Git-Referenzen, Branches und HEAD – Was wirklich passiert □□

Um Git wirklich zu verstehen, musst du begreifen, dass **Branches keine Kopien deines Codes** sind – sie sind lediglich winzige Textdateien, die auf einen Commit zeigen. Dieses Kapitel enthüllt die elegante Einfachheit, die hinter Gits scheinbar komplexem Branching-System steckt.

Das Konzept der Referenzen (refs)

Eine **Referenz** (kurz: *ref*) ist in Git nichts anderes als ein **lesbarer Name für einen SHA-1-Hash**. Anstatt dir kryptische Hashes wie `a1b2c3d4e5f6...` merken zu müssen, kannst du einfach `main`, `feature/login` oder `v1.0.0` verwenden.

Was refs technisch sind

Jede Referenz ist eine **simple Textdatei**, die genau 40 Zeichen enthält – den SHA-1-Hash eines Commits. Das ist alles. Keine Magie, keine komplexe Datenstruktur, nur eine Zeile Text.

```
a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4
```

Die verschiedenen Arten von Referenzen

Git kennt mehrere Kategorien von Referenzen, die alle im `.git/`-Verzeichnis organisiert sind:

Referenz-Typ	Speicherort	Beispiel	Zweck
Lokale Branches	`.git/refs/heads/`	`refs/heads/main`	Deine lokalen Entwicklungszweige
Remote-Tracking-Branches	`.git/refs/remotes/`	`refs/remotes/origin/main`	Abbild der Branches auf dem Remote

Referenz-Typ	Speicherort	Beispiel	Zweck
Tags	`.git/refs/tags/`	`refs/tags/v1.0.0`	Benannte Marker für Releases
HEAD	`.git/HEAD`	–	Zeigt auf den aktuellen Branch
Stash	`.git/refs/stash`	–	Zwischengespeicherte Änderungen

Die Anatomie des `.git/refs/`-Verzeichnisses

Schauen wir uns an, wie ein typisches Repository strukturiert ist:

```
.git/
├── HEAD                    ← Zeigt auf aktuellen Branch
├── refs/
│   ├── heads/              ← Alle lokalen Branches
│   │   ├── main            ← Der main-Branch (eine Textdatei!)
│   │   ├── develop         ← Der develop-Branch
│   │   └── feature/
│   │       └── login       ← Feature-Branch mit Unterordner
│   ├── remotes/
│   │   └── origin/         ← Remote-Tracking-Branches
│   │       ├── main
│   │       └── develop
│   └── tags/
│       ├── v1.0.0          ← Lightweight Tag
│       └── v2.0.0          ← Annotated Tag (zeigt auf Tag-Objekt)
└── packed-refs             ← Optimierte Speicherung vieler Refs
```

Praktische Untersuchung in deinem Repository

Du kannst dir die Refs direkt im Dateisystem ansehen:

```
# Zeige den Inhalt des main-Branch (nur der Hash!)
cat .git/refs/heads/main
# Ausgabe: a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4

# Liste alle lokalen Branches als Dateien
ls -la .git/refs/heads/

# Zeige alle Referenzen mit git
git show-ref
```

Die `packed-refs`-Datei

Bei Repositories mit vielen Branches und Tags optimiert Git die Speicherung, indem es Referenzen in einer einzigen Datei zusammenfasst:

```
cat .git/packed-refs
```

```
# pack-refs with: peeled fully-peeled sorted
a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4 refs/heads/main
b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5 refs/heads/develop
c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6 refs/tags/v1.0.0
```

Git sucht Referenzen zuerst in den einzelnen Dateien unter `refs/` und dann in `packed-refs`. Wenn du einen Branch aktualisierst, wird eine neue Einzeldatei erstellt, die Vorrang vor dem Eintrag in `packed-refs` hat.

Branches – Nichts als bewegliche Zeiger

Ein **Branch** ist eine Referenz, die automatisch mitwandert, wenn du neue Commits erstellst. Das unterscheidet Branches von Tags, die fest an einem Commit verankert bleiben.

Was einen Branch ausmacht

```
gitGraph
    commit id: "C1"
    commit id: "C2"
    branch feature/login
    commit id: "C3"
    commit id: "C4"
    checkout main
    commit id: "C5"
```

In diesem Diagramm sind `main` und `feature/login` **keine Kopien des Codes** – sie sind lediglich zwei 40-Byte-Dateien, die auf verschiedene Commits zeigen:

- `.git/refs/heads/main` → zeigt auf C5
- `.git/refs/heads/feature/login` → zeigt auf C4

Was passiert, wenn du einen Branch erstellst?

Wenn du `git branch feature/login` ausführst, passiert **erstaunlich wenig**:

1. **Git ermittelt den aktuellen Commit** – den Hash, auf den HEAD (über den aktuellen Branch) zeigt
2. **Git erstellt eine neue Datei** unter `.git/refs/heads/feature/login`
3. **Git schreibt den Hash** in diese Datei

Das ist alles! Kein Kopieren von Dateien, kein Duplizieren von Historie. Ein Branch zu erstellen ist deshalb in Git **instantan und kostet praktisch keinen Speicherplatz** – egal wie groß dein Repository ist.

```
# Diese beiden Befehle sind funktional identisch:

# Der "offizielle" Weg:
git branch feature/login

# Was Git intern macht (vereinfacht):
git rev-parse HEAD > .git/refs/heads/feature/login
```

Demonstration: Branch manuell erstellen

Du kannst tatsächlich einen Branch „von Hand" erstellen:

```
# Aktuellen Commit-Hash ermitteln
git rev-parse HEAD
# Ausgabe: a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4

# Branch manuell erstellen (nicht empfohlen, aber lehrreich!)
echo "a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4" > .git/refs/heads/mein-manueller-branch

# Überprüfen, ob es funktioniert hat
git branch
# Ausgabe enthält jetzt: mein-manueller-branch
```

HEAD – Der spezielle Zeiger auf „Wo bin ich jetzt?"

HEAD ist die wichtigste Referenz in Git. Sie beantwortet die Frage: *„In welchem Branch arbeite ich gerade, und welchen Commit sehe ich in meinem Working Directory?*"

HEAD ist (meistens) eine symbolische Referenz

Im Normalfall enthält `.git/HEAD` keinen direkten Commit-Hash, sondern einen **Verweis auf einen Branch**:

```
cat .git/HEAD
# Ausgabe: ref: refs/heads/main
```

Das bedeutet: *„HEAD zeigt auf den Branch main, und main zeigt auf einen bestimmten Commit.*" Diese Indirektion ist wichtig, denn sie ermöglicht es Git, den Branch automatisch mitzubewegen, wenn du einen neuen Commit erstellst.

Die HEAD-Kette visualisiert

```
flowchart LR
    HEAD["HEAD\n.git/HEAD"]
```

```
    main["main\n.git/refs/heads/main"]
    commit["Commit\na1b2c3d4..."]

    HEAD -->|"ref: refs/heads/main"| main
    main -->|"a1b2c3d4..."| commit

    style HEAD fill:#e1f5fe,color:#000000
    style main fill:#fff3e0,color:#000000
    style commit fill:#f3e5f5,color:#000000
```

Detached HEAD – Wenn HEAD direkt auf einen Commit zeigt

Manchmal zeigt HEAD **direkt auf einen Commit** statt auf einen Branch. Das nennt man „Detached HEAD":

```
# Einen bestimmten Commit auschecken (nicht einen Branch)
git checkout a1b2c3d4

cat .git/HEAD
# Ausgabe: a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4
```

Im Detached HEAD-Zustand zeigt HEAD nicht mehr auf `ref: refs/heads/...`, sondern enthält direkt den Hash. **Neue Commits, die du jetzt machst, gehören zu keinem Branch** und können verloren gehen, wenn du den Branch wechselst!

```
flowchart TB
    subgraph normal["Normaler Zustand"]
        HEAD1["HEAD"] --> main1["main"] --> C1["Commit C1"]
    end

    subgraph detached["Detached HEAD"]
        HEAD2["HEAD"] --> C2["Commit C2"]
        main2["main"] --> C3["Commit C3"]
    end

    style HEAD1 fill:#c8e6c9,color:#000000
    style HEAD2 fill:#ffcdd2,color:#000000
    style main1 fill:#fff3e0,color:#000000
```

```
    style main2 fill:#fff3e0,color:#000000
```

Was passiert beim Branch-Wechsel? □□

Wenn du `git checkout feature/login` oder `git switch feature/login` ausführst, passieren **mehrere Dinge**:

Schritt 1: Git aktualisiert HEAD

```
# Vorher:
cat .git/HEAD
# ref: refs/heads/main

# Nach git switch feature/login:
cat .git/HEAD
# ref: refs/heads/feature/login
```

Schritt 2: Git ermittelt den Ziel-Commit

Git liest den Hash aus `.git/refs/heads/feature/login` und weiß jetzt, welchen Zustand des Projekts es herstellen muss.

Schritt 3: Git aktualisiert das Working Directory

Git vergleicht den aktuellen Commit mit dem Ziel-Commit und:

- **Löscht Dateien**, die im Ziel-Commit nicht existieren
- **Erstellt Dateien**, die im Ziel-Commit neu sind
- **Aktualisiert Dateien**, die sich unterscheiden

Schritt 4: Git aktualisiert den Index (Staging Area)

Der Index wird aktualisiert, um den Zustand des Ziel-Commits widerzuspiegeln.

Visualisierung des gesamten Prozesses

```
flowchart TB
    subgraph vorher["Vor dem Wechsel"]
        HEAD_v["HEAD\nref: refs/heads/main"]
        main_v["main\na1b2c3d4"]
        feature_v["feature/login\nb2c3d4e5"]
        WD_v["Working Directory\nZustand von main"]
    end

    subgraph nachher["Nach git switch feature/login"]
        HEAD_n["HEAD\nref: refs/heads/feature/login"]
        main_n["main\na1b2c3d4"]
        feature_n["feature/login\nb2c3d4e5"]
        WD_n["Working Directory\nZustand von feature/login"]
    end

    vorher --> |"git switch feature/login"| nachher

    style HEAD_v fill:#fff3e0,color:#000000
    style HEAD_n fill:#c8e6c9,color:#000000
    style WD_v fill:#e3f2fd,color:#000000
    style WD_n fill:#e8f5e9,color:#000000
```

Warum Branch-Wechsel manchmal fehlschlägt

Git verhindert den Branch-Wechsel, wenn du **uncommittete Änderungen** hast, die beim Wechsel überschrieben würden:

```
git switch feature/login
# error: Your local changes to the following files would be overwritten by checkout:
#     src/Login.php
# Please commit your changes or stash them before you switch branches.
```

Dieses Verhalten schützt dich davor, Arbeit zu verlieren.

Was passiert beim Erstellen eines neuen Commits?

Wenn du `git commit` ausführst, passiert Folgendes mit den Referenzen:

1. **Git erstellt ein neues Commit-Objekt** mit:
   - Tree-Objekt (Snapshot aller Dateien)
   - Parent-Verweis (der vorherige Commit)
   - Autor, Committer, Nachricht, Zeitstempel
2. **Git ermittelt den aktuellen Branch** durch Lesen von HEAD
3. **Git aktualisiert die Branch-Referenz** – die Datei unter `refs/heads/` wird mit dem neuen Commit-Hash überschrieben

```
flowchart TB
    subgraph vorher["Vor dem Commit"]
        H1["HEAD\nref: refs/heads/main"]
        M1["main\na1b2c3d4"]
        C1["Commit\na1b2c3d4"]
    end

    subgraph nachher["Nach dem Commit"]
        H2["HEAD\nref: refs/heads/main"]
        M2["main\nx9y8z7w6"]
        C2["Neuer Commit\nx9y8z7w6"]
        C2_parent["Alter Commit\na1b2c3d4"]
        C2 -->|parent| C2_parent
    end

    vorher -->|"git commit"| nachher
```

```
    style C2 fill:#c8e6c9,color:#000000
    style M2 fill:#c8e6c9,color:#000000
```

Wichtig: HEAD selbst ändert sich nicht – es zeigt weiterhin auf `refs/heads/main`. Aber die Datei `refs/heads/main` enthält jetzt einen **neuen Hash**.

Alles zusammen in PhpStorm □□

PhpStorm macht die Arbeit mit Branches komfortabel, aber es hilft zu wissen, was dahinter passiert:

Branches anzeigen und verstehen

1. Klicke unten rechts in der Statusleiste auf den **Branch-Namen** (z.B. „main")
2. Du siehst eine Liste aller lokalen und Remote-Branches
3. Jeder Eintrag entspricht einer Datei in `.git/refs/`

Branch erstellen in PhpStorm

1. **Rechtsklick auf einen Commit** im Git-Log → *New Branch...*
2. **Oder:** Branch-Menü → + *New Branch*
3. PhpStorm erstellt die entsprechende Datei in `.git/refs/heads/`

HEAD und aktuellen Branch sehen

- Der **aktuelle Branch** ist in der Statusleiste unten rechts sichtbar
- Im **Git-Log** (Alt+9 → Log-Tab) ist der aktuelle Branch fett markiert und HEAD wird angezeigt
- Bei einem „Detached HEAD"-Zustand zeigt PhpStorm eine Warnung

Refs direkt untersuchen

Du kannst das Terminal in PhpStorm (Alt+F12) nutzen, um die Ref-Dateien direkt zu untersuchen:

```
# In PhpStorms Terminal:
cat .git/HEAD
cat .git/refs/heads/main
```

```
git show-ref --heads
```

Zusammenfassung der wichtigsten Konzepte

Konzept	Was es ist	Wo gespeichert	Verhalten
Referenz (ref)	Lesbarer Name für einen Commit-Hash	`.git/refs/`	Basis für Branches und Tags
Branch	Beweglicher Zeiger auf einen Commit	`.git/refs/heads/<name>`	Wandert mit bei neuen Commits
Remote-Tracking-Branch	Abbild eines Remote-Branches	`.git/refs/remotes/<remote>/<name>`	Wird bei fetch/pull aktualisiert
Tag	Fester Zeiger auf einen Commit	`.git/refs/tags/<name>`	Bleibt immer am selben Commit
HEAD	Zeiger auf aktuellen Branch	`.git/HEAD`	Bestimmt „wo bin ich jetzt"
Detached HEAD	HEAD zeigt direkt auf Commit	`.git/HEAD` (enthält Hash)	Gefährlich – neue Commits können verloren gehen

---

Praktische Übung □□

Probiere folgende Befehle in einem Test-Repository aus, um das Gelernte zu festigen:

```
# 1. Untersuche dein aktuelles Repository
cat .git/HEAD
ls -la .git/refs/heads/
cat .git/refs/heads/main

# 2. Erstelle einen neuen Branch und beobachte die Änderungen
git branch test-branch
ls -la .git/refs/heads/
cat .git/refs/heads/test-branch

# 3. Wechsle den Branch und beobachte HEAD
git switch test-branch
```

```
cat .git/HEAD

# 4. Mache einen Commit und beobachte die Branch-Referenz
echo "test" > test.txt
git add test.txt
git commit -m "Test commit"
cat .git/refs/heads/test-branch  # Neuer Hash!

# 5. Aufräumen
git switch main
git branch -d test-branch
```

Wenn du diese Übung durchgeführt hast, wirst du Gits Referenz-System nie wieder als mysteriös empfinden! □□

Revision #1
Created 2026-06-29 15:21:39 UTC by art10m
Updated 2026-06-29 15:22:14 UTC by art10m