PhpStorm für Git und GitHub konfigurieren

Dieses Kapitel integriert Git vollständig in PhpStorm. Du lernst die VCS-Einstellungen, Commit-Oberfläche, Diff-Ansichten, Branch-Menüs, integrierten Terminals und GitHub-Anmeldung kennen. Zudem erfährst du, wann die grafische IDE und wann die Kommandozeile geeigneter ist.

Git-Unterstützung in PhpStorm aktivieren

Ziel dieser Einrichtung

PhpStorm besitzt eine integrierte Unterstützung für Git. Sie ergänzt die Git-Kommandozeile um grafische Ansichten für Änderungen, Commits, Branches, Konflikte, Historie und Remotes.

Damit PhpStorm Git verwenden kann, müssen zwei Voraussetzungen erfüllt sein:

1. **Git for Windows ist installiert** und im Terminal verfügbar.
2. PhpStorm kennt den Pfad zur Git-Programmdatei und das gewünschte Projekt ist als Git-Repository eingerichtet.

> **Wichtig:** Git ist kein optionales PhpStorm-Plugin, das separat installiert werden muss. Die Versionskontrollintegration gehört standardmäßig zur IDE. PhpStorm nutzt jedoch eine lokale Git-Installation, um Git-Befehle auszuführen.

Voraussetzungen prüfen

Öffne vor der Einrichtung ein Terminal, beispielsweise **Git Bash**, **PowerShell** oder das integrierte Terminal von PhpStorm, und prüfe die Installation:

```
git --version
```

Eine erfolgreiche Ausgabe ähnelt diesem Beispiel:

```
git version 2.47.1.windows.1
```

Erscheint stattdessen eine Fehlermeldung wie `git is not recognized` oder `git: command not found`, ist Git entweder nicht installiert oder nicht in der `PATH`-Variable verfügbar. Richte in diesem Fall zunächst Git for Windows korrekt ein.

Prüfe außerdem, ob Git bereits eine Identität kennt:

```
git config --global user.name
git config --global user.email
```

Diese Werte sind nicht nötig, um Git in PhpStorm zu *aktivieren*, aber spätestens für den ersten Commit erforderlich.

Ein bestehendes Git-Repository in PhpStorm öffnen

Wenn dein Projekt bereits ein `.git`-Verzeichnis enthält, erkennt PhpStorm das Repository meistens automatisch.

1. Starte PhpStorm.
2. Wähle **File → Open**.
3. Öffne den **Stammordner** des Projekts — also den Ordner, in dem sich das versteckte Verzeichnis `.git` befindet.
4. Bestätige bei Bedarf, dass das Projekt in einem neuen Fenster oder im aktuellen Fenster geöffnet werden soll.

Nach dem Öffnen sollte PhpStorm die Git-Integration aktivieren. Typische Hinweise dafür sind:

- Der Projektname oder ein Branch-Name erscheint in der Statusleiste.
- Das Menü **Git** ist verfügbar.
- Im Projektfenster werden geänderte Dateien farblich markiert.
- Das Werkzeugfenster **Commit** kann über die linke oder untere Werkzeugleiste geöffnet werden.
- Unter **Git → Show Git Log** ist die Historie erreichbar.

Falls diese Elemente nicht erscheinen, prüfe die folgenden Schritte manuell.

Ein neues Projekt unter Versionskontrolle stellen

Ein lokaler Projektordner ohne `.git`-Verzeichnis ist zunächst noch kein Git-Repository. Du kannst Git direkt aus PhpStorm aktivieren.

1. Öffne das gewünschte Projekt in PhpStorm.
2. Wähle im Hauptmenü **VCS → Enable Version Control Integration**.
3. Wähle im Dialog **Git** aus.
4. Bestätige mit **OK**.

PhpStorm führt im Hintergrund im Wesentlichen diesen Befehl aus:

```
git init
```

Dadurch entsteht im Projektstamm ein verborgenes Verzeichnis namens `.git`. Es enthält die komplette lokale Git-Datenbank: Historie, Branch-Referenzen, Konfiguration, Index und weitere Metadaten.

Ergebnis kontrollieren

Öffne das integrierte Terminal über **View → Tool Windows → Terminal** und führe aus:

```
git status
```

Bei einem neuen Repository sieht die Ausgabe ungefähr so aus:

```
On branch main

No commits yet

nothing to commit
```

Der Name des Anfangs-Branches kann je nach Git-Konfiguration auch `master` oder anders lauten. Für neue Projekte ist `main` heute die übliche Konvention.

Git-Programm in PhpStorm konfigurieren

PhpStorm muss wissen, welche Git-Programmdatei es aufrufen soll. Öffne dazu:

File → Settings → Version Control → Git

Unter macOS wäre der entsprechende Pfad **PhpStorm → Settings → Version Control → Git**. In diesem Kurs liegt der Fokus jedoch auf Windows 11.

Im Feld **Path to Git executable** sollte normalerweise automatisch ein gültiger Pfad eingetragen sein, beispielsweise:

```
C:\Program Files\Git\bin\git.exe
```

oder:

```
C:\Program Files\Git\cmd\git.exe
```

Wähle anschließend **Test**.

Bei erfolgreicher Erkennung meldet PhpStorm etwa:

```
Git executed successfully
```

Zusätzlich wird die gefundene Git-Version angezeigt.

Welcher Git-Pfad ist korrekt?

Unter Windows existieren häufig mehrere ausführbare Git-Dateien. Für PhpStorm ist in der Regel einer dieser Pfade geeignet:

```
C:\Program Files\Git\bin\git.exe
```

```
C:\Program Files\Git\cmd\git.exe
```

Wenn PhpStorm Git automatisch gefunden hat und der Test erfolgreich ist, solltest du den Pfad nicht unnötig ändern.

> **Empfehlung:** Verwende dieselbe Git-for-Windows-Installation in PhpStorm, Git Bash und PowerShell. Unterschiedliche Git-Installationen können abweichende Konfigurationen, Credential Helper oder SSH-Einstellungen verwenden und später schwer nachvollziehbare Unterschiede verursachen.

Versionskontrollzuordnung des Projektordners prüfen

PhpStorm verwaltet, welche Ordner eines Projekts welchem Versionskontrollsystem zugeordnet sind. Das ist besonders relevant bei komplexen Projekten, Monorepos oder verschachtelten Repositories.

Öffne:

File → Settings → Version Control → Directory Mappings

Dort sollte eine Zuordnung sichtbar sein:

Verzeichnis	VCS
Projektstamm, beispielsweise `C:\Projekte\mein-projekt`	Git

Fehlt die Zuordnung:

1. Klicke auf **+**.
2. Wähle den Projektstamm oder den Ordner mit dem `.git`-Verzeichnis.
3. Wähle als VCS **Git**.
4. Übernimm die Einstellungen mit **Apply** und **OK**.

Nicht versehentlich einen Unterordner zuordnen

Die Zuordnung sollte normalerweise auf den Git-Projektstamm zeigen:

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

Würdest du nur `src/` zuordnen, obwohl `.git/` im übergeordneten Projektordner liegt, können Historie, Änderungen und Branches unvollständig oder gar nicht angezeigt werden.

Die Git-Integration im Alltag erkennen

Nach erfolgreicher Aktivierung arbeitet PhpStorm kontinuierlich mit dem lokalen Git-Repository. Du musst Git nicht vor jeder Änderung erneut einschalten.

Dateistatus im Projektfenster

PhpStorm markiert Dateien abhängig von ihrem Git-Zustand. Die genaue Farbgebung hängt vom Farbschema ab, typische Bedeutungen sind jedoch:

- **neu oder unversioniert:** Die Datei wurde noch nicht zu Git hinzugefügt.
- **geändert:** Eine bereits verfolgte Datei wurde verändert.
- **hinzugefügt:** Die Datei ist für den nächsten Commit vorgemerkt.
- **gelöscht:** Eine verfolgte Datei wurde entfernt.
- **ignoriert:** Eine Regel aus `.gitignore` schließt die Datei aus.

Die Farben sind nur eine visuelle Hilfe. Die zuverlässige Quelle bleibt immer der Git-Zustand, den du auch mit folgendem Befehl prüfen kannst:

```
git status
```

Branch-Anzeige

In der Statusleiste zeigt PhpStorm üblicherweise den aktuell ausgecheckten Branch an, etwa:

```
main
```

Ein Klick auf den Branch-Namen öffnet das Branch-Menü. Dort kannst du später Branches erstellen, wechseln, mergen, rebasen oder Remote-Branches ansehen.

Commit-Werkzeugfenster

Das Werkzeugfenster **Commit** sammelt lokale Änderungen und erlaubt unter anderem:

- Dateien für einen Commit auszuwählen,
- Diffs vor dem Commit zu prüfen,
- Commit-Nachrichten zu schreiben,

- Prüfungen auszuführen,
- Änderungen direkt zu committen oder zu committen und zu pushen.

Die IDE ersetzt dabei nicht das Git-Modell. Sie stellt lediglich eine Oberfläche für dieselben grundlegenden Git-Operationen bereit.

Aktivierung mit der Kommandozeile vergleichen

Die zentrale Aktion lässt sich sowohl in PhpStorm als auch im Terminal ausführen:

Aufgabe	PhpStorm	Kommandozeile
Neues Git-Repository erstellen	**VCS → Enable Version Control Integration → Git**	`git init`
Repository-Zustand prüfen	Commit-Fenster oder Git-Ansichten	`git status`
Branch anzeigen	Statusleiste oder Branch-Menü	`git branch --show-current`
Änderungen ansehen	Commit-Fenster oder Diff-Ansicht	`git diff`
Git-Historie öffnen	**Git → Show Git Log**	`git log`

Diese Zuordnung ist wichtig: Wenn du verstehst, welche Git-Operation PhpStorm ausführt, kannst du bei Problemen gezielt prüfen und bist nicht von der Benutzeroberfläche abhängig.

Aktivierung testen

Lege zum Test eine Datei im Projektstamm an, beispielsweise `README.md`, mit folgendem Inhalt:

```
# Mein Git-Übungsprojekt
```

Speichere die Datei. PhpStorm sollte sie nun als **unversionierte Datei** erkennen.

Prüfe anschließend im Terminal:

```
git status
```

Die Ausgabe sollte einen ähnlichen Abschnitt enthalten:

```
Untracked files:
  README.md
```

Füge die Datei testweise über PhpStorm zu Git hinzu:

1. Öffne das Werkzeugfenster **Commit**.
2. Aktiviere das Kontrollkästchen neben `README.md`.
3. Gib eine Commit-Nachricht ein, beispielsweise:

```
docs: README hinzufügen
```

4. Wähle **Commit**.

Alternativ erledigst du exakt denselben Ablauf im Terminal:

```
git add README.md
git commit -m "docs: README hinzufügen"
```

Zeige danach die Historie an:

```
git log --oneline
```

Wenn der Commit erscheint, ist Git sowohl lokal als auch in PhpStorm korrekt aktiv.

Häufige Probleme und ihre Ursachen

PhpStorm findet Git nicht

Symptom: Der Test im Bereich **Settings → Version Control → Git** schlägt fehl.

Typische Ursachen:

- Git for Windows ist nicht installiert.
- Der hinterlegte Pfad zeigt auf eine nicht vorhandene Datei.
- Eine alte Git-Installation wurde entfernt, aber PhpStorm verwendet noch den alten Pfad.
- Die Datei `git.exe` ist durch Sicherheitssoftware blockiert.

Lösung: Suche `git.exe` im Installationsordner von Git for Windows, trage einen gültigen Pfad ein und führe erneut **Test** aus.

Das Menü „Git“ fehlt oder zeigt keine Aktionen

Symptom: Das Projekt wird nicht als Git-Repository behandelt.

Typische Ursachen:

- Das Projekt enthält noch kein `.git`-Verzeichnis.
- Der falsche Ordner wurde geöffnet.
- Unter **Directory Mappings** ist keine Git-Zuordnung vorhanden.
- Das Repository liegt in einem übergeordneten Ordner, der nicht Teil des geöffneten Projekts ist.

Lösung: Öffne den tatsächlichen Repository-Stamm oder richte über **VCS → Enable Version Control Integration** ein neues Repository ein. Kontrolliere danach die Directory Mappings.

Änderungen werden nicht angezeigt

Symptom: Eine gespeicherte Datei erscheint nicht im Commit-Fenster.

Prüfe zuerst im Terminal:

```
git status
```

Mögliche Ursachen sind:

- Die Datei wird von `.gitignore` ausgeschlossen.
- Die Datei liegt außerhalb des Repository-Stamms.
- Die Datei wurde noch nicht gespeichert.
- PhpStorm ordnet das Verzeichnis nicht Git zu.
- Die Datei ist Teil einer anderen Changelist oder ein Filter blendet sie aus.

Ob eine Datei durch eine Ignore-Regel ausgeschlossen wird, kannst du präzise prüfen:

```
git check-ignore -v pfad\zur\datei
```

Git zeigt dann die konkrete Ignore-Datei und Regel an, die den Ausschluss verursacht.

Das Projekt enthält mehrere Git-Repositories

Ein Projekt kann mehrere unabhängige Repositories enthalten, etwa bei Modulen, Submodulen oder nebeneinanderliegenden Anwendungen:

```
arbeitsbereich/
├── backend/
│   └── .git/
└── frontend/
    └── .git/
```

In diesem Fall kann PhpStorm mehrere Directory Mappings verwalten. Jedes Repository erhält eine eigene Git-Zuordnung. Prüfe besonders sorgfältig, in welchem Repository du gerade commitest, Branches wechselst oder Remotes konfigurierst.

> **Achtung:** Ein verschachteltes `.git`-Verzeichnis ist nicht automatisch ein Submodule. Ein Submodule ist ein bewusst in Git eingetragener Verweis auf ein anderes Repository. Mehrere zufällig verschachtelte Repositories führen dagegen oft zu Verwirrung.

Sichere Ausgangskonfiguration

Nach diesem Abschnitt sollte dein Projekt diese Eigenschaften haben:

- Git for Windows ist installiert und über `git --version` erreichbar.
- PhpStorm kennt einen funktionierenden Pfad zu `git.exe`.
- Der Projektstamm ist Git zugeordnet.
- Ein `.git`-Verzeichnis existiert im Repository-Stamm.
- PhpStorm erkennt neue, geänderte und vorgemerkte Dateien.
- Der aktuelle Branch wird in der IDE angezeigt.
- `git status` im Terminal und die Anzeige in PhpStorm beschreiben denselben Zustand.

Damit ist die technische Grundlage gelegt. Als Nächstes prüfst du das Git-Programm und seine Version in PhpStorm genauer und richtest anschließend die Commit- und Historienwerkzeuge für den täglichen Workflow ein.

Git-Programm und Version prüfen

Bevor PhpStorm sinnvoll mit Git zusammenarbeiten kann, muss die IDE wissen, *welches* Git-Programm auf deinem System verwendet werden soll – und ob dessen Version aktuell genug ist. Gerade unter Windows 11, wo häufig mehrere Git-Installationen parallel existieren (Git for Windows, WSL-Git, in PhpStorm gebündelte Tools), ist diese Prüfung kein optionaler Schritt, sondern die Grundlage für alle weiteren Kapitel.

Warum die Version überhaupt eine Rolle spielt

Git entwickelt sich kontinuierlich weiter, und einige Befehle, die in diesem Kurs verwendet werden, setzen bestimmte Mindestversionen voraus. Läuft im Hintergrund eine veraltete Git-Version, meldet PhpStorm mitunter kryptische Fehler oder blendet Funktionen im Menü einfach aus, ohne dies deutlich zu kommentieren.

Funktion	Benötigte Git-Version (mindestens)
`git switch` / `git restore`	2.23
`--force-with-lease` als Standardverhalten stabil	2.30+
Sparse Checkout im Cone-Modus	2.25
Partielle Klone (`--filter`)	2.19
Bessere Rebase-Autostash-Optionen	2.9+

Empfehlung: Verwende durchgängig eine aktuelle Version aus der 2.4x-Reihe oder neuer, damit sämtliche in diesem Buch beschriebenen Befehle und PhpStorm-Funktionen zuverlässig zur Verfügung stehen.

Die Git-Version über die Kommandozeile prüfen

Der schnellste und zuverlässigste Weg führt unabhängig von der IDE über ein Terminal:

```
git --version
```

Die Ausgabe sollte in etwa so aussehen:

```
git version 2.46.0.windows.1
```

Der Zusatz `windows` zeigt, dass tatsächlich *Git for Windows* aufgerufen wurde – und nicht etwa eine über WSL installierte Version oder ein Relikt einer alten Installation. Solltest du stattdessen eine deutlich ältere Version oder eine Fehlermeldung wie `git is not recognized` erhalten, liegt entweder ein PATH-Problem vor (siehe Kapitel 2) oder Git wurde noch nicht installiert.

Den Git-Pfad in PhpStorm kontrollieren

PhpStorm besitzt eine eigene Einstellung, über die festgelegt wird, welche Git-Executable die IDE tatsächlich verwendet. Diese kann von der Version abweichen, die in deiner Kommandozeile aktiv ist – etwa wenn mehrere Installationen vorhanden sind.

1. Öffne **Datei → Einstellungen** (bzw. **File → Settings** bei englischer Oberfläche).
2. Navigiere zu **Version Control → Git**.
3. Prüfe das Feld **Path to Git executable**.

Im Normalfall trägt PhpStorm hier automatisch den Pfad zur global installierten Git-Version ein, üblicherweise:

```
C:\Program Files\Git\cmd\git.exe
```

Die Test-Schaltfläche nutzen

Neben dem Pfadfeld befindet sich die Schaltfläche **Test**. Ein Klick darauf führt intern denselben Befehl aus wie `git --version` auf der Kommandozeile und zeigt das Ergebnis direkt in einem Dialogfenster an:

```
Git version 2.46.0 detected
```

Erscheint stattdessen eine Fehlermeldung, deutet dies meist auf eines der folgenden Probleme hin:

- **Falscher oder veralteter Pfad** – Git wurde neu installiert oder verschoben, PhpStorm zeigt aber noch auf den alten Speicherort.
- **Beschädigte Installation** – die `.exe`-Datei existiert, ist aber nicht ausführbar oder unvollständig.
- **Mehrere konkurrierende Installationen** – etwa eine ältere Version aus einem Paketmanager, die zufällig zuerst im Pfad gefunden wurde.

Den Pfad manuell festlegen

Sollte die automatische Erkennung fehlschlagen oder du bewusst eine andere Git-Installation verwenden wollen, lässt sich der Pfad manuell setzen:

1. Klicke im Feld **Path to Git executable** auf das Ordnersymbol.
2. Navigiere zum gewünschten `git.exe`, typischerweise unter `C:\Program Files\Git\cmd\`.
3. Bestätige die Auswahl und klicke erneut auf **Test**, um die Erkennung zu verifizieren.

Hinweis: Verwende möglichst den Pfad im `cmd`-Unterordner (`Git\cmd\git.exe`) und nicht den im `bin`-Unterordner, da Letzterer für die Nutzung innerhalb von Bash-Umgebungen optimiert ist und in Kombination mit PhpStorms interner Prozessausführung gelegentlich zu Inkonsistenzen führen kann.

Mehrere Git-Installationen unter Windows 11 erkennen

Es ist unter Windows keine Seltenheit, mehrere Git-Varianten gleichzeitig vorzufinden:

- **Git for Windows** – die im Kurs empfohlene, eigenständige Installation.
- **Git innerhalb von WSL** (Windows Subsystem for Linux) – separat installiert, mit eigenem Pfad und eigener Konfiguration.
- **Git aus Paketmanagern** wie Chocolatey oder Scoop – funktioniert meist zuverlässig, kann aber zu Versionskonflikten führen, wenn parallel weitere Installationen existieren.

PhpStorm sollte ausschließlich auf die für dieses Buch vorgesehene Git-for-Windows-Installation verweisen. Solltest du zusätzlich mit WSL arbeiten, achte darauf, Projekte nicht unbeabsichtigt zwischen beiden Welten zu vermischen, da sich Zeilenenden, Dateirechte und Pfadformate unterscheiden können.

Automatische Erkennung vs. explizite Konfiguration

Für die meisten Einzelplatz-Setups reicht die automatische Erkennung vollkommen aus. In Team- oder Firmenumgebungen empfiehlt sich dennoch, den Pfad einmal bewusst zu prüfen und zu dokumentieren – besonders dann, wenn:

- Projekte über mehrere Rechner mit unterschiedlichen Installationswegen hinweg geteilt werden,

- Continuous-Integration-Umgebungen (siehe Kapitel 20) eine bestimmte Git-Version voraussetzen,
- ältere Legacy-Systeme im Unternehmen noch mit veralteten Git-Versionen arbeiten.

Git aktualisieren

Wird eine veraltete Version erkannt, lässt sich Git for Windows unkompliziert aktualisieren:

```
git update-git-for-windows
```

Dieser Befehl prüft auf eine neuere Version und führt bei Bedarf den Installer automatisch aus. Alternativ kann die aktuelle Version jederzeit manuell von der offiziellen Git-for-Windows-Seite heruntergeladen und installiert werden – bestehende Konfigurationswerte (siehe Kapitel 2) bleiben dabei in der Regel erhalten.

Checkliste für diesen Schritt

Bevor du mit den nächsten Abschnitten dieses Kapitels fortfährst, sollten folgende Punkte erfüllt sein:

- `git --version` liefert in der Kommandozeile eine aktuelle Version (mindestens 2.4x).
- PhpStorm zeigt unter **Version Control → Git** denselben Pfad und dieselbe Version an.
- Die **Test**-Schaltfläche bestätigt die Erkennung ohne Fehlermeldung.
- Es ist klar, welche Git-Installation tatsächlich aktiv ist, falls mehrere auf dem System vorhanden sind.

Ist all dies sichergestellt, steht einer reibungsvollen Integration von Git in PhpStorm nichts mehr im Weg – und die folgenden Abschnitte zur Versionskontrolle eines Projekts können ohne Umwege beginnen.

Ein Projekt unter Versionskontrolle stellen

Bevor Commits, Branches oder Pull Requests überhaupt möglich sind, muss PhpStorm wissen, dass ein Projektverzeichnis von Git verwaltet werden soll. Technisch entspricht dieser Schritt exakt dem Kommandozeilenbefehl `git init`, wird in PhpStorm jedoch über einen geführten Dialog ausgeführt, der zusätzlich die *VCS-Zuordnung* (Directory Mapping) im Projekt hinterlegt. Diese Zuordnung ist der Grund, warum die IDE anschließend Dateifarben, Änderungslisten und das Commit-Fenster überhaupt anzeigen kann.

Zwei Ausgangssituationen

In der Praxis triffst du in PhpStorm auf zwei unterschiedliche Startpunkte:

1. **Neues, noch unversioniertes Projekt** – ein lokaler Ordner mit PHP-Dateien, aber ohne `.git`-Verzeichnis.
2. **Bereits vorhandenes Repository** – das Projekt wurde per `git clone` oder außerhalb von PhpStorm erstellt und muss nur noch korrekt zugeordnet werden.

Dieses Unterkapitel behandelt den ersten Fall ausführlich, da er den eigentlichen Einrichtungsschritt darstellt.

Schritt für Schritt: Version Control Integration aktivieren

1. Öffne das Projekt in PhpStorm.
2. Wähle im Menü **VCS → Enable Version Control Integration…**
3. Im erscheinenden Dialog wählst du als System **Git** aus.
4. Bestätige mit **OK**.

PhpStorm führt daraufhin im Hintergrund `git init` im Projektstammverzeichnis aus und erstellt das `.git`-Verzeichnis. Gleichzeitig trägt die IDE das Projektverzeichnis als *VCS Root* ein – sichtbar unter:

```
Settings/Preferences → Version Control → Directory Mappings
```

> **Praxistipp:** Ist das VCS-Menü nicht sichtbar, wurde entweder kein Projekt geöffnet oder es liegt bereits eine Zuordnung vor. In diesem Fall erscheint statt *Enable Version Control Integration* direkt das erweiterte VCS-Menü mit Commit-, Branch- und Log-Optionen.

Was sich im Projektbaum ändert

Unmittelbar nach der Aktivierung markiert PhpStorm alle Dateien im *Project*-Werkzeugfenster farblich:

Farbe	Bedeutung
Braun/Orange	Datei ist *unversioniert* (noch nicht zu Git hinzugefügt)
Grün	Datei ist neu hinzugefügt und für den nächsten Commit vorgemerkt
Blau	Datei ist bereits verfolgt und wurde geändert
Grau	Datei ist ignoriert (siehe `.gitignore`)

Da direkt nach `git init` noch nichts vorgemerkt ist, erscheinen zunächst *alle* Dateien braun. Dies entspricht dem Zustand, den `git status` auf der Kommandozeile als „Untracked files“ ausweist.

Dateien zur Versionskontrolle hinzufügen

PhpStorm bietet nach der Aktivierung häufig automatisch einen Hinweis an:

> *„The following files are not currently tracked by Git…“*

Über **Add** werden die ausgewählten Dateien in die Staging Area übernommen – identisch zu `git add`. Alternativ erreichst du dies jederzeit über:

- Rechtsklick auf Datei oder Ordner → **Git → Add**

- Tastenkürzel `Ctrl + Alt + A`

Vor diesem Schritt lohnt es sich, kurz zu prüfen, ob bereits eine `.gitignore`-Datei existiert, damit keine Abhängigkeiten wie `vendor/` oder IDE-Konfigurationsdateien versehentlich mit aufgenommen werden. Eine vertiefte Behandlung dieses Themas folgt im Kapitel *„Dateien ignorieren und Repository-Hygiene“*.

Der Ablauf im Überblick

```
flowchart LR
    A["Projektordner ohne .git"] --> B["VCS -> Enable Version Control Integration"]
    B --> C["git init im Hintergrund"]
    C --> D["Directory Mapping wird angelegt"]
    D --> E["Dateien erscheinen als unversioniert"]
    E --> F["Dateien ueber Add vormerken"]
    F --> G["Bereit fuer ersten Commit"]

    style A fill:#374151,color:#ffffff
    style G fill:#065f46,color:#ffffff
```

Directory Mappings verstehen

Gerade bei PHP-Projekten mit mehreren Modulen, einer separaten `docs`-Struktur oder eingebundenen Bibliotheken ist es wichtig zu wissen, *welches* Verzeichnis PhpStorm als Wurzel des Repositories betrachtet. Unter

```
Settings/Preferences → Version Control → Directory Mappings
```

siehst du die Zuordnung von Dateisystempfad zu VCS-System. In den allermeisten Fällen genügt ein einziger Eintrag mit dem Projektstammverzeichnis. Enthält dein Projekt jedoch mehrere unabhängige Repositories – etwa ein separates Repository für ein Composer-Paket innerhalb desselben PhpStorm-Projekts – kannst du hier zusätzliche Mappings ergänzen.

> **Warnung:** Verschachtelte `.git`-Verzeichnisse (ein Repository innerhalb eines anderen) führen ohne Submodule oder Subtree-Konfiguration zu Verwirrung, da Git das innere Repository ignoriert, PhpStorm es aber unter Umständen separat

> anzeigt. Vermeide dies, solange du die Konzepte aus Kapitel 27 noch nicht behandelt hast.

Vergleich: IDE-Aktion und Git-Befehl

PhpStorm-Aktion	Entsprechender Git-Befehl
VCS → Enable Version Control Integration	`git init`
Rechtsklick → Git → Add	`git add <Datei>`
Directory Mappings anzeigen	`git rev-parse --show-toplevel`
Farbliche Statusanzeige im Projektbaum	`git status`

Diese Gegenüberstellung ist bewusst Teil jedes praktischen Kapitels dieses Kurses: Sie stellt sicher, dass du auch dann souverän bleibst, wenn kein grafisches Werkzeug zur Verfügung steht – etwa bei der Arbeit über SSH auf einem Server.

Kontrolle des Ergebnisses

Nach erfolgreicher Aktivierung solltest du folgende Punkte überprüfen:

- Im Projektverzeichnis existiert ein `.git`-Ordner (sichtbar über den Datei-Explorer bei aktivierter Anzeige versteckter Dateien).
- Das VCS-Menü in PhpStorm zeigt nun vollständige Optionen wie **Commit**, **Push** und **Branches** an.
- Das *Commit*-Werkzeugfenster listet die vorgemerkten Dateien korrekt auf.

Damit ist das Fundament gelegt: Das Projekt befindet sich nun vollständig unter Versionskontrolle, und die IDE ist bereit, den ersten Commit entgegenzunehmen – der Gegenstand des folgenden Unterkapitels zur Konfiguration des Commit-Werkzeugfensters. □□

Commit-Werkzeugfenster konfigurieren

Das *Commit*-Werkzeugfenster ist die zentrale Anlaufstelle für alle Änderungen, die du in ein Repository einpflegen möchtest. Es zeigt dir auf einen Blick, welche Dateien bearbeitet, neu erstellt oder gelöscht wurden, und erlaubt dir, diese gezielt zu gruppieren, zu prüfen und mit einer aussagekräftigen Nachricht zu versehen, bevor sie als Commit in die Historie wandern.

Du öffnest es über:

```
View → Tool Windows → Commit
```

oder mit dem Tastaturkürzel `Alt + 0`. Standardmäßig erscheint es links am Bildschirmrand, kann aber wie jedes andere Werkzeugfenster angedockt, verschoben oder als eigenständiges Fenster „abgerissen“ werden.

Modaler und nicht-modaler Commit

PhpStorm bietet zwei grundlegend unterschiedliche Arbeitsweisen für den Commit-Vorgang, die sich in ihrem Verhalten deutlich unterscheiden:

Modus	Verhalten	Empfehlung
Nicht-modal *(Standard seit neueren Versionen)*	Der Commit-Bereich ist dauerhaft im Werkzeugfenster sichtbar; du kannst währenddessen weiterarbeiten, Dateien wechseln und andere Fenster nutzen.	Für den Alltag empfehlenswert, da flüssiger und weniger blockierend.
Modal *(klassischer Commit-Dialog)*	Ein separates Dialogfenster öffnet sich, blockiert andere Aktionen und muss explizit bestätigt oder abgebrochen werden.	Sinnvoll, wenn du einen klaren, unterbrechungsfreien Prüfungsschritt bevorzugst.

Du steuerst dieses Verhalten unter:

```
Settings → Version Control → Commit
```

über die Option **„Use non-modal commit interface“**. Deaktivierst du sie, kehrst du zum klassischen, modalen Dialog zurück – eine Einstellung, die besonders Umsteiger von älteren PhpStorm-Versionen oder anderen IDEs oft bevorzugen.

Wichtige Grundeinstellungen

Unter `Settings → Version Control → Commit` findest du die zentralen Schalter, mit denen sich der Commit-Vorgang an deinen Arbeitsstil anpassen lässt:

- **„Show unversioned files“** – zeigt nicht verfolgte Dateien direkt in der Commit-Liste an, statt sie zu verstecken.
- **„Before Commit“-Checks** – eine Reihe automatischer Prüfungen, die *vor* jedem Commit ausgeführt werden können (siehe nächster Abschnitt).
- **„Clear initial commit message“** – legt fest, ob das Nachrichtenfeld bei jedem neuen Commit leer beginnt oder die letzte Eingabe behält.
- **„Move Focus to the Commit Message Field“** – springt der Cursor nach dem Öffnen automatisch in das Nachrichtenfeld, was den Arbeitsfluss beschleunigt.

Diese Optionen wirken unscheinbar, summieren sich aber über Hunderte von Commits zu einem spürbaren Unterschied in der täglichen Effizienz.

Automatische Prüfungen vor dem Commit

Eine der nützlichsten Funktionen des Commit-Werkzeugfensters sind die **„Before Commit“-Prüfungen**. Sie laufen automatisch ab, sobald du auf *Commit* klickst, und verhindern, dass unsaubere oder fehlerhafte Änderungen versehentlich in die Historie gelangen.

Zu den wichtigsten gehören:

- **Reformat code** – wendet den konfigurierten Code-Style automatisch auf die geänderten Dateien an.
- **Optimize imports** – entfernt ungenutzte und sortiert vorhandene Imports, besonders relevant in PHP-Klassen mit vielen `use`-Anweisungen.
- **Rearrange code** – ordnet Klassenmitglieder gemäß den definierten Regeln neu an.

- **Perform code analysis** – führt eine Inspektion durch und warnt vor Fehlern, bevor sie festgeschrieben werden.
- **Check TODO** – weist auf offene `TODO`-Kommentare in den betroffenen Dateien hin.

> *Praxistipp:* Aktiviere zu Beginn nicht alle Prüfungen gleichzeitig. Beginne mit „Reformat code“ und „Optimize imports“, um ein Gefühl für die automatischen Eingriffe zu entwickeln, bevor du strengere Analysen hinzufügst.

Jede Prüfung lässt sich einzeln ein- oder ausschalten und kann so konfiguriert werden, dass sie bei Verstößen den Commit blockiert oder lediglich eine Warnung anzeigt.

Commit-Nachrichten strukturieren

Direkt im Werkzeugfenster befindet sich das Nachrichtenfeld für die Commit-Beschreibung. PhpStorm unterstützt dich dabei auf mehreren Ebenen:

- **Vorlagen (Commit Message Templates):** Unter `Settings → Version Control → Commit Message` kannst du eine feste Vorlage hinterlegen, etwa im Sinne der *Conventional Commits*:

```
<type>(<scope>): <subject>


<body>
```

- **Zeilenlängen-Hinweis:** Eine vertikale Linie im Eingabefeld markiert die empfohlene maximale Zeilenlänge der ersten Zeile (üblicherweise 72 Zeichen), was der gängigen Git-Konvention entspricht.
- **Verlauf früherer Nachrichten:** Über das Uhrsymbol im Nachrichtenfeld rufst du zuvor verwendete Commit-Nachrichten ab – praktisch bei wiederkehrenden Formulierungen innerhalb eines Features.

Changelists gezielt nutzen

Das Commit-Werkzeugfenster gruppiert Änderungen standardmäßig in der *Default*-Changelist. Du kannst jedoch beliebig viele weitere Changelists anlegen, um thematisch getrennte Änderungen parallel vorzubereiten, ohne sie sofort zu committen.

- Über das Kontextmenü einer Datei wählst du **„Move to Another Changelist“**, um sie einer neuen Gruppe zuzuweisen.
- Jede Changelist besitzt einen eigenen Namen und optional einen Kommentar, was besonders hilfreich ist, wenn du an mehreren logisch getrennten Themen gleichzeitig arbeitest.
- Beim Commit wählst du gezielt aus, welche Changelist eingecheckt werden soll – die übrigen bleiben unberührt im Arbeitsverzeichnis erhalten.

Diese Funktion ersetzt in vielen Alltagssituationen das manuelle Stashen und erlaubt eine deutlich übersichtlichere Arbeitsweise, wenn mehrere kleine Aufgaben nebeneinander existieren.

Diff-Vorschau direkt im Fenster

Ein oft übersehenes, aber äußerst hilfreiches Detail ist die integrierte Diff-Vorschau. Aktivierst du unter `Settings → Version Control → Commit` die Option **„Show diff preview on double click“**, öffnest du mit einem Doppelklick auf eine Datei sofort die Änderungsansicht, ohne das Commit-Fenster verlassen zu müssen.

Alternativ lässt sich die Vorschau dauerhaft als eingebettetes Panel neben der Dateiliste anzeigen – so behältst du beim Formulieren der Commit-Nachricht stets den inhaltlichen Kontext im Blick, was die Qualität deiner Beschreibungen merklich verbessert.

Zusammenfassung

Das Commit-Werkzeugfenster ist weit mehr als ein einfacher „Speichern“-Knopf. Durch die Kombination aus **automatischen Prüfungen**, **strukturierten Nachrichtenvorlagen**, **flexiblen Changelists** und **integrierter Diff-Ansicht** wird es zu einem Werkzeug, das Disziplin und Geschwindigkeit gleichzeitig fördert. Wer diese Einstellungen bewusst an den eigenen Arbeitsstil anpasst, reduziert Flüchtigkeitsfehler erheblich und legt damit den Grundstein für eine saubere, nachvollziehbare Projekthistorie – ein Thema, das im weiteren Verlauf des Kurses noch vertieft wird.

Git- und Log-Werkzeugfenster erkunden

Das Git-Werkzeugfenster als Kommandozentrale

Während das Commit-Werkzeugfenster den *Weg in* die Historie beschreibt, ist das **Git-Werkzeugfenster** dein Blick *auf* die Historie. Es fasst alles zusammen, was mit dem aktuellen Zustand deines Repositorys zu tun hat: Branches, Commits, lokale Änderungen und die tatsächlich ausgeführten Git-Befehle. Du öffnest es über `Alt` + `9` oder über *View → Tool Windows → Git*.

Das Fenster ist in mehrere Reiter unterteilt, die je nach PhpStorm-Version leicht unterschiedlich benannt sind, inhaltlich aber stets dieselben drei Aufgaben abbilden:

Reiter	Aufgabe
Log	Commit-Graph, Branches, Tags, Details zu einzelnen Commits
Local Changes	Aktueller Stand von Arbeitsverzeichnis und Index (siehe Commit-Werkzeugfenster)
Console	Protokoll aller intern ausgeführten Git-Kommandozeilenbefehle

Der Log-Tab im Detail

Der **Log-Tab** ist das Herzstück des Fensters. Er visualisiert die Commit-Historie als Graph – vergleichbar mit `git log --graph --oneline --all`, nur interaktiv und deutlich übersichtlicher.

Der Graphbereich zeigt links die Commit-Linien mit farblich getrennten Branches. Jeder Knotenpunkt ist ein Commit; Verzweigungen und Zusammenführungen sind sofort erkennbar. Rechts daneben stehen Commit-Nachricht, Autor und Zeitpunkt.

Das Detailpanel öffnet sich, sobald du einen Commit anklickst. Es zeigt:

- die vollständige Commit-Nachricht,
- Autor, Committer und Zeitstempel,
- die betroffenen Dateien inklusive Diff-Vorschau per Doppelklick.

Die Branch-Spalte listet, welche lokalen und entfernten Branches auf den jeweiligen Commit zeigen – praktisch, um schnell zu erkennen, ob eine Änderung bereits gepusht wurde.

Filtern und Suchen

Über der Graphansicht befindet sich eine Filterleiste, mit der du die angezeigte Historie gezielt einschränkst:

- **Branch-Filter**: nur bestimmte Branches oder *alle* anzeigen.
- **Benutzer-Filter**: Commits eines bestimmten Autors isolieren.
- **Datumsfilter**: Zeiträume eingrenzen.
- **Textsuche**: Commit-Nachrichten oder – über die Option „Search in commit contents“ – auch geänderte Dateiinhalte durchsuchen.

Diese Filter lassen sich kombinieren, was besonders bei größeren Projekten mit vielen parallelen Branches wertvoll ist, um sich schnell einen Überblick über den Fortschritt eines Features zu verschaffen.

Kontextaktionen direkt im Graph

Ein Rechtsklick auf einen Commit öffnet ein Kontextmenü mit den wichtigsten Operationen, die du sonst über die Kommandozeile ausführen würdest:

- `Checkout Revision` – entspricht `git checkout <commit>`,
- `New Branch from Selected Commit`,
- `Cherry-Pick`,
- `Revert Commit`,
- `Reset Current Branch to Here`.

So lassen sich viele fortgeschrittene Git-Operationen ausführen, ohne den Befehl im Detail zu kennen – wichtig ist aber, dass du in späteren Kapiteln verstehst, *was* im Hintergrund tatsächlich passiert, um Fehlbedienungen zu vermeiden.

Der Console-Tab: Transparenz statt Blackbox

Ein zentraler Vorteil von PhpStorm gegenüber rein grafischen Git-Clients ist der **Console-Tab**. Hier protokolliert die IDE jeden intern ausgeführten Git-Befehl im Klartext – inklusive Parametern und Rückgabewerten.

Dieser Reiter ist didaktisch besonders wertvoll: Klickst du beispielsweise in der Oberfläche auf *Pull*, kannst du in der Console exakt nachlesen, dass PhpStorm etwa

```
git fetch origin
git merge origin/main
```

ausgeführt hat. So verknüpfst du grafische Aktionen direkt mit dem zugrunde liegenden Kommandozeilenwissen, das dir in den folgenden Kapiteln immer wieder begegnet.

Zusammenspiel der Reiter

```
flowchart LR
    A["Log-Tab: Commit-Historie und Branches"] --> D["Git-Werkzeugfenster"]
    B["Local Changes: Arbeitsverzeichnis und Index"] --> D
    C["Console: Ausgeführte Git-Befehle"] --> D
    D --> E["Vollständiger Überblick über Repository-Zustand"]
```

Praktische Hinweise für den Alltag

- **Doppelklick auf einen Commit** öffnet direkt die Diff-Ansicht der geänderten Dateien – der schnellste Weg, eine fremde Änderung zu verstehen.
- **„Show Diff Preview on Double Click“** in den Einstellungen beschleunigt wiederkehrende Prüfungen.
- Nutze den Log-Tab regelmäßig *vor* riskanten Operationen wie Rebase oder Reset, um dir den aktuellen Branch-Zustand bewusst zu machen.
- Die **Console** solltest du dir angewöhnen zu beobachten, gerade in der Lernphase – sie ist dein Fenster zur eigentlichen Git-Mechanik hinter der Oberfläche.

Mit einem sicheren Umgang mit Log- und Console-Tab hast du die Grundlage geschaffen, um im nächsten Schritt Änderungen und Diffs gezielt zu untersuchen – ein Werkzeug, das dich durch den gesamten weiteren Kurs begleiten wird.

Änderungen und Diffs anzeigen

Änderungen sichtbar machen: Die Diff-Engine von PhpStorm

Bevor ein Commit entsteht, solltest du genau wissen, *was* sich geändert hat – nicht nur *dass* sich etwas geändert hat. PhpStorm bietet dafür eine der ausgereiftesten Diff-Darstellungen unter allen Git-Werkzeugen. Dieses Kapitel zeigt dir, wie du Änderungen auf Datei-, Zeilen- und Projektebene erkennst, interpretierst und gezielt nutzt.

Warum die visuelle Diff-Ansicht wichtiger ist als `git diff`

Auf der Kommandozeile liefert `git diff` textbasierte Unterschiede mit `+`- und `-`-Präfixen. Das funktioniert, ist aber bei komplexen Änderungen, umformatiertem Code oder verschobenen Codeblöcken schnell unübersichtlich. PhpStorm übersetzt diese Rohdaten in eine strukturierte, farblich kodierte Ansicht mit:

- **Syntax-Highlighting** – auch im Diff bleibt PHP-, JavaScript- oder Twig-Code lesbar.
- **Intelligenter Zeilen- und Wort-Erkennung** – nicht nur ganze Zeilen, sondern auch einzelne geänderte Wörter innerhalb einer Zeile werden markiert.
- **Navigierbaren Blöcken** – du kannst zwischen Änderungsblöcken springen, statt zu scrollen.

Damit wird der Diff nicht nur zur Kontrolle vor dem Commit genutzt, sondern zu einem echten Analysewerkzeug im Alltag.

Der Gutter: Änderungen direkt im Editor erkennen

Der wichtigste und unauffälligste Helfer ist der **Gutter** – der schmale Streifen links neben der Zeilennummer im Editor. Sobald eine Datei unter Versionskontrolle steht, markiert PhpStorm dort automatisch jede Abweichung gegenüber dem letzten Commit:

Markierung	Bedeutung
Grüner Balken	Neu hinzugefügte Zeile
Blauer Balken	Geänderte Zeile
Roter Pfeil	Gelöschte Zeile *(an der Stelle, an der sie fehlt)*

Ein Klick auf einen dieser Balken öffnet ein Kontextmenü mit drei zentralen Aktionen:

1. **Diff anzeigen** – öffnet die Detailansicht für genau diesen Änderungsblock.
2. **Rollback** – verwirft die Änderung dieser Zeile, ohne die gesamte Datei zurückzusetzen.
3. **Copy** – kopiert den ursprünglichen Inhalt zur Wiederverwendung.

Dieser Mechanismus ist besonders wertvoll, weil er *ohne* den Umweg über ein separates Fenster funktioniert – die Historie ist direkt im Arbeitskontext sichtbar.

Die Diff-Ansicht öffnen

Es gibt mehrere gleichwertige Wege, eine vollständige Diff-Ansicht aufzurufen:

- Im **Commit-Werkzeugfenster** einen doppelten Klick auf eine geänderte Datei.
- Im **Git-Werkzeugfenster** (Reiter *Log*) einen Commit auswählen und die betroffene Datei doppelklicken.
- Über das Kontextmenü einer Datei im Projektbaum: *Git* → *Show Diff*.
- Mit der Tastenkombination (\text{Strg} + \text{D}) bei markierter Datei.

PhpStorm öffnet daraufhin die **Zwei-Spalten-Ansicht**: links der letzte committete Stand, rechts dein aktueller Arbeitsstand. Änderungsblöcke sind farblich hinterlegt, und über kleine Pfeile zwischen den Spalten lassen sich einzelne Blöcke gezielt in die eine oder andere Richtung übernehmen.

Navigation innerhalb des Diffs

Für größere Dateien mit vielen Änderungen bietet die Diff-Ansicht eine eigene Navigationsleiste:

- **Nächster/vorheriger Unterschied** – über die Pfeiltasten am oberen Rand oder (\text{F7}) bzw. (\text{Umschalt} + \text{F7}).
- **Übersichtskarte** – am rechten Rand zeigt ein Miniatur-Streifen die Position aller Änderungen im Gesamtdokument, vergleichbar mit einer Landkarte.
- **Zeilenweiser Vergleich** – bei aktivierter Wort-Diff-Option werden nur die tatsächlich veränderten Zeichen hervorgehoben, nicht die ganze Zeile.

Diese Übersichtskarte ist besonders hilfreich, wenn eine Datei hunderte Zeilen umfasst, aber nur an wenigen Stellen tatsächlich bearbeitet wurde.

Diffs zwischen beliebigen Revisionen

Die Diff-Funktion ist nicht auf den Vergleich „aktuell gegen letzten Commit“ beschränkt. Im **Git-Log** kannst du:

- **zwei beliebige Commits markieren** (mit gedrückter (\text{Strg})-Taste) und über das Kontextmenü *Compare Versions* wählen,
- **einen Commit gegen den aktuellen Arbeitsstand** vergleichen,
- **eine einzelne Datei über mehrere Commits hinweg** verfolgen, indem du mit der rechten Maustaste auf die Datei im Log klickst und *Show Diff* wählst.

Damit lässt sich exakt nachvollziehen, wann und durch wen eine bestimmte Zeile eingeführt wurde – eine Vorstufe zur später behandelten `git blame`-Funktionalität.

Der Unterschied zwischen „Changes“ und „Diff“ verstehen

Ein häufiges Missverständnis betrifft die Begriffe *Changes* und *Diff*:

- **Changes** bezeichnet die *Liste* der veränderten Dateien – sichtbar im Commit-Werkzeugfenster.
- **Diff** bezeichnet den *inhaltlichen Vergleich* einer einzelnen Datei zwischen zwei Zuständen.

PhpStorm verknüpft beide Konzepte eng: Aus der Änderungsliste gelangst du per Doppelklick direkt in den passenden Diff, ohne den Kontext zu verlieren.

Lokale Historie als zusätzliches Netz

Unabhängig von Git führt PhpStorm eine eigene **Local History**, die jede gespeicherte Änderung protokolliert – auch solche, die nie committet wurden. Über *VCS → Local History → Show History* lässt sich der Zustand einer Datei zu jedem beliebigen Zeitpunkt einsehen und mit dem aktuellen Stand vergleichen.

> **Praktischer Hinweis:** Die lokale Historie ersetzt kein Commit, ist aber ein wertvolles Sicherheitsnetz für den Moment, bevor überhaupt ein `git add` erfolgt

> ist.

Zusammenfassung

Die Diff-Fähigkeiten von PhpStorm reichen von der unauffälligen Gutter-Markierung bis zur detaillierten Mehrfach-Revisionsansicht. Wer diese Werkzeuge beherrscht, muss nicht mehr raten, was sich seit dem letzten Commit verändert hat – jede Zeile, jedes Wort und jede Historie lässt sich präzise und visuell nachvollziehen. Im nächsten Schritt verbindest du dieses Wissen mit deinem **GitHub-Konto**, um Änderungen nicht nur lokal zu betrachten, sondern auch remote zu teilen. □□

GitHub-Konto mit PhpStorm verbinden

Damit PhpStorm Pull Requests anzeigen, Repositories veröffentlichen und Push- oder Fetch-Vorgänge gegenüber GitHub authentifizieren kann, benötigt die IDE eine eigene, von der reinen Git-Konfiguration unabhängige Verbindung zu deinem GitHub-Konto. Diese Verbindung ist mehr als nur eine Anmeldung: Sie stellt einen *Sicherheits-Token* bereit, über den PhpStorm im Namen deines Kontos handelt – etwa beim Erstellen von Repositories, beim Kommentieren von Pull Requests oder beim Anzeigen von Issues direkt im Editor.

Warum eine separate Verbindung notwendig ist

Git selbst kennt GitHub nicht. Für Git ist ein Remote-Repository lediglich eine URL, die per HTTPS oder SSH angesprochen wird. GitHub-spezifische Funktionen – etwa das *Pull-Request-Werkzeugfenster*, die Anzeige von Reviews oder das direkte Erstellen eines Repositories aus der IDE – benötigen jedoch die *GitHub-API*. Diese API erfordert eine eigenständige Authentifizierung, die PhpStorm getrennt von den lokalen Git-Zugangsdaten verwaltet.

> 🧠 **Merke:** Die Anmeldung bei GitHub in PhpStorm betrifft die *API-Integration*, nicht automatisch den Datenaustausch per `git push` oder `git pull`. Für Letzteres bleibt zusätzlich eine funktionierende HTTPS- oder SSH-Authentifizierung erforderlich, wie sie in einem späteren Kapitel vertieft wird.

Voraussetzungen

Bevor du die Verbindung herstellst, sollten folgende Punkte erfüllt sein:

- Ein bestehendes **GitHub-Konto** mit verifizierter E-Mail-Adresse.
- **Zwei-Faktor-Authentifizierung** ist empfehlenswert, aber für die Erstverbindung nicht zwingend.

- Ein aktueller **Browser**, da die Anmeldung meist über den systemeigenen Webbrowser erfolgt.
- Eine funktionierende **Internetverbindung** ohne blockierende Firewall-Regeln für `github.com` und `api.github.com`.

Die Verbindung einrichten

Schritt 1: Einstellungen öffnen

Navigiere zu:

```
Datei → Einstellungen → Version Control → GitHub
```

Alternativ erreichst du denselben Dialog über das Menü `Git → GitHub → Zu GitHub anmelden…`, sobald du dich innerhalb eines Projekts befindest.

Schritt 2: Anmeldemethode wählen

PhpStorm bietet in der Regel zwei gleichwertige Wege an:

Methode	Ablauf	Empfehlung
Log In via GitHub	Öffnet den Standardbrowser, du meldest dich dort an und autorisierst PhpStorm per OAuth	Für die meisten Anwender die einfachste und sicherste Variante
Log In with Token	Du erzeugst manuell einen *Personal Access Token* auf GitHub und fügst ihn in PhpStorm ein	Sinnvoll bei eingeschränkten Berechtigungen, Firmenrichtlinien oder Automatisierung

Der folgende Ablauf verdeutlicht den OAuth-basierten Anmeldeprozess:

```
sequenceDiagram
    participant P as PhpStorm
    participant B as Browser
    participant G as GitHub

    P->>B: Öffnet Autorisierungsseite
    B->>G: Login-Daten senden
    G->>B: Autorisierung bestätigen
    B->>G: Zugriff für PhpStorm erlauben
```

```
    G->>P: Token an PhpStorm übergeben
    P->>P: Token verschlüsselt speichern
```

Schritt 3: Anmeldung über den Browser abschließen

Bei der Browser-Variante geschieht Folgendes:

1. PhpStorm öffnet eine GitHub-Autorisierungsseite.
2. Du meldest dich – falls noch nicht geschehen – mit deinem GitHub-Konto an.
3. GitHub zeigt an, welche **Berechtigungen** (Scopes) PhpStorm anfordert, etwa Zugriff auf Repositories, Pull Requests und Benutzerinformationen.
4. Nach Bestätigung kehrst du automatisch zu PhpStorm zurück; das Konto erscheint in der Liste der verbundenen Konten.

Schritt 4: Anmeldung per Token als Alternative

Falls dein Unternehmen OAuth-Anmeldungen einschränkt oder du volle Kontrolle über die Berechtigungen behalten möchtest, erstellst du stattdessen einen Token direkt auf GitHub:

1. Auf GitHub: `Settings → Developer settings → Personal access tokens → Fine-grained tokens`.
2. Einen neuen Token mit einem aussagekräftigen Namen wie `PhpStorm – Arbeitsplatz` erzeugen.
3. **Ablaufdatum** und **Berechtigungsumfang** bewusst einschränken – etwa nur `repo` und `read:user`, statt vollständigen Kontozugriff zu gewähren.
4. Den Token einmalig kopieren, da er danach nicht erneut angezeigt wird.
5. In PhpStorm bei `Log In with Token` einfügen und bestätigen.

> ⚠ **Sicherheitshinweis:** Ein Personal Access Token ist funktional einem Passwort gleichwertig. Bewahre ihn niemals im Klartext in Projektdateien oder Commits auf. Kapitel 21 vertieft den sicheren Umgang mit Tokens und deren Rotation.

Mehrere GitHub-Konten verwalten

Gerade bei der Trennung von privaten und beruflichen Projekten ist es üblich, mehrere GitHub-Konten zu nutzen. PhpStorm unterstützt dies direkt:

- Im gleichen Einstellungsdialog kannst du über das **Plus-Symbol** ein weiteres Konto hinzufügen.
- Beim Veröffentlichen eines Repositories oder Erstellen eines Pull Requests fragt PhpStorm anschließend, **welches Konto** verwendet werden soll, sofern mehrere hinterlegt sind.
- Konten lassen sich jederzeit über das **Minus-Symbol** entfernen, ohne dass dies lokale Repository-Daten beeinflusst.

Die Verbindung überprüfen

Nach erfolgreicher Anmeldung solltest du folgende Punkte kontrollieren:

1. Im Menü `Git → GitHub` erscheinen nun Aktionen wie `Create Pull Request` oder `View Pull Requests`.
2. Im Werkzeugfenster **Pull Requests** lassen sich bestehende Anfragen des verbundenen Kontos laden.
3. Beim Veröffentlichen eines neuen, lokalen Projekts über `Git → GitHub → Share Project on GitHub` wird das verbundene Konto automatisch als Ziel vorgeschlagen.

Typische Probleme und ihre Lösung

Symptom	Wahrscheinliche Ursache	Lösung
Browserfenster öffnet sich, aber Login bleibt hängen	Firewall oder Proxy blockiert `github.com`	Netzwerkeinstellungen unter `Einstellungen → Appearance & Behavior → System Settings → HTTP Proxy` prüfen
Token wird abgelehnt	Fehlende Berechtigungen oder abgelaufener Token	Neuen Token mit korrektem Scope erzeugen
Pull-Request-Werkzeugfenster bleibt leer	Konto ohne Zugriff auf das jeweilige Repository verbunden	Richtiges Konto im Projekt auswählen oder Repository-Berechtigung prüfen
Anmeldung wiederholt sich bei jedem Start	Zugangsdaten werden nicht dauerhaft gespeichert	Systemeigenen *Credential Store* statt reinen Arbeitsspeicher-Cache in den Einstellungen aktivieren

Zusammenfassung

Die Verbindung zwischen PhpStorm und GitHub bildet die Grundlage für nahezu alle komfortablen GitHub-Funktionen innerhalb der IDE – von der Veröffentlichung neuer Repositories bis zur Verwaltung von Pull Requests. Ob du dich per Browser-Login oder mit einem fein abgestimmten Personal Access Token anmeldest, hängt von deinen Sicherheitsanforderungen und organisatorischen Vorgaben ab. Entscheidend ist, dass du die vergebenen Berechtigungen bewusst wählst und die Verbindung regelmäßig überprüfst, bevor du im nächsten Schritt das **integrierte Terminal** einrichtest, um GitHub-Funktionen und klassische Git-Kommandozeile flexibel zu kombinieren.

Integriertes Terminal einrichten

Warum ein eigenes Git-Terminal in PhpStorm sinnvoll ist

Die grafischen Werkzeuge von PhpStorm decken den überwiegenden Teil des täglichen Git-Workflows ab – Commit-Dialog, Diff-Ansicht, Branch-Menü. Doch bestimmte Aufgaben lassen sich nur oder deutlich effizienter über die Kommandozeile lösen: interaktives Rebasing, komplexe `git log`-Filter, Plumbing-Befehle oder das schnelle Ausprobieren eines Befehls, bevor er zur Gewohnheit wird. Ein korrekt konfiguriertes *integriertes Terminal* verbindet beide Welten, ohne dass du zwischen Fenstern wechseln musst.

Unter Windows 11 ist diese Konfiguration nicht trivial, da mehrere Shells parallel existieren – **Git Bash**, **PowerShell** und **CMD** – und jede ihre eigenen Stärken sowie Eigenheiten bei Zeilenenden, Pfaden und Zeichenkodierung mitbringt.

Terminal-Einstellungen öffnen

Die Konfiguration findest du unter:

```
Datei → Einstellungen → Werkzeuge → Terminal
```

Hier legst du fest, welche Shell standardmäßig gestartet wird, in welchem Verzeichnis das Terminal öffnet und wie es sich optisch sowie funktional verhält.

Die richtige Shell auswählen

Für die Arbeit mit Git unter Windows kommen drei Kandidaten in Frage. Die Wahl beeinflusst, welche Befehle nativ funktionieren und wie Pfade interpretiert werden.

Shell	Git-Integration	Besonderheiten	Empfehlung
Git Bash	Nativ, POSIX-kompatibel	Unix-Befehle wie `grep`, `sed`, `ls -la` verfügbar	*Standardwahl* für Git-Arbeit
PowerShell	Über Git for Windows nutzbar	Objektorientierte Pipeline, gute Skriptfähigkeiten	Für Automatisierung/Skripte
CMD	Funktional, aber eingeschränkt	Keine moderne Syntax, wenig Komfort	Nur bei Altlasten nötig

Für nahezu alle Aufgaben in diesem Kurs ist **Git Bash** die sinnvollste Voreinstellung, da sie exakt dem Verhalten entspricht, das auch auf Linux- und macOS-Systemen üblich ist – ein klarer Vorteil, wenn du Anleitungen aus der offiziellen Git-Dokumentation eins zu eins übernehmen möchtest.

```
flowchart TD
    A["Aufgabe waehlen"] --> B{"Reine Git Befehle?"}
    B -->|Ja| C["Git Bash verwenden"]
    B -->|Nein| D{"Windows Automatisierung?"}
    D -->|Ja| E["PowerShell verwenden"]
    D -->|Nein| F["CMD nur bei Bedarf"]

    style C fill:#2e7d32,color:#ffffff
    style E fill:#1565c0,color:#ffffff
    style F fill:#616161,color:#ffffff
```

Shell-Pfad korrekt hinterlegen

Damit PhpStorm Git Bash findet, muss der Pfad zur ausführbaren Datei exakt stimmen. Nach einer Standardinstallation von *Git for Windows* lautet er üblicherweise:

```
C:\Program Files\Git\bin\bash.exe
```

Trage diesen Pfad im Feld **Shell-Pfad** ein. Alternativ kannst du über die Schaltfläche mit dem Ordnersymbol direkt zur Datei navigieren, was Tippfehler vermeidet. Sollte PhpStorm die Shell nicht automatisch vorschlagen, ist dies meist ein Hinweis darauf, dass die Installation nicht in der PATH-Variable registriert wurde – ein Punkt, der bereits im vorherigen Kapitel geprüft wurde.

Optionale Startparameter kannst du im Feld *Shell-Pfad* direkt anhängen, etwa `--login -i`, um eine interaktive Login-Shell zu erzwingen und damit sicherzustellen, dass alle Umgebungsvariablen

aus der `.bashrc` geladen werden.

Startverzeichnis festlegen

Standardmäßig öffnet PhpStorm das Terminal im Stammverzeichnis des aktuellen Projekts. Das ist in der Regel die gewünschte Einstellung, da du sofort im richtigen Repository-Kontext arbeitest. Über die Option **Startverzeichnis** lässt sich dies bei Bedarf überschreiben – sinnvoll etwa bei Monorepo-Strukturen, in denen du meist in einem Unterordner arbeitest.

Darstellung und Bedienkomfort

Für die tägliche Arbeit lohnt sich ein Blick auf folgende Einstellungen:

- **Schriftart und -größe**: Eine Monospace-Schrift mit guter Lesbarkeit für Zeichen wie `|`, `~` und `^`, die in Git-Befehlen häufig vorkommen.
- **Farbschema**: PhpStorm übernimmt standardmäßig das Editor-Farbschema; für Terminals ist oft ein dunkleres, kontrastreicheres Schema angenehmer.
- **Zeilenumbruch und Verlauf**: Die Anzahl der im Puffer gespeicherten Zeilen sollte großzügig bemessen sein, damit lange `git log`-Ausgaben nicht vorzeitig verworfen werden.
- **Kopieren bei Auswahl**: Diese Option beschleunigt das Übertragen von Commit-Hashes oder Branch-Namen erheblich, da kein explizites `Strg+C` nötig ist.

Mehrere Terminals und Split-Ansicht

PhpStorm erlaubt beliebig viele Terminal-Tabs sowie eine horizontale oder vertikale Aufteilung innerhalb eines Tabs. In der Praxis hat sich folgende Aufteilung bewährt:

1. Ein Tab für allgemeine Git-Befehle (`status`, `log`, `diff`).
2. Ein Tab für laufende Prozesse wie `php artisan serve` oder Testläufe.
3. Bei Bedarf ein dritter Tab für Composer- oder npm-Befehle.

Diese Trennung verhindert, dass lange laufende Prozesse den Blick auf Git-Ausgaben verdecken.

Zusammenspiel mit der Windows-Konfiguration

Da das integrierte Terminal letztlich dieselbe Git-Installation nutzt wie die eigenständige Git Bash aus Kapitel 2, gelten alle dort getroffenen Einstellungen unverändert weiter – insbesondere:

- `core.autocrlf` für die Behandlung von Zeilenenden,
- der konfigurierte Standard-Editor,
- der Credential Manager für die Authentifizierung.

Ein häufiger Stolperstein: Wird PhpStorm mit einem anderen Benutzerkontext oder über eine Verknüpfung mit abweichenden Umgebungsvariablen gestartet, kann das integrierte Terminal eine andere `PATH`-Variable sehen als eine manuell gestartete Git Bash. Im Zweifel hilft ein einfacher Test:

```
which git
git --version
echo $PATH
```

Weichen die Ausgaben von jenen ab, die du außerhalb von PhpStorm erhältst, liegt meist ein Konfigurationsproblem der Shell-Startdateien (`.bashrc`, `.bash_profile`) vor.

Eigene Anpassungen dauerhaft einbinden

Wer regelmäßig mit Git-Aliasen oder Prompt-Anpassungen arbeitet, sollte diese nicht in PhpStorm selbst, sondern in der jeweiligen Shell-Konfigurationsdatei hinterlegen, damit sie unabhängig vom verwendeten Werkzeug greifen. Für Git Bash ist dies typischerweise die Datei:

```
~/.bashrc
```

Ein Beispiel für eine sinnvolle Ergänzung, die den aktuellen Branch direkt im Prompt anzeigt:

```
parse_git_branch() {
    git branch 2>/dev/null | sed -n '/\* /s///p'
}
export PS1='\u@\h \W$(parse_git_branch) \$ '
```

Nach dem Speichern lädst du die Datei im Terminal neu:

```
source ~/.bashrc
```

Diese Anpassung ist besonders wertvoll, weil sie *unabhängig* von PhpStorm funktioniert – ein Vorteil, sobald du auch außerhalb der IDE arbeitest.

Terminal versus grafische Werkzeuge

Das integrierte Terminal ersetzt die GUI nicht, sondern ergänzt sie gezielt. Eine grobe Orientierung, wann welches Werkzeug vorzuziehen ist:

Aufgabe	Empfohlenes Werkzeug
Einzelne Dateien stagen, Commit schreiben	*PhpStorm-Commit-Fenster*
Diffs visuell vergleichen	*PhpStorm-Diff-Ansicht*
Interaktiver Rebase mit vielen Commits	*Terminal*
Schnelle Statusprüfung zwischendurch	*Terminal* (`git status`)
Merge-Konflikte mit Drei-Wege-Ansicht	*PhpStorm-Merge-Werkzeug*
Plumbing- oder Debug-Befehle	*Terminal*

Diese Faustregel wird dich durch den gesamten weiteren Kurs begleiten: Immer dort, wo Präzision und Geschwindigkeit der Tastatur gefragt sind, greifst du zum Terminal; immer dort, wo visuelle Übersicht zählt, nutzt du die grafischen Werkzeuge von PhpStorm.

Kurzcheck zur Konfiguration

Bevor du mit dem nächsten Kapitel fortfährst, sollten folgende Punkte erfüllt sein:

- ☐ Git Bash ist als Standard-Shell im integrierten Terminal hinterlegt.
- ☐ `git --version` liefert im Terminal dieselbe Ausgabe wie außerhalb von PhpStorm.
- ☐ Das Startverzeichnis entspricht dem Projektstamm.
- ☐ Farbschema und Schriftgröße sind angenehm lesbar eingestellt.
- ☐ Eigene `.bashrc`-Anpassungen werden korrekt geladen.

Damit steht dir ein vollwertiges, produktiv nutzbares Terminal zur Verfügung, das nahtlos in den weiteren Kurs übergeht – etwa bei der Arbeit mit *Changelists* im nächsten Abschnitt.

Changelists und Änderungsgruppen verwenden

Die Idee hinter Changelists

Git selbst kennt nur *einen* Index – die Staging Area, in der alle vorgemerkten Änderungen unabhängig von ihrem thematischen Zusammenhang landen. PhpStorm erweitert dieses Modell um eine eigene, rein lokale Organisationsebene: **Changelists**. Sie gruppieren geänderte, hinzugefügte oder gelöschte Dateien nach thematischer Zugehörigkeit, *bevor* sie überhaupt zu einem Commit werden.

Das ist besonders dann wertvoll, wenn du an mehreren Aufgaben gleichzeitig arbeitest, ohne für jede sofort einen eigenen Branch anzulegen – etwa ein kleiner Bugfix „nebenbei“, während du an einem größeren Feature arbeitest. Changelists sorgen dafür, dass du diese Änderungen getrennt betrachten, einzeln committen und nicht versehentlich miteinander vermischen kannst.

> **Wichtig:** Changelists sind eine reine PhpStorm-Funktion. Sie werden nicht in Git gespeichert und existieren nicht auf der Kommandozeile. Ein Kollege, der dasselbe Repository ausschließlich mit `git` bearbeitet, sieht davon nichts.

Die Standard-Changelist „Default“

Nach dem Öffnen eines Projekts legt PhpStorm automatisch eine Changelist mit dem Namen *Default* an. Alle neuen und geänderten Dateien landen zunächst dort, sofern du nichts anderes konfigurierst. Für einfache, lineare Arbeit reicht diese eine Liste völlig aus – erst bei paralleler oder verschachtelter Arbeit lohnt sich eine bewusste Aufteilung.

Eine neue Changelist erstellen

Im **Commit**-Werkzeugfenster (siehe vorheriges Kapitel) kannst du zusätzliche Changelists anlegen:

1. Rechtsklick in den Bereich der Changelists → **New Changelist…**
2. Namen und optional eine Beschreibung vergeben
3. Festlegen, ob die neue Liste sofort **aktiv** werden soll

Die *aktive* Changelist ist diejenige, der PhpStorm automatisch neu erkannte oder neu hinzugefügte Dateien zuordnet. Es existiert immer genau eine aktive Changelist.

Dateien zwischen Changelists verschieben

Änderungen lassen sich jederzeit einer anderen Changelist zuordnen:

- Datei im Commit-Fenster per **Drag & Drop** in eine andere Liste ziehen
- Über das Kontextmenü **Move to Another Changelist…** wählen
- Mehrere Dateien gleichzeitig markieren und gemeinsam verschieben

Das ist der zentrale Arbeitsschritt: Statt Änderungen nur über den Dateibaum zu betrachten, sortierst du sie inhaltlich – etwa „Bugfix Login“, „Refactoring Repository-Klasse“ oder „Experimentelle Änderung, noch nicht fertig“.

Commits aus einzelnen Changelists erzeugen

Beim Commit im PhpStorm-Werkzeugfenster wählst du gezielt **eine** Changelist als Grundlage aus:

1. Changelist markieren, deren Inhalt committet werden soll
2. Über das Kontextmenü **Commit** oder den Commit-Button im Fenster ausführen
3. Nur die Dateien dieser Liste werden Bestandteil des Commits – alle anderen Changelists bleiben unangetastet

So entstehen saubere, thematisch klare Commits, selbst wenn im Arbeitsverzeichnis gleichzeitig mehrere unabhängige Änderungen offen sind – ganz ohne `git add -p` oder manuelles Selektieren einzelner Zeilen bei jedem Commit erneut.

```
flowchart LR
    A["Arbeitsverzeichnis mit vielen Aenderungen"] --> B["Changelist Bugfix"]
    A --> C["Changelist Feature"]
    A --> D["Changelist Experiment"]
    B --> E["Commit 1"]
    C --> F["Commit 2"]
    D -.->|bleibt offen| D
```

Inaktive Changelists und ihr Nutzen

Eine Changelist muss nicht sofort committet werden. Du kannst sie beliebig lange „liegen lassen“, während du an einer anderen Liste weiterarbeitest. Das ist hilfreich für:

- **Angefangene, aber unfertige Experimente**, die noch nicht reif für einen Commit sind
- **Vorbereitete Änderungen für einen späteren Zeitpunkt**, etwa nach Abschluss eines Reviews
- **Getrennte Verantwortlichkeiten**, wenn du absichtlich mehrere kleine, unabhängige Themen parallel im selben Arbeitsverzeichnis verfolgst

Changelists und Diffs

Jede Changelist besitzt eine eigene Diff-Übersicht. Ein Doppelklick auf eine Datei innerhalb einer Changelist öffnet den Diff nur für diese Datei – unabhängig davon, wie viele andere Änderungen im Projekt insgesamt offen sind. Das erleichtert die Kontrolle vor dem Commit erheblich, weil du dich ausschließlich auf den relevanten Ausschnitt konzentrierst.

Abgrenzung zu Shelving

Changelists solltest du nicht mit dem *Shelf* verwechseln, einem verwandten, aber eigenständigen PhpStorm-Werkzeug:

Merkmal	Changelist	Shelf
Zweck	Thematische Gruppierung offener Änderungen	Temporäres Zurücklegen von Änderungen

Merkmal	Changelist	Shelf
Sichtbarkeit im Arbeitsverzeichnis	Ja, Dateien bleiben geändert sichtbar	Nein, Änderungen werden aus dem Arbeitsverzeichnis entfernt
Git-Äquivalent	Kein direktes Äquivalent	Ähnlich zu `git stash`
Typischer Einsatz	Parallele Commits vorbereiten	Kurzfristig „Platz schaffen“, z. B. für einen dringenden Branch-Wechsel

Wenn du also nur kurzfristig störende Änderungen aus dem Weg räumen möchtest, ist der Shelf das passendere Werkzeug; für die dauerhafte, inhaltliche Organisation mehrerer parallel offener Themen sind Changelists die richtige Wahl.

Praktische Hinweise für den Alltag

- **Sprechende Namen verwenden:** Eine Changelist namens „Fix #482 Validierung“ ist hilfreicher als „Changelist 2“.
- **Nicht zu granular arbeiten:** Für die meisten Aufgaben reicht die Default-Liste. Zusätzliche Changelists lohnen sich erst bei echter Parallelität.
- **Vor dem Commit prüfen:** Kontrolliere immer, welche Changelist aktuell aktiv ist, bevor du neue Dateien erstellst – sonst landen sie versehentlich in der falschen Gruppe.
- **Kein Ersatz für Branches:** Changelists ersetzen keine Branch-Strategie. Sie helfen bei kurzfristiger, lokaler Organisation, nicht bei langfristig getrennter Entwicklung.

Mit diesem Werkzeug behältst du auch in unübersichtlichen Arbeitsphasen die Kontrolle darüber, *was* in welchen Commit einfließt – ein wichtiger Baustein für die im nächsten Abschnitt behandelte Zuordnung von IDE-Aktionen zu den entsprechenden Git-Befehlen auf der Kommandozeile.

IDE-Aktionen und CLI-Befehle zuordnen

Warum diese Zuordnung wichtig ist

PhpStorm übersetzt jede Schaltfläche, jeden Menüpunkt und jede Diff-Ansicht letztlich in einen oder mehrere Git-Befehle, die im Hintergrund ausgeführt werden. Wer nur die Oberfläche bedient, ohne die zugrunde liegenden Kommandos zu kennen, gerät bei Fehlermeldungen, ungewöhnlichen Repository-Zuständen oder in fremden Umgebungen ohne IDE schnell an eine Grenze. Wer umgekehrt nur die Kommandozeile beherrscht, verschenkt die visuellen Vorteile von PhpStorm bei Diffs, Konflikten und Historienübersicht.

Dieses Kapitel schafft daher eine bewusste *Brücke*: Für jede zentrale IDE-Aktion wird der entsprechende Git-Befehl benannt, damit du jederzeit zwischen beiden Werkzeugen wechseln kannst – je nachdem, was in der jeweiligen Situation effizienter oder transparenter ist.

Grundprinzip: PhpStorm ist eine Oberfläche über Git

PhpStorm ruft standardmäßig das auf deinem System installierte Git-Programm auf (siehe Abschnitt *„Git-Programm und Version prüfen“*). Es erzeugt keine eigene, abweichende Versionslogik. Das bedeutet:

- Jeder Commit über die Oberfläche entspricht exakt einem `git commit`.
- Jeder Branch-Wechsel entspricht einem `git switch` bzw. `git checkout`.
- Konflikte, Merges und Rebases folgen denselben internen Regeln wie auf der Kommandozeile.

Diese Deckungsgleichheit ist beabsichtigt und erlaubt es dir, ein und dasselbe Repository nahtlos mit beiden Werkzeugen zu bearbeiten – auch im Wechsel innerhalb eines Arbeitstages.

Zuordnungstabelle: Kernaktionen

Die folgende Übersicht zeigt die wichtigsten Aktionen aus PhpStorms VCS-Menü, dem *Commit*-Werkzeugfenster und dem *Git*-Werkzeugfenster mit ihren CLI-Entsprechungen.

PhpStorm-Aktion	Ort in der Oberfläche	Entsprechender Git-Befehl
Projekt unter Versionskontrolle stellen	*VCS → Enable Version Control Integration*	`git init`
Dateien vormerken	*Commit*-Fenster, Checkbox aktivieren	`git add <Datei>`
Commit erstellen	*Commit*-Fenster → *Commit*	`git commit -m "…"`
Commit und Push in einem Schritt	*Commit*-Fenster → *Commit and Push…*	`git commit -m "…"` gefolgt von `git push`
Letzten Commit ändern	*Commit*-Fenster → *Amend Commit*	`git commit --amend`
Änderungen ansehen	Doppelklick auf Datei im *Commit*-Fenster	`git diff`
Datei aus dem Commit ausschließen	Checkbox deaktivieren	`git restore --staged <Datei>`
Branch erstellen	*Git → Branches… → New Branch*	`git branch <Name>` oder `git checkout -b <Name>`
Branch wechseln	*Git → Branches…* → Branch auswählen → *Checkout*	`git switch <Name>`
Branches zusammenführen	*Git → Branches… → Merge into Current*	`git merge <Name>`
Rebase durchführen	*Git → Branches… → Rebase Current onto Selected*	`git rebase <Name>`
Änderungen abrufen	*Git → Fetch*	`git fetch`
Änderungen integrieren	*Git → Pull…*	`git pull`
Änderungen veröffentlichen	*Git → Push…*	`git push`
Arbeit zwischenlagern	*Git → Uncommitted Changes → Stash Changes…*	`git stash`
Stash anwenden	*Git → Uncommitted Changes → Unstash Changes…*	`git stash pop` bzw. `git stash apply`
Commit rückgängig machen	*Git → Uncommitted Changes* oder Log-Kontextmenü → *Revert Commit*	`git revert <Commit>`
Zu einem Commit zurücksetzen	Log-Kontextmenü → *Reset Current Branch to Here*	`git reset`
Datei-Historie ansehen	Rechtsklick auf Datei → *Git → Show History*	`git log --follow <Datei>`

PhpStorm-Aktion	Ort in der Oberfläche	Entsprechender Git-Befehl
Zeilenverantwortung prüfen	*Annotate*-Randleiste	`git blame <Datei>`
Repository klonen	*Get from VCS...*	`git clone <URL>`

Warum die Reihenfolge manchmal abweicht

Ein wichtiger Unterschied betrifft zusammengesetzte Aktionen. Der Button *Commit and Push* etwa führt intern zwei getrennte Git-Operationen aus, die auf der Kommandozeile ebenfalls zwei Befehle erfordern würden. PhpStorm bündelt dies lediglich komfortabel in einem Klick, ohne die eigentliche Git-Semantik zu verändern.

Ähnlich verhält es sich beim Branch-Wechsel mit gleichzeitigem Erstellen: *New Branch from Selected* entspricht `git checkout -b <Name> <Basis>` – ein einzelner CLI-Befehl, der in der IDE über einen kleinen Dialog abgebildet wird.

$$
\text{IDE-Aktion} ;=; \text{ein oder mehrere Git-Befehle in fester Reihenfolge}
$$

Entscheidungshilfe: IDE oder Kommandozeile?

```
flowchart TD
    A["Aufgabe steht an"] --> B{"Visuelle Übersicht nötig?"}
    B -->|Ja, z.B. Diff, Konflikt, Branch-Graph| C["PhpStorm verwenden"]
    B -->|Nein, einfacher Einzelbefehl| D{"Reproduzierbarkeit oder Skript nötig?"}
    D -->|Ja, z.B. CI, Automatisierung| E["Kommandozeile verwenden"]
    D -->|Nein, spontane Aktion| C

    style C fill:#2f6f4f,color:#ffffff
    style E fill:#37474f,color:#ffffff
```

Als grobe Faustregel gilt:

- **PhpStorm** eignet sich besonders für Diffs, Merge-Konflikte, Historienvergleiche und alles, was von einer visuellen Darstellung profitiert.
- **Die Kommandozeile** eignet sich für seltene, präzise Spezialbefehle, Automatisierung, Skripte und Situationen, in denen du exakt nachvollziehen musst, was passiert – etwa bei der Fehlersuche.

Praktischer Tipp: Befehle in PhpStorm sichtbar machen

PhpStorm protokolliert die tatsächlich ausgeführten Git-Befehle im Werkzeugfenster **Version Control → Console**. Diese Ansicht ist besonders lehrreich: Jede Aktion, die du über Menüs auslöst, erscheint dort als exakter Befehl inklusive Parameter.

> **Empfehlung:** Aktiviere die *Console*-Registerkarte dauerhaft während der ersten Wochen mit PhpStorm. So verinnerlichst du die Zuordnung zwischen Oberfläche und Kommandozeile automatisch – ganz ohne Auswendiglernen.

Diese bewusste Verknüpfung beider Arbeitsweisen bildet die Grundlage für alle folgenden Kapitel, in denen Befehle und IDE-Funktionen gleichrangig und wechselseitig ergänzend eingesetzt werden.