Git als Snapshot-System

Die zentrale Idee: Git speichert Zustände, nicht nur Änderungen

Viele Versionskontrollsysteme werden zunächst als eine Folge von Änderungen verstanden:

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

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

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

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

Statt gedanklich nur zu fragen:

> „Welche Zeilen haben sich seit gestern geändert?“

kannst du bei Git fragen:

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

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

Was ein Snapshot enthält

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

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

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

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

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

Später ergänzt du eine README-Datei:

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

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

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

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

Git zeigt Änderungen durch den Vergleich von Snapshots

Wenn du diesen Befehl ausführst:

```
git diff
```

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

Git vergleicht beispielsweise:

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

Beispiel:

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

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

Die sichtbare Änderung könnte so aussehen:

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

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

Ein Commit ist kein vollständiges Dateikopie-Archiv

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

Das wäre bei größeren Projekten ineffizient.

Stattdessen speichert Git Inhalte objektbasiert:

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

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

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

Das verbindet Verständlichkeit mit hoher Effizienz.

Beispiel: Drei Zustände eines Projekts

Stell dir diese Entwicklung vor.

Commit A: Projekt anlegen

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

`src/Greeting.php`:

```
<?php

declare(strict_types=1);

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

Commit B: Test ergänzen

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

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

Commit C: Begrüßung ändern

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

Nun wird der Inhalt verändert:

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

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

Vereinfacht lässt sich die Historie so lesen:

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

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

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

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

Warum dieses Modell so nützlich ist

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

Alte Projektzustände ansehen

Du kannst jederzeit einen früheren Commit untersuchen:

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

Oder du wechselst gezielt zu einem historischen Zustand:

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

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

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

Einen Branch erstellen

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

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

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

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

Einen Merge durchführen

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

Git betrachtet dafür typischerweise:

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

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

Änderungen zurückholen

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

Beispiel:

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

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

Snapshots und die drei Bereiche von Git

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

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

Ein typischer Ablauf sieht so aus:

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

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

Das ist ein entscheidender Unterschied zu einem bloßen automatischen Speichern.

Der Index: Entwurf des nächsten Snapshots

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

Angenommen, du hast parallel:

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

Mit dem Index kannst du gezielt einen sauberen Snapshot vorbereiten:

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

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

Der Commit beschreibt dadurch einen klaren, überprüfbaren Zustand:

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

Dateien werden nicht als „umbenannt“ gespeichert

Eine häufig überraschende Konsequenz des Snapshot-Modells betrifft Umbenennungen.

Wenn du ausführst:

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

speichert Git grundsätzlich einen neuen Snapshot mit:

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

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

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

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

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

Snapshot-Modell und Speicherbedarf

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

Praktisch bedeutet das:

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

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

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

Typische Missverständnisse vermeiden

„Git speichert nur Unterschiede“

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

„Jeder Commit kopiert das gesamte Projekt“

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

„Ein Commit sichert automatisch alles“

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

Prüfe daher vor jedem Commit:

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

„Git erkennt jede Umbenennung dauerhaft“

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

„Ein Snapshot ist ein Backup ohne Grenzen“

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

Praktische Kontrolle mit Git

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

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

# Nicht vorgemerkte Änderungen ansehen
git diff

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

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

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

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

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

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

Merksätze

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

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

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

Revision #1
Created 2026-07-25 11:13:39 UTC by art10m
Updated 2026-07-25 11:13:39 UTC by art10m