn sie fertig, getestet und überprüft ist, wird sie in den Hauptbranch integriert.

Ausnahme: Kleine persönliche Experimente

In einem lokalen Lernrepository oder bei einer sehr kleinen, isolierten Änderung kann direkte Arbeit auf `main` vertretbar sein. Entscheidend ist nicht ein starres Verbot, sondern die Frage:

> **Muss diese Änderung getrennt entwickelt, geprüft oder später nachvollzogen werden?**

Wenn die Antwort „ja“ lautet, ist ein eigener Branch sinnvoll.

Häufiger Fehler: Alles mit `git add .` vormerken

Der Befehl

```
git add .
```

ist nicht grundsätzlich falsch. Er nimmt jedoch viele Änderungen auf einmal in die Staging Area. Das kann versehentlich einschließen:

- Debug-Ausgaben,
- lokale Konfigurationsdateien,
- nicht fertiggestellte Änderungen,
- Zugangsdaten,
- IDE-Dateien,
- Änderungen einer anderen Aufgabe.

Besseres Denkmodell: Die Staging Area ist eine bewusste Auswahl

Die Staging Area ist kein lästiger Zwischenschritt. Sie ist eine **Zusammenstellung des nächsten Commits**.

Statt gedankenlos alles vorzumerken, prüfe zunächst:

```
git status
git diff
```

Danach kannst du gezielt einzelne Dateien hinzufügen:

```
git add src/Validator.php
git add tests/ValidatorTest.php
```

Später lernst du auch, einzelne Teile einer Datei vorzumerken. Damit lassen sich logisch getrennte Commits erstellen, selbst wenn mehrere Änderungen gleichzeitig im Arbeitsverzeichnis liegen.

> **Merksatz:** `git add` bedeutet nicht „speichern“, sondern „für den nächsten Snapshot auswählen“.

Häufiger Fehler: Den Status nicht prüfen

Viele riskante Git-Aktionen beginnen mit einer Vermutung:

- „Ich glaube, alles ist committed.“
- „Ich bin bestimmt auf dem richtigen Branch.“
- „Die Änderung müsste schon auf GitHub sein.“
- „Der Merge müsste fertig sein.“

Git verlangt jedoch keine Vermutungen. Es liefert Informationen.

Der sichere Mini-Check

Vor und nach wichtigen Schritten lohnt sich:

```
git status
```

Achte dabei auf:

- den aktuellen Branch,
- nicht verfolgte Dateien,
- geänderte, aber nicht vorgemerkte Dateien,
- vorgemerkte Änderungen,
- lokale Commits vor oder hinter dem Upstream-Branch,
- laufende Operationen wie Merge oder Rebase.

Ergänzend helfen:

```
git log --oneline --decorate -n 10
```

```
git diff
```

```
git diff --staged
```

Damit beantwortest du drei verschiedene Fragen:

Frage	Geeigneter Befehl
In welchem Zustand befindet sich das Repository?	`git status`
Was wurde noch nicht vorgemerkt?	`git diff`
Was landet im nächsten Commit?	`git diff --staged`
Welche Commits liegen zuletzt vor?	`git log --oneline`

Häufiger Fehler: Commit und Push verwechseln

Ein klassischer Ablauf sieht so aus:

1. Datei bearbeiten
2. Commit erstellen
3. Browser öffnen
4. Auf GitHub keine Änderung finden
5. Verunsicherung

Die Ursache ist meist einfach: Der Push fehlt.

```
git push
```

Umgekehrt kann auch ein Push scheitern, obwohl der Commit erfolgreich erstellt wurde. Das betrifft dann nicht die lokale Historie, sondern den Datentransfer zum Remote. Typische Ursachen sind:

- fehlende Anmeldung,
- falsche Berechtigung,
- ein anderer Stand auf dem Remote,
- Netzwerkprobleme,
- ein noch nicht eingerichteter Upstream-Branch.

Sicheres Vorgehen

Wenn ein Push fehlschlägt:

1. Lies die Fehlermeldung vollständig.
2. Prüfe mit `git status`, ob der lokale Commit vorhanden ist.
3. Prüfe den aktuellen Branch.
4. Prüfe erst dann Anmeldung, Remote-URL oder Synchronisationskonflikte.

Wiederhole nicht blind denselben Befehl und nutze insbesondere keinen erzwungenen Push, nur um eine Fehlermeldung verschwinden zu lassen.

Häufiger Fehler: `pull` als harmlose Aktualisierung betrachten

`git pull` wird oft als reiner Download verstanden. Tatsächlich besteht ein Pull vereinfacht aus zwei Schritten:

1. Änderungen vom Remote abrufen.
2. Diese Änderungen lokal integrieren.

Das Integrieren kann einen Merge oder Rebase auslösen und unter Umständen Konflikte erzeugen.

Besseres Denkmodell

Ein Pull verändert möglicherweise den lokalen Projektzustand. Deshalb ist es sinnvoll, vorher zu prüfen:

```
git status
```

Sind lokale Änderungen vorhanden, solltest du wissen, ob sie mit den eingehenden Änderungen kollidieren könnten.

Für mehr Kontrolle kann es hilfreich sein, gedanklich zwischen zwei Aufgaben zu unterscheiden:

```
git fetch
```

holt Informationen und neue Commits vom Remote, ohne den aktuellen Branch direkt zu verändern.

Danach kannst du untersuchen, was sich geändert hat, und die Integration bewusst durchführen. Die konkreten Varianten werden später ausführlich behandelt; wichtig ist hier vor allem das Verständnis:

> Ein Pull ist nicht nur Empfang, sondern auch Integration.

Häufiger Fehler: Konfliktmarker als Git-Fehler missverstehen

Wenn Git einen Merge-Konflikt meldet, bedeutet das nicht, dass Git beschädigt ist oder dass etwas „kaputtgegangen“ ist. Git kann zwei Änderungen lediglich nicht eindeutig automatisch kombinieren.

In einer Datei können dann Markierungen wie diese erscheinen:

```
<<<<<<< HEAD
Lokale Variante
=======
Eingehende Variante
>>>>>>> anderer-branch
```

Diese Markierungen sind eine Aufforderung zur Entscheidung:

- Welche Variante ist fachlich korrekt?
- Müssen beide Varianten kombiniert werden?
- Fehlt Kontext aus Tests, Anforderungen oder Gesprächen?

Der richtige Umgang mit Konflikten

1. Nicht in Panik geraten.
2. Den Konfliktbereich und den fachlichen Zweck verstehen.
3. Die gewünschte Endfassung schreiben.
4. Konfliktmarker vollständig entfernen.
5. Datei testen und prüfen.
6. Die Git-Operation kontrolliert fortsetzen.

Ein Konflikt ist kein Makel. Er zeigt, dass zwei Entwicklungslinien dieselbe Stelle unterschiedlich verändert haben und eine menschliche Entscheidung erforderlich ist.

Häufiger Fehler: Änderungen vor einem Branch-Wechsel ignorieren

Vor dem Wechsel des Branches sollten lokale Änderungen bewusst behandelt werden. Andernfalls entstehen Unsicherheiten:

- Gehört diese Änderung zum alten oder neuen Branch?
- Wurde sie unbeabsichtigt übernommen?
- Verhindert sie den Wechsel?
- Wird sie später versehentlich committet?

Praktische Regel

Vor einem Branch-Wechsel:

```
git status
```

Danach bewusst entscheiden:

- Änderungen committen, wenn sie einen sinnvollen Zwischenstand bilden.
- Änderungen gezielt verwerfen, wenn sie nicht gebraucht werden.
- Änderungen vorübergehend sicher ablegen, wenn sie später fortgesetzt werden sollen.

Wichtig ist nicht, immer sofort zu committen. Wichtig ist, den Arbeitszustand nicht unbemerkt mitzunehmen.

Häufiger Fehler: Unfertige oder gemischte Commits erstellen

Ein Commit mit vielen unabhängigen Änderungen erschwert fast alles:

- Reviews,

- Fehlersuche,
- Rücknahme einzelner Änderungen,
- Cherry-Picks,
- spätere Historienanalyse,
- Verständlichkeit für andere Personen.

Ein Beispiel für einen problematischen Commit:

```
Fixes und Änderungen
```

Er enthält vielleicht gleichzeitig:

- einen Bugfix,
- eine Umbenennung,
- Formatierungsänderungen,
- eine neue Funktion,
- aktualisierte Abhängigkeiten.

Besseres Denkmodell: Ein Commit beantwortet eine Frage

Ein guter Commit sollte idealerweise eine klare Frage beantworten können:

> **Welchen Zweck erfüllt diese Änderung?**

Beispiele:

- „Warum wurde die Passwortlänge angepasst?“
- „Warum wird diese Ausnahme jetzt abgefangen?“
- „Warum wurde die Abhängigkeit aktualisiert?“
- „Warum wurde die Datenbankabfrage umgestellt?“

Wenn ein Commit mehrere unabhängige Antworten benötigt, sollte er wahrscheinlich aufgeteilt werden.

Häufiger Fehler: Geheimnisse und lokale Dateien committen

Ein Repository ist kein privater Arbeitsordner. Alles, was committed wird, kann langfristig in der Historie verbleiben und nach einem Push für weitere Personen sichtbar werden.

Besonders kritisch sind:

- Passwörter,
- API-Schlüssel,
- Tokens,
- private Schlüssel,
- Datenbank-Zugangsdaten,
- produktive Konfigurationsdateien,
- personenbezogene Testdaten.

Auch wenn eine Datei später gelöscht wird, kann sie in älteren Commits weiterhin vorhanden sein.

Sicheres Denkmodell

> Was committed wird, gehört zur nachvollziehbaren Projektgeschichte.

Prüfe daher vor jedem Commit die vorgemerkten Änderungen:

```
git diff --staged
```

Für Konfigurationen eignet sich oft dieses Muster:

```
.env.example
```

wird versioniert und enthält nur Platzhalter, während die echte lokale Datei etwa über `.gitignore` ausgeschlossen wird:

```
.env
```

Die Regeln für Ignore-Dateien und die sichere Behandlung von Geheimnissen folgen in einem eigenen Kapitel. Die wichtigste Gewohnheit beginnt jedoch sofort: **Nie sensible Inhalte ungeprüft stagen oder committen.**

Häufiger Fehler: Dateilöschung mit Versionsverlust gleichsetzen

Wenn eine Datei im Arbeitsverzeichnis gelöscht wird, ist sie nicht automatisch für immer verloren. Wenn sie bereits committed war, kennt Git ältere Versionen weiterhin.

Das führt zu einem wichtigen Denkmodell:

> Die aktuelle Arbeitskopie ist nur ein sichtbarer Zustand, nicht die gesamte Historie.

Solange ein Commit erreichbar ist, lassen sich frühere Inhalte meist wieder anzeigen oder wiederherstellen. Dennoch gilt: Nicht jede ungespeicherte oder nie committete Änderung kann Git retten.

Was Git typischerweise retten kann

Situation	Rettungschance
Datei war in einem Commit enthalten	Sehr gut
Commit wurde lokal erstellt, aber noch nicht gepusht	Meist sehr gut
Branch wurde versehentlich gelöscht	Häufig gut
Datei wurde nur bearbeitet, nie gespeichert und nie committed	Git kann nicht helfen
Nicht vorgemerkte Änderung wurde bewusst überschrieben	Nur mit anderen Sicherungen eventuell möglich

Git ist eine starke Absicherung für versionierte Zustände, aber kein Ersatz für bewusstes Speichern, Backups oder vorsichtigen Umgang mit destruktiven Befehlen.

Häufiger Fehler: Befehle aus dem Internet blind kopieren

Bei Fehlermeldungen ist die Versuchung groß, einen gefundenen Befehl sofort auszuführen. Besonders gefährlich sind Befehle mit Optionen wie:

```
--hard
--force
--force-with-lease
```

Diese Optionen sind nicht grundsätzlich falsch. Sie haben wichtige, legitime Anwendungsfälle. Ohne Verständnis können sie jedoch lokale Arbeit entfernen oder gemeinsame Historien überschreiben.

Die Drei-Fragen-Regel vor risikoreichen Befehlen

Bevor du einen Befehl ausführst, frage:

1. **Was wird dadurch verändert?**
   Arbeitsverzeichnis, Staging Area, lokale Historie, Remote oder mehrere Bereiche?
2. **Welche Daten könnten verloren gehen?**
   Nicht gespeicherte, nicht vorgemerkte, lokale oder veröffentlichte Änderungen?
3. **Wie komme ich zurück, falls das Ergebnis falsch ist?**
   Gibt es einen Commit, einen Branch, einen Stash, ein Backup oder einen nachvollziehbaren Ausgangspunkt?

Wenn du diese Fragen nicht beantworten kannst, stoppe und untersuche den Zustand. Ein zusätzlicher Blick auf `git status` kostet Sekunden; eine beschädigte gemeinsame Historie kann dagegen viel Zeit kosten.

Häufiger Fehler: Git als Gegner betrachten

Git-Meldungen wirken am Anfang oft streng oder unverständlich. Häufig schützen sie aber vor einem riskanten Zustand.

Beispiele:

- Git verweigert einen Branch-Wechsel, damit lokale Änderungen nicht überschrieben werden.

- Git verweigert einen Push, damit fremde Commits nicht unbemerkt verloren gehen.
- Git meldet einen Konflikt, weil keine sichere automatische Entscheidung möglich ist.
- Git verweigert einen Commit, wenn nichts vorgemerkt wurde.

Statt die Meldung als Hindernis zu sehen, lies sie als Zustandsbeschreibung:

> „Git erklärt mir, welche Annahme gerade nicht erfüllt ist.“

Diese Haltung verändert die Fehlersuche grundlegend. Nicht „Wie zwinge ich Git dazu?“, sondern:

> **„Welchen Zustand schützt Git – und was möchte ich fachlich wirklich erreichen?“**

Ein systematisches Denkmodell für jede Git-Situation

Wenn etwas unerwartet passiert, hilft dieser Ablauf:

1. Anhalten

Keine weiteren Befehle auf Verdacht ausführen. Insbesondere keine Befehle, die Historie oder Arbeitsdateien überschreiben könnten.

2. Zustand erfassen

```
git status
```

Optional ergänzen:

```
git log --oneline --decorate -n 10
```

```
git branch --all
```

3. Ziel formulieren

Beschreibe ohne Git-Befehl, was erreicht werden soll:

- „Diese Datei soll nicht im nächsten Commit landen.“
- „Mein lokaler Branch soll die Änderungen vom Server enthalten.“
- „Der letzte Commit hat die falsche Nachricht.“
- „Ich möchte den Bugfix aus einem anderen Branch übernehmen.“

Eine klare Zielformulierung verhindert, dass ein unpassender Befehl nur zufällig plausibel wirkt.

4. Betroffenen Bereich bestimmen

Ziel	Hauptbereich
Datei bearbeiten oder verwerfen	Arbeitsverzeichnis
Auswahl für nächsten Commit ändern	Staging Area
Lokalen Commit korrigieren	Lokale Historie
Änderungen austauschen	Remote und Tracking-Branch
Entwicklung trennen	Branches

5. Kleinste sichere Aktion wählen

Bevorzuge eine gezielte, nachvollziehbare Aktion gegenüber einer globalen oder erzwungenen Aktion.

Beispiel: Wenn nur eine Datei versehentlich vorgemerkt wurde, ist es besser, genau diese Datei aus der Staging Area zu nehmen, als pauschal große Teile des Repository-Zustands zurückzusetzen.

6. Ergebnis prüfen

Nach jeder Veränderung:

```
git status
```

Und bei relevanten Änderungen:

```
git diff
```

Diese Schleife aus **analysieren → handeln → prüfen** ist eine der wichtigsten professionellen Git-Gewohnheiten.

Ein praktischer Vor-Commit-Check

Vor einem Commit kannst du diese kurze Checkliste verwenden:

- ☐ Bin ich auf dem richtigen Branch?
- ☐ Weiß ich, welchen Zweck dieser Commit erfüllt?
- ☐ Enthält der Commit nur zusammengehörige Änderungen?
- ☐ Habe ich die vorgemerkten Änderungen geprüft?
- ☐ Sind keine Zugangsdaten, Debug-Ausgaben oder lokalen Dateien enthalten?
- ☐ Ist die Commit-Nachricht konkret und verständlich?
- ☐ Wurde die relevante Anwendung oder der relevante Test ausgeführt?

Die technische Prüfung beginnt mit:

```
git status
```

Dann:

```
git diff --staged
```

Erst danach folgt der Commit.

Ein praktischer Vor-Push-Check

Ein Push veröffentlicht Arbeit für andere Personen und häufig auch für Automatisierungen. Prüfe daher kurz:

- ☐ Liegen die richtigen Commits auf dem aktuellen Branch?
- ☐ Sind Tests oder Qualitätsprüfungen erfolgreich?
- ☐ Enthält die Historie keine versehentlichen oder geheimen Inhalte?
- ☐ Ist klar, in welches Remote-Repository und auf welchen Branch gepusht wird?
- ☐ Wurden aktuelle Änderungen des Teams berücksichtigt?

Danach ist ein Push keine blinde Routine, sondern eine bewusste Veröffentlichung.

Die wichtigsten Merksätze

> **Git verwaltet Zustände und Historien, nicht nur Dateien.**

> **Ein Commit ist ein lokaler Snapshot des vorgemerkten Projektzustands.**

> **`git add` wählt Inhalte für den nächsten Commit aus.**

> **Ein Push veröffentlicht lokale Commits auf einem Remote; ein Commit allein tut das nicht.**

> **Ein Branch ist eine leichte Entwicklungslinie, keine vollständige Projektkopie.**

> **Ein Konflikt ist eine notwendige fachliche Entscheidung, kein Git-Defekt.**

> **Vor jeder riskanten Aktion zuerst den Zustand prüfen.**

> **`git status` ist kein Notfallwerkzeug, sondern ein alltäglicher Kompass.**

Mit diesen Denkmodellen werden die folgenden Git-Befehle nicht mehr wie isolierte Zauberformeln wirken. Sie werden zu gezielten Werkzeugen, mit denen du Änderungen sicher zwischen klar erkennbaren Zuständen bewegst.