Dateien mit git add vormerken

Dateien mit `git add` vormerken

Nachdem du mit `git status` den Zustand deines Projekts eingeordnet hast, folgt der nächste logische Schritt: Du entscheidest bewusst, *welche* Änderungen Teil des nächsten Commits werden sollen. Genau das ist die Aufgabe von `git add` – dem Befehl, mit dem Dateien in die sogenannte *Staging Area* (auch *Index* genannt) aufgenommen werden.

Vom Arbeitsverzeichnis in den Index

Git trennt bewusst zwischen dem, was in deinen Dateien steht, und dem, was tatsächlich im nächsten Commit landen soll. Diese Trennung ist kein technisches Detail, sondern eines der zentralen Konzepte von Git überhaupt – sie wird in Kapitel 5 vertieft. An dieser Stelle reicht folgendes Modell:

```
flowchart LR
    A["Arbeitsverzeichnis"] -->|git add| B["Staging Area Index"]
    B -->|git commit| C["Repository .git"]
```

`git add` kopiert also nicht einfach eine Datei, sondern merkt den *aktuellen Inhalt* dieser Datei für den nächsten Commit vor. Änderst du die Datei danach erneut, musst du sie erneut hinzufügen – ein Verhalten, das anfangs oft verwundert, aber sehr präzise Kontrolle ermöglicht.

Der Grundbefehl

Am einfachsten fügst du eine einzelne Datei hinzu:

```
git add index.php
```

Mehrere Dateien lassen sich in einem Aufruf übergeben:

```
git add index.php composer.json README.md
```

Auch Wildcards funktionieren, sofern deine Shell sie unterstützt:

```
git add src/*.php
```

Ganze Verzeichnisse und das Projekt komplett stagen

Für PHP-Projekte mit mehreren Verzeichnissen – etwa `src/`, `tests/` und `public/` – ist es oft praktischer, ganze Ordner vorzumerken:

```
git add src
git add tests
```

Willst du *alle* Änderungen im aktuellen Verzeichnis und dessen Unterverzeichnissen erfassen, verwendest du:

```
git add .
```

Wichtig: Der Punkt bezieht sich auf das aktuelle Arbeitsverzeichnis. Führst du den Befehl aus einem Unterordner aus, werden auch nur dessen Inhalte berücksichtigt – nicht das gesamte Repository.

`git add .` vs. `git add -A` vs. `git add -u`

Diese drei Varianten werden häufig verwechselt, unterscheiden sich aber in einem entscheidenden Punkt: dem Umgang mit **gelöschten** Dateien.

Befehl	Neue Dateien	Geänderte Dateien	Gelöschte Dateien	Bereich
`git add .`	✔	✔	✔ *(moderne Git-Versionen)*	Aktuelles Verzeichnis abwärts
`git add -A`	✔	✔	✔	Gesamtes Repository
`git add -u`	✘	✔	✔	Gesamtes Repository, nur verfolgte Dateien

Hinweis: Seit Git 2.x verhält sich `git add .` im gesamten Repository-Kontext praktisch identisch zu `git add -A`, sofern du dich im Wurzelverzeichnis befindest. In älteren Versionen gab es hier Unterschiede – solltest du auf ein Legacy-System stoßen, prüfe das Verhalten sicherheitshalber mit `git status`

.

Den Effekt kontrollieren

`git add` verändert nichts sichtbar an deinen Dateien – der Effekt zeigt sich ausschließlich im Status:

```
git status
```

Vor dem Staging:

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        src/Calculator.php
```

Nach `git add src/Calculator.php`:

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   src/Calculator.php
```

Diese farbliche und textuelle Unterscheidung zwischen *„Changes not staged“* und *„Changes to be committed“* ist eine der wichtigsten Orientierungshilfen im täglichen Git-Alltag.

Ein praktisches Beispiel

Angenommen, du arbeitest an einem kleinen PHP-Projekt und hast folgende Änderungen vorliegen:

- eine neue Klasse `src/Calculator.php`
- eine geänderte `composer.json`
- eine gelöschte, nicht mehr benötigte Datei `legacy.php`

```
git status
```

```
Changes not staged for commit:
        modified:   composer.json
        deleted:    legacy.php

Untracked files:
        src/Calculator.php
```

Möchtest du **alles** in den nächsten Commit übernehmen:

```
git add -A
```

Möchtest du hingegen nur die neue Klasse vormerken und die anderen Änderungen bewusst für einen späteren, separaten Commit zurückhalten:

```
git add src/Calculator.php
```

`git status` zeigt danach exakt, was gestaged ist und was noch offen bleibt – ein zentrales Werkzeug, um Commits klein und thematisch sauber zu halten (siehe Kapitel 6).

Ein häufiges Missverständnis

Ein weit verbreiteter Irrtum lautet: *„git add speichert meine Datei."* Das ist nicht korrekt. `git add` verändert **nur den Index**, nicht das Repository. Erst `git commit` überführt den Inhalt des Index dauerhaft in die Historie. Bis dahin lässt sich jede Vormerkung problemlos wieder zurücknehmen – etwa mit:

```
git restore --staged <datei>
```

Diese Rücknahme wird in Kapitel 5 ausführlich behandelt; wichtig ist an dieser Stelle nur das Bewusstsein, dass Staging ein **reversibler Zwischenschritt** ist.

Selektives Vorgehen als Gewohnheit

Auch wenn `git add -A` in vielen Situationen bequem erscheint, lohnt es sich, frühzeitig ein bewussteres Vorgehen einzuüben:

1. Änderungen mit `git status` und `git diff` sichten.
2. Nur thematisch zusammengehörige Dateien stagen.
3. Bei Bedarf mit `git status` erneut prüfen, ob wirklich nur das Gewünschte vorgemerkt ist.
4. Erst dann committen.

Diese Reihenfolge verhindert eines der häufigsten Anfängerprobleme: Commits, die versehentlich unfertigen Code, temporäre Debug-Ausgaben oder unabsichtlich geänderte Konfigurationsdateien enthalten.

Ausblick

Der hier vorgestellte Grundmechanismus reicht für den Einstieg vollkommen aus. Im weiteren Kursverlauf – insbesondere in Kapitel 5 – vertiefst du das Staging deutlich: interaktives Hinzufügen

einzelner Codeblöcke mit `git add -p`, das gezielte Entfernen aus dem Index sowie die Diagnose komplexer Zustände. Zunächst genügt es, den zentralen Grundsatz verinnerlicht zu haben:

> *Nichts gelangt in einen Commit, das nicht zuvor bewusst mit `git add` vorgemerkt wurde.*

Damit ist die Grundlage gelegt, um im nächsten Abschnitt den ersten echten Commit zu erstellen.

Revision #1
Created 2026-07-26 17:12:07 UTC by art10m
Updated 2026-07-26 17:12:23 UTC by art10m