Kapitel 3: Das erste Repository erstellen

Einleitung: Jetzt wird es praktisch! Ein „Repository" (kurz: Repo) ist im Grunde dein Projekt-Ordner, der von Git überwacht wird. Sobald du einen Ordner in ein Repository verwandelst, beginnt Git damit, alle Änderungen zu tracken. Dabei ist es wichtig zu verstehen, dass Git nicht einfach alles automatisch speichert – du entscheidest bewusst, welche Änderungen aufgezeichnet werden sollen. Außerdem lernst du die `.gitignore`-Datei kennen, mit der du Git sagst, welche Dateien es ignorieren soll (zum Beispiel temporäre Dateien oder sensible Konfigurationsdaten). Dieses Kapitel vermittelt dir das Verständnis für Gits dreistufiges System aus Working Directory, Staging Area und Repository.

- Ein bestehendes Projekt in PhpStorm zum Git-Repository machen
- Die .gitignore-Datei – was sie ist und warum du sie brauchst
- Die drei Bereiche in Git: Working Directory, Staging Area und Repository

Ein bestehendes Projekt in PhpStorm zum Git-Repository machen □□

Du hast also schon einen Projektordner mit PHP-Dateien und möchtest jetzt anfangen, Git dafür zu nutzen. Das ist ein sehr häufiger Anwendungsfall – und PhpStorm macht es dir dabei besonders leicht.

Schritt-für-Schritt-Anleitung

1. **Projekt in PhpStorm öffnen**
   Falls noch nicht geschehen, öffne dein bestehendes Projekt über **File → Open** und wähle den Projektordner aus.
2. **Git-Repository initialisieren**
   Gehe im Menü auf **VCS → Enable Version Control Integration…** (alternativ findest du es manchmal unter **Git → Enable Version Control Integration…**, je nach PhpStorm-Version).
   Es öffnet sich ein kleines Dialogfenster, in dem du das Versionskontrollsystem auswählen kannst. Wähle **Git** aus und bestätige mit **OK**.
3. **Änderungen im Interface beobachten**
   Nach der Initialisierung passieren sofort sichtbare Dinge:
   - Die Dateinamen in der Projektansicht werden **rot eingefärbt** – das bedeutet, dass diese Dateien noch nicht von Git „getrackt" werden (also noch nicht zur Versionskontrolle hinzugefügt wurden).
   - Unten in PhpStorm erscheint ein neuer Reiter namens **Git** (neben „Terminal", „Problems" usw.), über den du später Commits, Branches und die Historie verwalten kannst.
   - Das Menü **VCS** wird teilweise durch **Git** ersetzt oder erweitert.
4. **Dateien zur Staging Area hinzufügen**
   Damit Git deine Dateien überhaupt „kennt", musst du sie **stagen**. Du kannst:
   - Im Projektbaum mit Rechtsklick auf eine Datei oder einen Ordner gehen und **Git → Add** wählen.
   - Oder einfach alle Dateien auf einmal hinzufügen: Rechtsklick auf den Projektstammordner → **Git → Add**.

Die Dateien wechseln ihre Farbe von *rot* zu **grün** – das zeigt an, dass sie jetzt in der Staging Area liegen und beim nächsten Commit berücksichtigt werden.

5. **Ersten Commit erstellen**
   Jetzt ist der perfekte Moment für deinen **Initial Commit**:
   - Drücke `Ctrl + K` (Windows/Linux) oder `Cmd + K` (macOS) – oder gehe über **Git → Commit…**
   - Es öffnet sich das Commit-Fenster. Hier siehst du alle gestagten Dateien.
   - Gib eine **Commit-Message** ein, z. B.: `Initial commit: Projektstruktur hinzugefügt`
   - Klicke auf **Commit**.

   > □□ *Tipp:* Beim ersten Commit ist es üblich, eine kurze, beschreibende Nachricht wie „Initial commit" zu verwenden. Später solltest du aussagekräftigere Messages schreiben.

Was passiert dabei im Hintergrund? □□

Auch wenn PhpStorm dir die Arbeit abnimmt, laufen im Hintergrund ganz normale **Git-Befehle** ab. Hier die Entsprechung zu den Schritten oben:

PhpStorm-Aktion	Git-Befehl im Terminal
„Enable Version Control Integration" mit Git	`git init`
Dateien per „Git → Add" hinzufügen	`git add <datei>` oder `git add .`
Commit erstellen	`git commit -m "Nachricht"`

Der versteckte `.git`-Ordner

Sobald du Git initialisierst, erstellt Git einen **versteckten Ordner** namens `.git` im Wurzelverzeichnis deines Projekts. Dieser Ordner enthält:

- Die **gesamte Versionshistorie** deines Projekts
- Informationen über Branches, Commits und Konfigurationen
- Die Staging Area (auch „Index" genannt)

> ⚠ **Wichtig:** Lösche diesen Ordner niemals manuell, sonst verlierst du deine gesamte Git-Historie!

Du kannst den Ordner sehen, wenn du in deinem Dateimanager versteckte Dateien anzeigen lässt oder im Terminal `ls -la` (macOS/Linux) bzw. `dir /a` (Windows) eingibst.

Die drei Zustände deiner Dateien – visualisiert

Nach der Initialisierung befinden sich deine Dateien zunächst nur im **Working Directory**. Durch das Hinzufügen wandern sie in die **Staging Area**, und erst durch den Commit landen sie im **Repository**:

```
flowchart LR
    A["□□Working Directory<br>„Deine Dateien auf der Festplatte“"]
    B["□□Staging Area<br>„Vorgemerkt für den nächsten Commit“"]
    C["□□□Repository<br>„Dauerhaft gespeicherte Versionen“"]

    A -->|"git add<br>„Git → Add“ in PhpStorm"| B
    B -->|"git commit<br>„Commit“ in PhpStorm"| C
```

Was du jetzt hast

Nach diesen Schritten ist dein Projekt ein **vollwertiges lokales Git-Repository**. Du kannst ab sofort:

- Änderungen committen und eine Historie aufbauen
- Branches erstellen, um neue Features zu entwickeln
- Jederzeit zu früheren Versionen zurückkehren

Das Repository liegt aktuell nur **lokal auf deinem Computer**. Wenn du es später auf GitHub hochladen möchtest, um ein Backup zu haben oder mit anderen zusammenzuarbeiten, ist das ein separater Schritt – aber die Grundlage dafür hast du jetzt geschaffen. □□

Die .gitignore-Datei – was sie ist und warum du sie brauchst □□

Wenn du mit Git arbeitest, möchtest du **nicht jede Datei** in deinem Projektordner auch tatsächlich versionieren. Manche Dateien sind temporär, automatisch generiert oder enthalten sensible Informationen – sie gehören schlicht nicht ins Repository. Genau hier kommt die `.gitignore`-Datei ins Spiel.

Was ist die `.gitignore`-Datei?

Die `.gitignore` ist eine **einfache Textdatei** im Hauptverzeichnis deines Git-Repositories. Sie enthält eine Liste von Dateinamen, Ordnern oder Mustern, die Git **ignorieren** soll. Das bedeutet: Dateien, die auf diese Muster passen, werden von Git nicht getrackt – sie tauchen nicht in der Staging Area auf, werden nicht committet und landen somit auch nicht auf GitHub oder in deiner Versionshistorie.

> □□ Die Datei heißt wirklich `.gitignore` (mit Punkt am Anfang, ohne Dateiendung). Unter Windows kann das Erstellen einer solchen Datei manchmal etwas umständlich sein – in PhpStorm kannst du sie aber problemlos über **Rechtsklick → New → File** anlegen.

Warum ist das wichtig?

Es gibt mehrere gute Gründe, bestimmte Dateien **nicht** ins Repository aufzunehmen:

1. **Abhängigkeiten und generierte Dateien**
   Ordner wie `vendor/` (Composer) oder `node_modules/` (npm) können **tausende Dateien** enthalten und viele hundert Megabyte groß werden. Diese Dateien werden durch

`composer install` bzw. `npm install` automatisch aus den Paketquellen heruntergeladen – sie müssen also nicht versioniert werden. Die Konfigurationsdateien (`composer.json`, `package.json`) reichen völlig aus, um die Abhängigkeiten jederzeit wiederherzustellen.

2. **Sensible Daten**
   Dateien wie `.env` enthalten oft Passwörter, API-Schlüssel oder Datenbank-Zugangsdaten. Wenn du diese ins Repository eincheckst und auf GitHub pushst, sind sie unter Umständen **öffentlich sichtbar** – ein erhebliches Sicherheitsrisiko.
3. **IDE- und systemspezifische Dateien**
   PhpStorm speichert Projekteinstellungen im Ordner `.idea/`. Dein Betriebssystem legt möglicherweise Dateien wie `.DS_Store` (macOS) oder `Thumbs.db` (Windows) an. Diese Dateien sind nur für dich lokal relevant und würden bei anderen Teammitgliedern im besten Fall nur stören – im schlimmsten Fall Konflikte verursachen.
4. **Temporäre und Cache-Dateien**
   Logs, Caches und temporäre Dateien ändern sich ständig und haben keinen Mehrwert in der Versionshistorie. Sie würden nur die Historie „verschmutzen" und das Repository unnötig aufblähen.

Beispiel-`.gitignore` für ein PHP-Projekt

Hier ist eine **praxisnahe `.gitignore`-Datei**, wie du sie für ein typisches PHP-Projekt mit Composer und eventuell Node.js-basierten Build-Tools (z. B. für Frontend-Assets) verwenden könntest:

```
# ========================================
# Abhängigkeiten (werden über Paketmanager installiert)
# ========================================

# Composer-Abhängigkeiten
/vendor/

# Node.js-Abhängigkeiten (falls du npm/Yarn für Frontend-Tools nutzt)
/node_modules/



# ========================================
# Umgebungs- und Konfigurationsdateien mit sensiblen Daten
# ========================================
```

```
# Umgebungsvariablen (Passwörter, API-Keys, DB-Zugangsdaten)
.env
.env.local
.env.*.local


# ========================================
# IDE- und Editor-Einstellungen
# ========================================

# PhpStorm / JetBrains IDEs
/.idea/

# Visual Studio Code
/.vscode/


# ========================================
# Betriebssystem-spezifische Dateien
# ========================================

# macOS
.DS_Store

# Windows
Thumbs.db
Desktop.ini


# ========================================
# Logs, Caches und temporäre Dateien
# ========================================

# Allgemeine Log-Dateien
*.log

# Composer-Cache (normalerweise global, aber sicherheitshalber)
/composer.phar

# PHP-Cache-Dateien (z. B. von Frameworks wie Laravel oder Symfony)
```

```
/storage/logs/
/storage/framework/cache/
/storage/framework/sessions/
/storage/framework/views/
/bootstrap/cache/

# Build-Artefakte (falls du Frontend-Assets kompilierst)
/public/build/
/public/hot
/public/mix-manifest.json



# ========================================
# Tests und Coverage-Reports
# ========================================

# PHPUnit Coverage-Reports
/coverage/
.phpunit.result.cache
```

Erklärung der wichtigsten Einträge

Eintrag	Bedeutung
`/vendor/`	Schließt den kompletten Composer-Abhängigkeitsordner aus. Der führende Slash `/` bedeutet: nur im Hauptverzeichnis.
`/node_modules/`	Dasselbe für npm/Yarn-Pakete. Kann bei Frontend-Projekten leicht mehrere zehntausend Dateien enthalten.
`.env`	Umgebungsdatei mit sensiblen Konfigurationswerten – **niemals** ins Repository!
`/.idea/`	PhpStorm speichert hier Projekteinstellungen, die nur lokal relevant sind.
`.DS_Store`	Versteckte macOS-Systemdatei, die Finder-Einstellungen speichert.
`*.log`	Alle Dateien mit der Endung `.log` – also sämtliche Log-Dateien im gesamten Projekt.
`/storage/` und `/bootstrap/cache/`	Typische Cache-Verzeichnisse bei Laravel-Projekten.

Wie lege ich die Datei in PhpStorm an?

1. **Rechtsklick auf das Projektstammverzeichnis** im Projektbaum (links in PhpStorm).
2. Wähle **New → File** und gib als Namen `.gitignore` ein.
3. Füge die gewünschten Einträge ein und speichere die Datei.
4. **Committe die `.gitignore`-Datei selbst** – sie gehört ins Repository, damit alle Teammitglieder (oder du selbst auf anderen Rechnern) dieselben Regeln haben.

> ⚠ **Wichtig:** Wenn du eine Datei *bereits* committet hast und sie *danach* zur `.gitignore` hinzufügst, wird Git sie trotzdem weiter tracken. Du musst sie erst aus dem Index entfernen:
>
> ```
> git rm --cached dateiname
> ```
>
> Das entfernt die Datei aus dem Repository, lässt sie aber lokal auf deiner Festplatte bestehen.

Fazit

Die `.gitignore`-Datei ist ein **unverzichtbares Werkzeug** für saubere Git-Repositories. Sie sorgt dafür, dass nur relevanter Code und Konfiguration versioniert wird – ohne unnötigen Ballast, ohne Sicherheitsrisiken und ohne Konflikte durch nutzerspezifische Dateien. Nimm dir zu Beginn jedes Projekts kurz Zeit, eine sinnvolle `.gitignore` anzulegen – dein zukünftiges Ich wird es dir danken.

Die drei Bereiche in Git: Working Directory, Staging Area und Repository □□

Wenn du zum ersten Mal mit Git arbeitest, fragst du dich vielleicht, warum das Speichern von Änderungen so „kompliziert" ist. Statt einfach auf „Speichern" zu klicken, musst du Dateien erst *stagen* und dann *committen*. Das wirkt anfangs umständlich – hat aber sehr gute Gründe, die dir mit der Zeit enorm helfen werden.

Die drei Bereiche im Überblick

Git organisiert dein Projekt in **drei logische Bereiche**, die jeweils eine bestimmte Rolle spielen:

```
flowchart LR
    WD["□□□Working Directory\nDein Arbeitsordner"]
    SA["□□Staging Area\nVorbereitungsbereich"]
    REPO["□□Repository\nDie Versionshistorie"]

    WD -->|git add| SA
    SA -->|git commit| REPO
    REPO -->|git checkout| WD
```

1. Working Directory – dein Arbeitsordner

Das **Working Directory** ist schlicht der Ordner auf deiner Festplatte, in dem du arbeitest. Hier liegen alle Dateien deines Projekts so, wie du sie gerade siehst und bearbeitest. Wenn du eine PHP-Datei öffnest, etwas änderst und speicherst (im klassischen Sinne mit `Strg+S`), dann ist diese Änderung *nur* im Working Directory – Git weiß davon noch nichts Besonderes.

> **Merke:** Das Working Directory ist der „echte" Zustand deiner Dateien, so wie du sie gerade auf dem Bildschirm siehst.

2. Staging Area – der Vorbereitungsbereich

Die **Staging Area** (manchmal auch *Index* genannt) ist ein Zwischenbereich, in dem du **auswählst, welche Änderungen in den nächsten Commit kommen sollen**. Du kannst sie dir wie einen Einkaufswagen vorstellen: Du legst Dinge hinein, aber erst an der Kasse (beim Commit) wird wirklich „abgerechnet".

Mit dem Befehl `git add datei.php` (oder in PhpStorm per Rechtsklick → *Git* → *Add*) verschiebst du eine Änderung vom Working Directory in die Staging Area. Die Datei ist damit **vorgemerkt**, aber noch nicht dauerhaft gespeichert.

> **Merke:** Die Staging Area gibt dir die Kontrolle darüber, *was genau* in einem Commit landet – nicht einfach „alles, was ich geändert habe".

3. Repository – die Versionshistorie

Das **Repository** (genauer: das *lokale Repository*) ist die eigentliche Git-Datenbank. Hier werden alle Commits gespeichert – also alle „Schnappschüsse" deines Projekts mit Autor, Zeitstempel und einer Nachricht. Sobald du `git commit` ausführst, wandern die Änderungen aus der Staging Area ins Repository und werden **dauerhaft Teil der Versionsgeschichte**.

Das Repository liegt im versteckten Ordner `.git` in deinem Projektverzeichnis. Du arbeitest normalerweise nicht direkt damit, sondern über Git-Befehle.

> **Merke:** Das Repository ist das „Gedächtnis" deines Projekts – hier kannst du jederzeit zu früheren Zuständen zurückkehren.

Warum nicht einfach „speichern"?

Die Dreiteilung mag auf den ersten Blick übertrieben wirken, aber sie löst mehrere echte Probleme:

1. **Gezielte Commits statt „alles auf einmal"**
   Stell dir vor, du hast an drei verschiedenen Dingen gleichzeitig gearbeitet: einem Bugfix, einem neuen Feature und einer kleinen Textkorrektur. Ohne Staging Area müsstest du *alles zusammen* in einem Commit speichern – oder umständlich Dateien hin- und herkopieren. Mit der Staging Area kannst du **gezielt auswählen**: Erst den Bugfix committen, dann das Feature, dann die Textkorrektur. So bleibt deine Historie sauber und nachvollziehbar.
2. **Überprüfen vor dem Festschreiben**
   Die Staging Area gibt dir einen Moment zum Innehalten. Du kannst mit `git status` oder in PhpStorm im *Commit*-Fenster genau sehen, welche Änderungen du gleich committen wirst. Das verhindert, dass versehentlich Debug-Code, temporäre Dateien oder unfertige Änderungen in die Versionshistorie gelangen.
3. **Teilweises Stagen möglich**
   Du kannst sogar *Teile* einer Datei stagen (sogenanntes „partial staging" oder „hunks"). Wenn du in einer Datei zwei unabhängige Änderungen gemacht hast, kannst du nur eine davon in den nächsten Commit aufnehmen. Das ist mit einem einfachen „Speichern"-Konzept nicht möglich.
4. **Sicherheit und Flexibilität**
   Solange Änderungen nur im Working Directory sind, kannst du sie jederzeit verwerfen. Solange sie nur in der Staging Area sind, kannst du sie wieder „unstagen". Erst wenn sie im Repository sind, sind sie dauerhaft (und selbst dann kannst du mit Git noch einiges korrigieren, aber das ist ein Thema für später).

Ein praktisches Beispiel

Angenommen, du arbeitest an einer Website und hast folgende Änderungen gemacht:

- `index.php` – neues Kontaktformular hinzugefügt
- `style.css` – Farben angepasst
- `config.php` – versehentlich dein Datenbank-Passwort im Klartext eingetragen

Ohne Staging Area würdest du alle drei Dateien auf einmal committen – inklusive des Passworts.

Mit Staging Area machst du Folgendes:

```
git add index.php
git add style.css
# config.php wird NICHT hinzugefügt!
git commit -m "Kontaktformular und Farbänderungen hinzugefügt"
```

Du hast volle Kontrolle darüber, was ins Repository kommt. Das Passwort bleibt draußen, du kannst es in Ruhe korrigieren und später separat committen (oder die Datei in die `.gitignore` aufnehmen).

Zusammenfassung als Tabelle

Bereich	Was ist das?	Git-Befehl zum Wechsel
Working Directory	Dein Projektordner mit allen aktuellen Dateien	–
Staging Area	Vorbereitungsbereich für den nächsten Commit	`git add`
Repository	Dauerhafte Versionshistorie aller Commits	`git commit`

Fazit □

Die drei Bereiche in Git sind kein unnötiger Overhead, sondern ein **durchdachtes System**, das dir erlaubt:

- saubere, thematisch getrennte Commits zu erstellen,
- Änderungen vor dem Festschreiben zu überprüfen,
- Fehler zu vermeiden, bevor sie in der Historie landen.

Sobald du dich daran gewöhnt hast, wirst du die Staging Area als eines der nützlichsten Features von Git schätzen lernen. In PhpStorm siehst du diese drei Bereiche übrigens sehr anschaulich im *Commit*-Fenster: Links die geänderten Dateien (Working Directory), in der Mitte die zum Commit vorgemerkten (Staging Area), und nach dem Commit landen sie in der Historie (Repository). □□