Kapitel 1: Orientierung – Was sind Git, GitHub und warum brauche ich das?

Einleitung: Bevor du in die Praxis einsteigst, ist es wichtig zu verstehen, welches Problem Git eigentlich löst. Stell dir vor, du arbeitest wochenlang an einem Projekt, änderst Code, löschst Dateien – und plötzlich funktioniert nichts mehr. Ohne Versionsverwaltung hast du keine Möglichkeit, zu einem funktionierenden Stand zurückzukehren. Git ist wie eine Zeitmaschine für deinen Code: Es speichert jeden Zwischenstand deines Projekts und ermöglicht dir, jederzeit in die Vergangenheit zu reisen. GitHub ergänzt Git um eine Online-Plattform, auf der du deinen Code sichern und mit anderen teilen kannst. In diesem Kapitel lernst du die grundlegenden Konzepte und Begriffe kennen, die du für den Rest des Kurses brauchst.

Was ist Versionsverwaltung – und warum ist Git so wichtig? □□

Versionsverwaltung ist ein System, mit dem du Änderungen an Dateien — vor allem an Quellcode — **nachvollziehen, organisieren und bei Bedarf rückgängig machen** kannst.

Stell dir vor, du arbeitest an einem Projekt und speicherst Dateien so ab:

- `projekt_final.php`
- `projekt_final_neu.php`
- `projekt_final_wirklich_final.php`
- `projekt_final_v2_korrigiert.php`

□□ Genau dieses Chaos versucht Versionsverwaltung zu verhindern.

Stattdessen speichert ein Versionsverwaltungssystem **gezielt, wer was wann geändert hat** und erlaubt dir, auf frühere Stände zurückzugehen, Änderungen zu vergleichen und sauber weiterzuarbeiten.

Das Grundproblem ohne Versionsverwaltung

Wenn du **ohne Git** an echten Projekten arbeitest, tauchen sehr schnell typische Probleme auf:

1. **Du verlierst den Überblick über Änderungen**
   - Was hast du gestern geändert?
   - Warum funktioniert plötzlich etwas nicht mehr?
   - Welche Datei war zuletzt noch korrekt?
2. **Fehler lassen sich schwer zurückverfolgen**
   - Ein Bug ist plötzlich da, aber du weißt nicht, *ab wann*.
   - Du kannst nicht einfach elegant zu einem funktionierenden Stand zurückspringen.
3. **Experimentieren wird riskant**

- Du willst etwas umbauen, hast aber Angst, das funktionierende System kaputtzumachen.
- Deshalb kopierst du oft ganze Projektordner als „Sicherheitskopie“.

4. **Zusammenarbeit wird chaotisch**
   - Wenn mehrere Personen dieselben Dateien ändern, ist schnell unklar:
     - Welche Version ist die richtige?
     - Welche Änderung soll behalten werden?
     - Wer hat was gemacht?
5. **Deployment und Releases werden unübersichtlich**
   - Du weißt irgendwann nicht mehr genau, welcher Stand live gegangen ist.
   - Hotfixes und neue Features vermischen sich.

Was Versionsverwaltung konkret macht

Eine Versionsverwaltung hilft dir dabei, dein Projekt **historisch und strukturiert** zu verwalten.

Typische Funktionen sind:

- **Änderungen speichern**
- **Frühere Zustände wiederherstellen**
- **Änderungen vergleichen**
- **Entwicklungsstände dokumentieren**
- **Paralleles Arbeiten ermöglichen**
- **Experimente in separaten Entwicklungszweigen erlauben**

Im Kern entsteht eine **Projektgeschichte**, die nicht nur den aktuellen Stand zeigt, sondern auch den Weg dorthin.

Und was ist Git dabei genau?

Git ist ein sehr verbreitetes **Versionsverwaltungssystem**, speziell für Softwareprojekte.

Es ist dafür gemacht, dass du:

- Änderungen lokal auf deinem Rechner verwalten kannst
- in sinnvollen Schritten speichern kannst
- verschiedene Entwicklungsstände verwalten kannst

- mit anderen gemeinsam an einem Projekt arbeiten kannst

Der wichtigste Gedanke dabei ist:

> Git speichert nicht einfach nur „Dateien“, sondern die **Entwicklung deines Projekts in nachvollziehbaren Schritten**.

Diese Schritte heißen **Commits**.

Ein Commit ist vereinfacht gesagt ein **gespeicherter Meilenstein** deines Projekts.

Zum Beispiel:

- „Login-Formular erstellt“
- „Fehler bei Passwortprüfung behoben“
- „Navigation umgebaut“
- „Datenbankverbindung auf PDO umgestellt“

So entsteht nach und nach eine saubere Historie.

Welches Problem löst Git in echten Programmierprojekten?

Git löst nicht nur *ein* Problem, sondern gleich mehrere zentrale Probleme in der Praxis.

1. Git schafft Sicherheit

Wenn du an Code arbeitest, wirst du Fehler machen — das ist völlig normal.
Mit Git ist das nicht dramatisch, weil du frühere Stände wiederfinden kannst.

Das bedeutet:

- Du kannst mutiger arbeiten
- Du kannst Änderungen testen
- Du kannst notfalls zurückrollen

Ohne Git fühlt sich jeder größere Umbau riskant an.
Mit Git weißt du: „Wenn etwas schiefgeht, komme ich wieder zurück.“

---

2. Git macht Änderungen nachvollziehbar

Git zeigt dir genau:

- welche Dateien geändert wurden
- *was* in diesen Dateien geändert wurde
- wann die Änderung passiert ist
- mit welcher Nachricht sie gespeichert wurde

Das ist extrem hilfreich, wenn du dich fragst:

- „Warum funktioniert diese Funktion heute anders als gestern?“
- „Wann habe ich diesen Code eingebaut?“
- „Welche Änderung hat wahrscheinlich den Bug verursacht?“

Gerade bei Lernprojekten ist das Gold wert, weil du dadurch auch deinen eigenen Fortschritt besser verstehst.

---

3. Git ermöglicht sauberes Experimentieren mit Branches

Ein **Branch** ist ein separater Entwicklungszweig.

Damit kannst du zum Beispiel:

- ein neues Feature bauen
- einen Bugfix testen
- ein Redesign ausprobieren

…ohne den stabilen Hauptstand direkt zu gefährden.

Das ist viel besser als:

- Dateien umzubenennen
- Ordner zu duplizieren
- Code auszukommentieren „für später“

Mit Git kannst du also **geordnet experimentieren** statt **chaotisch improvisieren**.

---

4. Git verbessert die Zusammenarbeit

Sobald mehrere Menschen an einem Projekt arbeiten, wird Versionsverwaltung praktisch unverzichtbar.

Git hilft dabei, dass:

1. jede Person ihre Änderungen machen kann
2. diese Änderungen zusammengeführt werden können
3. Konflikte sichtbar werden, wenn zwei Personen dieselbe Stelle geändert haben
4. die Projektgeschichte erhalten bleibt

Dadurch wird Teamarbeit überhaupt erst kontrollierbar.

---

5. Git schafft professionelle Arbeitsweise

Fast alle modernen Softwareprojekte nutzen Git oder ein ähnliches System.
Wenn du Git lernst, lernst du also nicht nur ein Tool, sondern auch eine **grundlegende Arbeitsweise der Softwareentwicklung**.

Dazu gehören unter anderem:

- in kleinen, sinnvollen Schritten arbeiten
- Änderungen bewusst dokumentieren
- neue Features getrennt entwickeln
- Bugs gezielt zurückverfolgen
- mit Online-Plattformen wie GitHub zusammenarbeiten

Ein einfaches Alltagsbeispiel

Stell dir vor, du schreibst an einer Hausarbeit.

Ohne Versionsverwaltung würdest du vielleicht so arbeiten:

- `hausarbeit.docx`
- `hausarbeit_neu.docx`
- `hausarbeit_mit_korrektur.docx`
- `hausarbeit_final.docx`
- `hausarbeit_final_final.docx`

Mit Versionsverwaltung hättest du stattdessen eine geordnete Historie:

1. Gliederung erstellt
2. Einleitung geschrieben
3. Kapitel 2 ergänzt
4. Rechtschreibung korrigiert
5. Fazit überarbeitet

Und du könntest jederzeit sagen:

- „Zeig mir, was sich seit gestern geändert hat."
- „Stell den Stand von vorgestern wieder her."
- „Ich möchte eine alternative Version des Fazits ausprobieren."

Genau das macht Git — nur eben für Code und Projekte.

Warum ist das gerade für dich als Einsteiger sinnvoll?

Vielleicht denkst du am Anfang:

> „Ich programmiere doch erstmal nur allein — brauche ich Git wirklich schon?"

Ja, absolut. Gerade dann.

Denn Git hilft dir schon früh dabei:

- strukturiert zu arbeiten
- weniger Angst vor Fehlern zu haben
- Änderungen bewusst zu machen
- Projekte sauber aufzubauen
- später leichter mit GitHub und Teams zu arbeiten

Wenn du Git erst sehr spät lernst, hast du dir oft schon unpraktische Gewohnheiten angewöhnt.
Wenn du Git früh lernst, wird sauberes Arbeiten von Anfang an normal. □□

Was Git *nicht* ist

Wichtig ist auch, ein paar Missverständnisse zu vermeiden:

Git ist **nicht**:

- nur ein Backup-System
- nur für große Teams gedacht
- nur für Profis
- nur für Open-Source-Projekte
- dasselbe wie GitHub

Git ist das Versionsverwaltungssystem.
GitHub ist eine Online-Plattform, auf der Git-Repositories gespeichert und geteilt werden können.

Diesen Unterschied schauen wir uns im nächsten Schritt noch genauer an.

Kurz zusammengefasst

Versionsverwaltung bedeutet, Änderungen an einem Projekt systematisch zu speichern und nachvollziehbar zu machen.

Git löst dabei in echten Programmierprojekten vor allem diese Probleme:

- Chaos durch viele Dateikopien
- fehlende Rückgängig-Möglichkeiten
- unklare Änderungsverläufe
- riskante Experimente
- schwierige Zusammenarbeit

Oder in einem Satz:

> **Git gibt deinem Projekt ein Gedächtnis.**

Merksatz □□

Ohne Versionsverwaltung arbeitest du oft nur am *aktuellen Zustand*.
Mit Git arbeitest du zusätzlich mit der *gesamten Geschichte deines Projekts*.

Die wichtigsten Git-Grundbegriffe für Einsteiger □□

Bevor du mit Git arbeitest, solltest du ein paar **zentrale Begriffe** verstehen. Diese tauchen in fast jeder Anleitung auf – und sobald du sie verinnerlicht hast, wird alles andere viel leichter verständlich.

Repository – dein Projektarchiv □□

Ein **Repository** (kurz: *Repo*) ist im Grunde ein **Ordner mit Gedächtnis**. Es enthält alle Dateien deines Projekts *und* die komplette Änderungshistorie – also wer wann was geändert hat.

Es gibt zwei Arten:

- **Lokales Repository:** Liegt auf deinem Computer, nur du hast Zugriff.
- **Remote Repository:** Liegt auf einem Server (z. B. GitHub), kann von mehreren Personen genutzt werden.

> □□ *Wenn du in PhpStorm ein Projekt „unter Git-Kontrolle stellst", erstellst du ein lokales Repository.*

Stage (Staging Area) – die Vorbereitungszone □□

Die **Staging Area** ist eine Art **Wartezone zwischen deinen Änderungen und dem nächsten Commit**. Wenn du eine Datei änderst, ist sie zunächst nur im sogenannten *Working Directory*

verändert. Erst wenn du sie „stagst", signalisierst du Git: *„Diese Änderung soll Teil des nächsten Commits sein."*

Das Vorgehen ist also:

1. Du änderst Dateien (Working Directory)
2. Du wählst aus, welche Änderungen gespeichert werden sollen → **Stage**
3. Du speicherst diese Auswahl dauerhaft → **Commit**

> □□ *Das ist praktisch, weil du nicht immer alle Änderungen auf einmal speichern musst, sondern gezielt auswählen kannst.*

Commit – ein Schnappschuss deines Projekts □□

Ein **Commit** ist ein **gespeicherter Zustand deines Projekts zu einem bestimmten Zeitpunkt**. Jeder Commit enthält eine eindeutige ID (einen sogenannten *Hash*), eine Nachricht, die beschreibt was geändert wurde, den Autor und das Datum sowie die tatsächlichen Änderungen an den Dateien.

Commits sind das Herzstück von Git. Du kannst sie dir wie **Speicherpunkte in einem Videospiel** vorstellen – du kannst jederzeit zu einem früheren Commit zurückkehren.

```
Beispiel einer Commit-Historie:
──────────────────────────────────

Commit 3: "Login-Formular validiert"
Commit 2: "Login-Seite erstellt"
Commit 1: "Projektstruktur angelegt"
```

Branch – parallele Entwicklungslinien □□

Ein **Branch** ist ein **eigenständiger Entwicklungszweig**. Standardmäßig arbeitest du auf dem Branch namens `main` (früher oft `master`). Du kannst aber jederzeit einen neuen Branch erstellen, um z. B. ein neues Feature zu entwickeln – ohne den Hauptzweig zu beeinflussen.

```
gitGraph
    commit id: "Start"
    commit id: "Basis fertig"
    branch feature/login
    commit id: "Login-Seite"
    commit id: "Validierung"
    checkout main
    commit id: "Bugfix"
    merge feature/login id: "Merge"
```

> □□ *Branches sind wie parallele Universen deines Projekts – du kannst experimentieren, ohne das „echte" Projekt zu gefährden.*

Merge – Zweige zusammenführen □□

Wenn du mit der Arbeit in einem Branch fertig bist, möchtest du diese Änderungen meist zurück in den `main`-Branch bringen. Diesen Vorgang nennt man **Merge**.

Git versucht dabei, die Änderungen automatisch zusammenzuführen. Falls jedoch *dieselbe Stelle* in beiden Branches unterschiedlich geändert wurde, entsteht ein **Merge-Konflikt**, den du manuell lösen musst. PhpStorm bietet dafür ein praktisches visuelles Tool.

Remote – die Verbindung zur Außenwelt □□

Ein **Remote** ist eine **Referenz auf ein externes Repository**, typischerweise auf GitHub, GitLab oder Bitbucket. Der Standard-Remote heißt meist `origin`.

Wenn du ein Projekt auf GitHub hast und lokal damit arbeitest, ist `origin` die Brücke zwischen deinem Computer und GitHub:

Begriff	Bedeutung
`origin`	Der Name des Remote-Repositorys (meist auf GitHub)
Remote-URL	Die Adresse, z. B. `https://github.com/user/projekt.git`

Clone – ein bestehendes Projekt herunterladen □□

Clone bedeutet, ein komplettes Remote-Repository auf deinen Computer zu kopieren – inklusive aller Dateien *und* der gesamten Historie.

```
git clone https://github.com/beispiel/projekt.git
```

> □□ *In PhpStorm kannst du über „Get from VCS" direkt ein GitHub-Projekt clonen und sofort loslegen.*

Push und Pull – Synchronisation mit dem Remote □□

Diese beiden Begriffe beschreiben den **Datenaustausch zwischen deinem lokalen Repository und dem Remote**:

- **Push** □□: Du **sendest** deine lokalen Commits an das Remote-Repository (z. B. GitHub). Damit werden deine Änderungen für andere sichtbar bzw. gesichert.
- **Pull** □□: Du **holst** die neuesten Änderungen vom Remote-Repository auf deinen Computer. Falls andere Personen Änderungen gepusht haben, bekommst du diese so.

```
flowchart LR
    A["Lokales Repository<br>auf deinem PC"] -->|Push| B["Remote Repository<br>z.B. auf GitHub"]
```

```
    B -->|Pull| A
```

Zusammenfassung auf einen Blick

Begriff	Kurzerklärung
Repository	Projektordner mit kompletter Änderungshistorie
Stage	Vorauswahl der Änderungen für den nächsten Commit
Commit	Gespeicherter Schnappschuss des Projektzustands
Branch	Paralleler Entwicklungszweig
Merge	Zusammenführen zweier Branches
Remote	Verbindung zu einem externen Repository
Clone	Komplettes Repository herunterladen
Push	Lokale Commits zum Remote hochladen
Pull	Änderungen vom Remote herunterladen

Mit diesen Begriffen im Hinterkopf wirst du die folgenden Kapitel deutlich leichter verstehen – sie bilden das **Vokabular**, mit dem Git „spricht".