git init und das .git-Verzeichnis

Der Moment, in dem ein Repository entsteht

Bevor Git eine einzige Datei verfolgen kann, muss es einen Ort haben, an dem es seine gesamte Buchführung ablegt. Genau das leistet der Befehl `git init`: Er verwandelt ein gewöhnliches Verzeichnis in ein *Git-Repository*, indem er darin ein verstecktes Unterverzeichnis namens `.git` anlegt. Dieses Verzeichnis ist keine Nebensächlichkeit – es *ist* das Repository. Alles, was du später über Commits, Branches, Remotes oder die Historie erfährst, ist letztlich nur eine strukturierte Sammlung von Dateien innerhalb dieses einen Ordners.

In diesem Abschnitt öffnest du das `.git`-Verzeichnis bewusst, statt es als Blackbox zu behandeln. Das schafft ein solides mentales Modell für alles Folgende.

`git init` ausführen

Wechsle in Git Bash oder PowerShell in dein Übungsprojekt und führe aus:

```
cd C:\Users\DeinName\Projekte\uebungsprojekt
git init
```

Die Ausgabe sieht sinngemäß so aus:

```
Initialized empty Git repository in C:/Users/DeinName/Projekte/uebungsprojekt/.git/
```

Wichtig ist die Formulierung *„empty Git repository“*. Es wurden noch keine Dateien verfolgt, kein Commit erstellt – lediglich das Grundgerüst wurde angelegt. Ein Repository existiert also bereits, bevor überhaupt ein Commit gemacht wurde.

> **Hinweis:** Seit neueren Git-Versionen kannst du den initialen Branch direkt festlegen, etwa mit `git init -b main`. Das ist besonders unter Windows sinnvoll, da die Standardeinstellung je nach Konfiguration variieren kann (siehe Kapitel 2, Abschnitt *„Standard-Branch und Standard-Editor festlegen“*).

Das `.git`-Verzeichnis sichtbar machen

Unter Windows 11 sind versteckte Dateien standardmäßig ausgeblendet. Um `.git` im Windows-Explorer zu sehen, aktiviere unter *Ansicht → Ausgeblendete Elemente* die Anzeige versteckter Ordner. Schneller geht es über die Kommandozeile:

```
ls -la
```

In PowerShell:

```
Get-ChildItem -Force
```

Du solltest neben deinen Projektdateien einen Eintrag `.git` sehen – ein Verzeichnis, kein Symlink, keine spezielle Systemdatei. Das ist bewusst so gestaltet: Git funktioniert vollständig lokal und plattformunabhängig, ohne Betriebssystem-Dienste oder Datenbanken.

Struktur des `.git`-Verzeichnisses

Wirf einen Blick in den Inhalt:

```
ls -la .git
```

Ein frisch initialisiertes Repository enthält typischerweise folgende Elemente:

Element	Bedeutung
`HEAD`	Zeigt symbolisch auf den aktuell aktiven Branch
`config`	Repository-spezifische Konfiguration
`description`	Beschreibungstext, relevant für ältere Web-Oberflächen wie GitWeb
`hooks/`	Vorlagen für clientseitige und serverseitige Hooks
`info/`	Lokale Ausschlussregeln und Metainformationen
`objects/`	Der eigentliche Objektspeicher für Blobs, Trees und Commits

Element	Bedeutung
`refs/`	Referenzen auf Branches und Tags

Zum aktuellen Zeitpunkt sind `objects/` und `refs/` noch weitgehend leer – schließlich wurde noch kein Inhalt committet. Das ändert sich, sobald du im nächsten Abschnitt deinen ersten Commit erstellst.

`HEAD`: der Zeiger auf die Gegenwart

Öffne die Datei `HEAD` in einem Texteditor oder gib ihren Inhalt aus:

```
cat .git/HEAD
```

Die Ausgabe lautet üblicherweise:

```
ref: refs/heads/main
```

Das bedeutet: `HEAD` verweist nicht direkt auf einen Commit, sondern *symbolisch* auf einen Branch – konkret auf `refs/heads/main`. Da dieser Branch noch keinen Commit besitzt, existiert die referenzierte Datei `refs/heads/main` momentan noch gar nicht. Erst der erste Commit erzeugt sie. Dieses Verhalten ist ein gutes Beispiel dafür, dass Git mit *Verweisen auf Verweise* arbeitet – ein Konzept, das in Kapitel 23 vertieft wird.

`config`: lokale Einstellungen

Die Datei `.git/config` enthält repository-spezifische Konfiguration, die Vorrang vor globalen Einstellungen hat:

```
cat .git/config
```

Direkt nach `git init` findest du dort meist nur einen minimalen Abschnitt:

```
[core]
    repositoryformatversion = 0
    filemode = false
    bare = false
    symlinks = false
    ignorecase = true
```

Bemerkenswert für Windows-Nutzer sind besonders `filemode = false` und `ignorecase = true` – beide spiegeln Eigenschaften des NTFS-Dateisystems wider und werden in Kapitel 29 im Detail

erklärt.

`objects/` und `refs/`: die beiden tragenden Säulen

Auch wenn diese Verzeichnisse aktuell leer wirken, lohnt sich ein kurzer Blick:

```
ls .git/objects
ls .git/refs
```

`objects/` wird später sämtliche Inhalte deines Projekts als inhaltsadressierte, unveränderliche Objekte speichern – Kapitel 22 widmet sich diesem *Content-Addressable Storage* ausführlich. `refs/` enthält die Unterordner `heads` und `tags`, die Branches beziehungsweise Tags als einfache Textdateien mit einem Commit-Hash abbilden.

Visualisierung des Zusammenhangs

Das folgende Diagramm zeigt, wie die zentralen Bestandteile direkt nach `git init` zusammenhängen:

```
graph TD
    A["Arbeitsverzeichnis"] -->|git init| B[".git Verzeichnis"]
    B --> C["HEAD - verweist auf refs/heads/main"]
    B --> D["config - lokale Einstellungen"]
    B --> E["objects - noch leer"]
    B --> F["refs/heads - noch leer"]
    C -.->|symbolischer Verweis| F

    style B fill:#2f3b52,color:#ffffff
    style C fill:#3b5b3b,color:#ffffff
```

Der gestrichelte Pfeil verdeutlicht, dass `HEAD` lediglich *symbolisch* auf `refs/heads/main` zeigt – ein Ziel, das erst mit dem ersten Commit real wird.

Ein bestehendes Verzeichnis erneut initialisieren

Führst du `git init` in einem Verzeichnis aus, das bereits ein Repository ist, richtet Git keinen Schaden an. Der Befehl ist *idempotent*: Bestehende Konfiguration, Objekte und Referenzen bleiben unverändert, Git meldet lediglich eine Reinitialisierung:

```
Reinitialized existing Git repository in ...
```

Das ist nützlich, wenn du beispielsweise versehentlich gelöschte Vorlagendateien in `hooks/` wiederherstellen möchtest, ohne die eigentliche Historie zu gefährden.

Ein Repository wieder entfernen

Da das gesamte Repository ausschließlich im `.git`-Verzeichnis lebt, genügt dessen Löschung, um alle Versionsinformationen vollständig und unwiderruflich zu entfernen – deine eigentlichen Projektdateien bleiben dabei unberührt:

```
Remove-Item -Recurse -Force .git
```

Dies solltest du nur bewusst und niemals in einem produktiven Projekt ohne Sicherung ausführen. Es demonstriert aber sehr eindringlich, dass Git keine externe Datenbank, sondern schlicht ein gut organisiertes Dateisystem nutzt.

Blick in PhpStorm

Öffnest du das gerade initialisierte Verzeichnis in PhpStorm, erkennt die IDE das Repository automatisch anhand des `.git`-Ordners und aktiviert die VCS-Funktionen im Menü *Git*. PhpStorm blendet `.git` standardmäßig aus der Projektübersicht aus, da es sich um Metadaten und nicht um Projektinhalt handelt. Über *Einstellungen → Editor → File Types* lässt sich diese Ausblendung zwar anpassen, für den Alltag ist sie jedoch sinnvoll: Du arbeitest mit den komfortablen Werkzeugen der IDE, während das rohe `.git`-Verzeichnis unangetastet im Hintergrund verwaltet wird.

Zwischenfazit

Mit `git init` hast du kein „Werkzeug gestartet“, sondern eine konkrete, inspizierbare Datenstruktur erzeugt. `HEAD`, `config`, `objects/` und `refs/` sind keine abstrakten Begriffe, sondern reale Dateien und Verzeichnisse, die du jederzeit öffnen kannst. Dieses Verständnis trägt dich durch den gesamten Kurs: Jeder spätere Befehl – ob `commit`, `branch`, `merge` oder `reset` – verändert am Ende nichts anderes als genau diese Dateien im `.git`-Verzeichnis.

Im nächsten Abschnitt nutzt du `git status`, um den aktuellen Zustand deines frisch initialisierten Repositorys systematisch zu lesen, bevor du die erste Datei zur Nachverfolgung vormerkst. □□

Revision #1
Created 2026-07-26 17:11:30 UTC by art10m

Updated 2026-07-26 17:11:45 UTC by art10m