Lange Pfade und Dateirechte unter Windows

Warum Windows bei Pfaden und Dateirechten besondere Aufmerksamkeit braucht

Git wurde ursprünglich in einer Unix-Umgebung entwickelt. Windows 11 unterstützt Git hervorragend, unterscheidet sich jedoch bei zwei Themen grundlegend von Linux und macOS:

- **Pfadlängen:** Tiefe Verzeichnisstrukturen können technische Längenlimits überschreiten.
- **Dateirechte:** Windows nutzt primär NTFS-Berechtigungen statt des Unix-Ausführungsbits.

Diese Unterschiede werden besonders sichtbar, wenn ein Team plattformübergreifend arbeitet, etwa mit Windows-Entwicklung, Linux-Containern und einer CI-Pipeline auf GitHub Actions.

> **Ziel:** Das Repository soll auf jedem System zuverlässig klonbar sein, ohne dass lokale Windows-Einstellungen versehentlich Änderungen an Dateirechten erzeugen.

Lange Pfade verstehen

Ein Dateipfad setzt sich aus Laufwerk, Verzeichnissen und Dateinamen zusammen, zum Beispiel:

```
C:\Users\Anna\source\mein-projekt\vendor\beispiel\paket\src\Komponente\SehrLangerDateiname.php
```

Historisch sind viele Windows-Programme auf eine maximale Pfadlänge von **260 Zeichen** ausgerichtet. Dieses klassische Limit wird oft als `MAX_PATH` bezeichnet.

Ein PHP-Projekt kann dieses Limit schnell erreichen:

- Ein langer Benutzername und ein tiefes Arbeitsverzeichnis.
- Verschachtelte Composer-Abhängigkeiten in `vendor`.
- Lange Namespace- und Klassennamen.
- Generierte Dateien, Caches oder Testdaten.
- Monorepos mit vielen Ebenen.

Ein problematischer Pfad könnte beispielsweise so aussehen:

```
C:\Users\Anna\Documents\Kunden\SehrWichtigesUnternehmensprojekt\backend\vendor\anbieter\framework\src\Component\Configuration\Generated\VeryLongGeneratedConfigurationFileName.php
```

Typische Symptome

Wenn Git oder ein anderes Werkzeug mit zu langen Pfaden scheitert, erscheinen unter anderem Meldungen wie:

```
Filename too long
```

```
error: unable to create file ...: Filename too long
```

```
Path too long
```

Mögliche Folgen sind:

- `git clone` bricht unvollständig ab.
- `git checkout` kann einzelne Dateien nicht erzeugen.
- `git pull` oder ein Branch-Wechsel scheitert.
- Composer kann Abhängigkeiten nicht vollständig installieren.
- PhpStorm zeigt Dateien als fehlend oder nicht synchronisiert an.

Unterstützung für lange Pfade in Windows aktivieren

Windows 11 kann lange Pfade grundsätzlich unterstützen. Damit dies zuverlässig funktioniert, müssen jedoch sowohl Windows als auch die jeweilige Anwendung dafür vorbereitet sein.

Lange Win32-Pfade in Windows aktivieren

Die Einstellung ist in Windows 11 Pro, Enterprise und Education über Gruppenrichtlinien erreichbar.

1. Öffne das Startmenü.
2. Suche nach `gpedit.msc`.
3. Starte den **Editor für lokale Gruppenrichtlinien**.
4. Navigiere zu:

   ```
   Computerkonfiguration
   > Administrative Vorlagen
   > System
   > Dateisystem
   ```

5. Öffne die Richtlinie **„Win32 lange Pfade aktivieren“**.
6. Wähle **„Aktiviert“**.
7. Starte Windows neu.

In Windows 11 Home ist der Gruppenrichtlinien-Editor normalerweise nicht verfügbar. Die Einstellung kann stattdessen über die Registrierung gesetzt werden.

> ⚠ Änderungen an der Windows-Registrierung sollten nur bewusst und mit den nötigen Berechtigungen durchgeführt werden.

Öffne PowerShell **als Administrator** und führe aus:

```
New-ItemProperty `
  -Path "HKLM:\SYSTEM\CurrentControlSet\Control\FileSystem" `
  -Name "LongPathsEnabled" `
  -Value 1 `
  -PropertyType DWORD `
  -Force
```

Danach Windows neu starten.

Den aktuellen Wert kannst du prüfen:

```
Get-ItemProperty `
  -Path "HKLM:\SYSTEM\CurrentControlSet\Control\FileSystem" `
  -Name "LongPathsEnabled"
```

Ein aktivierter Wert sieht so aus:

```
LongPathsEnabled : 1
```

Lange Pfade für Git aktivieren

Zusätzlich muss Git for Windows angewiesen werden, lange Pfade zu akzeptieren:

```
git config --global core.longpaths true
```

Die Einstellung überprüfen:

```
git config --global --get core.longpaths
```

Erwartete Ausgabe:

```
true
```

Wenn Git für alle Benutzer des Computers konfiguriert werden soll, verwende eine administrative Konsole:

```
git config --system core.longpaths true
```

Die Herkunft aller gesetzten Werte zeigt:

```
git config --show-origin --get-all core.longpaths
```

Beispielausgabe:

```
file:C:/Program Files/Git/etc/gitconfig    true
```

> **Wichtig:** `core.longpaths` ist eine lokale Git-Client-Einstellung. Sie wird nicht in das Repository übernommen und betrifft keine anderen Teammitglieder automatisch.

Die beste Prävention: kurze lokale Projektpfade

Die Aktivierung langer Pfade ist sinnvoll, aber sie löst nicht jedes Problem. Nicht jede Anwendung, jedes Archivierungswerkzeug oder jede IDE-Komponente unterstützt lange Pfade vollständig.

Die wirksamste Prävention ist deshalb ein **kurzer Projektstammordner**.

Ungünstige Struktur

```
C:\Users\Anna\Documents\Projekte\Kunden\Unternehmen\Produkt\Backend\Repository
```

Günstige Struktur

```
C:\src\produkt
```

oder:

```
D:\code\produkt
```

Ein kurzer Basisordner schafft viel Reserve für Abhängigkeiten, generierte Dateien und tief verschachtelte Testdaten.

Für diesen Kurs bietet sich beispielsweise folgende Struktur an:

```
C:\src\git-kurs
```

Ein neues Repository klonst du dann etwa so:

```
cd /c/src
git clone git@github.com:beispielorganisation/git-kurs.git
```

Das Ergebnis:

```
C:\src\git-kurs
```

Empfehlungen für Teams

- Lege einen dokumentierten Arbeitsordner fest, etwa `C:\src` oder `D:\code`.
- Vermeide sehr lange Repository-Namen.
- Halte Verzeichnisnamen verständlich, aber kompakt.
- Prüfe generierte Ausgaben auf unnötig tiefe Strukturen.
- Versioniere `vendor` in PHP-Projekten in der Regel **nicht**. Composer erzeugt den Ordner lokal oder in der CI.
- Berücksichtige lange Pfade bei ZIP-Archiven, Deployment-Werkzeugen und Build-Prozessen.

Pfadlängen gezielt untersuchen

PowerShell kann lange Dateipfade im aktuellen Projekt finden. Wechsle zunächst in das Repository:

```
Set-Location C:\src\git-kurs
```

Anschließend lassen sich besonders lange Pfade auflisten:

```
Get-ChildItem -Recurse -Force |
  Sort-Object { $_.FullName.Length } -Descending |
  Select-Object -First 20 FullName, @{Name="Laenge"; Expression={ $_.FullName.Length }}
```

Damit erhältst du die zwanzig längsten Pfade samt Zeichenanzahl.

Um nur Pfade ab 240 Zeichen anzuzeigen:

```
Get-ChildItem -Recurse -Force |
  Where-Object { $_.FullName.Length -ge 240 } |
  Select-Object FullName, @{Name="Laenge"; Expression={ $_.FullName.Length }}
```

Die Grenze von 240 Zeichen ist bewusst etwas niedriger als das klassische Limit. Der verbleibende Spielraum schützt vor zusätzlichen Zeichen, die Werkzeuge temporär erzeugen können.

Dateirechte: Windows und Unix denken unterschiedlich

Auf Linux und macOS besitzen Dateien klassische Unix-Modi. Besonders wichtig ist dabei das **Ausführungsbit**:

```
-rwxr-xr-x
```

Die drei Gruppen stehen für:

- Eigentümer
- Gruppe
- andere Benutzer

Das `x` bedeutet: Die Datei darf als Programm oder Skript ausgeführt werden.

Unter Windows steuern dagegen NTFS-Berechtigungen den Zugriff. Diese sind wesentlich detaillierter und beziehen sich auf Benutzer, Gruppen und Zugriffsarten wie Lesen, Schreiben oder Ändern.

Eine typische Windows-Datei besitzt daher nicht einfach ein Unix-Ausführungsbit. Ob eine Datei gestartet werden kann, hängt unter anderem von Folgendem ab:

- Dateiendung, etwa `.exe`, `.bat`, `.cmd` oder `.ps1`
- NTFS-Berechtigungen
- PowerShell-Ausführungsrichtlinien
- Zuordnung zu installierten Programmen
- Sicherheitsmechanismen wie Microsoft Defender SmartScreen

Git kann die vollständigen Windows-ACLs nicht sinnvoll und plattformübergreifend versionieren. Stattdessen speichert Git im Normalfall nur eine vereinfachte Dateimodus-Information.

Das Ausführungsbit in Git

Git unterscheidet bei regulären Dateien vor allem zwischen zwei Modi:

```
100644
```

Dies beschreibt eine normale, **nicht ausführbare** Datei.

```
100755
```

Dies beschreibt eine **ausführbare** Datei.

Weitere häufige Modi sind:

```
040000
```

Ein Verzeichnis beziehungsweise ein Tree-Objekt.

```
120000
```

Ein symbolischer Link.

Die Modusangaben sind Teil des Git-Index und damit Teil eines Commits. Deshalb bleiben sie beim Klonen und zwischen Betriebssystemen erhalten.

Ein Shell-Skript kann unter Linux beispielsweise ausführbar sein:

```
scripts/deploy.sh
```

In Git wäre es dann als `100755` gespeichert. Ein Windows-Entwickler kann den Inhalt bearbeiten und committen, ohne das Ausführungsbit zwangsläufig zu verändern — sofern Git passend konfiguriert ist.

`core.filemode` unter Windows

Die Einstellung `core.filemode` legt fest, ob Git Änderungen am Ausführungsbit im Arbeitsverzeichnis erkennen soll.

Prüfe den aktuellen Wert:

```
git config --get core.filemode
```

Auf Windows ist häufig dieser Wert sinnvoll:

```
false
```

Falls kein Wert ausgegeben wird, prüfe alle Konfigurationsebenen:

```
git config --show-origin --get-all core.filemode
```

Mit folgendem Befehl ignoriert Git lokale Änderungen am Ausführungsbit:

```
git config --global core.filemode false
```

Das ist für die meisten Windows-Arbeitsplätze empfehlenswert, weil NTFS und Git Bash Dateimodi nicht immer so abbilden wie ein Linux-Dateisystem.

> **Wichtig:** `core.filemode false` entfernt keine bereits versionierten Ausführungsrechte. Es verhindert lediglich, dass Git lokale Änderungen an diesem Bit als Arbeitsverzeichnisänderung meldet.

Beispiel: Unerwartete Modusänderung

Angenommen, ein Skript wurde auf einem Linux-System ausführbar gemacht. Git zeigt dann unter Umständen:

```
old mode 100644
new mode 100755
```

Oder umgekehrt:

```
old mode 100755
new mode 100644
```

Wenn diese Änderung beabsichtigt ist, kann sie normal committed werden. Wenn sie nur durch die lokale Umgebung entstanden ist, sollte sie nicht versehentlich in einen Commit gelangen.

Prüfe stets zuerst den Status:

```
git status
```

Untersuche anschließend den Unterschied:

```
git diff --summary
```

Eine reine Rechteänderung erscheint beispielsweise so:

```
mode change 100644 => 100755 scripts/deploy.sh
```

Ausführungsrechte bewusst mit Git setzen

Wenn ein Skript auf Linux-Servern, in Docker-Containern oder in GitHub Actions direkt ausgeführt werden soll, setze das Ausführungsbit explizit in Git.

Datei ausführbar machen

```
git update-index --chmod=+x scripts/deploy.sh
git commit -m "chore: make deployment script executable"
```

Ausführungsbit entfernen

```
git update-index --chmod=-x scripts/deploy.sh
git commit -m "chore: remove executable permission from deployment script"
```

Kontrolliere die Änderung:

```
git diff --cached --summary
```

Beispiel:

```
mode change 100644 => 100755 scripts/deploy.sh
```

Diese Befehle sind besonders nützlich, weil sie das gewünschte Git-Metadatum direkt setzen — unabhängig davon, wie Windows die Datei im lokalen Dateisystem behandelt.

Rechte eines bereits committeten Pfads prüfen

```
git ls-tree HEAD scripts/deploy.sh
```

Mögliche Ausgabe:

```
100755 blob 1234567890abcdef1234567890abcdef12345678    scripts/deploy.sh
```

Die führende Angabe `100755` bestätigt, dass Git die Datei als ausführbar speichert.

Relevanz für PHP-Projekte

PHP-Dateien benötigen unter Windows normalerweise kein Ausführungsbit. Sie werden über den PHP-Interpreter gestartet:

```
php bin/console
```

Auf Linux kann dieselbe Datei jedoch direkt ausführbar sein:

```
./bin/console
```

Damit die zweite Variante funktioniert, braucht die Datei in der Regel:

1. Eine Shebang-Zeile, etwa:
   ```
   #!/usr/bin/env php
   ```
2. Das Ausführungsbit `100755` in Git.

Das gilt häufig für:

- CLI-Werkzeuge in `bin/`
- Deployment-Skripte in `scripts/`
- Shell-Skripte mit der Endung `.sh`
- Entwicklerwerkzeuge und Generatoren
- CI-Hilfsskripte

Für eine reine PHP-Klassendatei wie `src/Service/UserService.php` wäre ein Ausführungsbit dagegen ungewöhnlich und sollte geprüft werden.

Windows-Attribute sind keine Git-Dateirechte

Windows kennt Attribute wie:

- schreibgeschützt

- versteckt
- Systemdatei
- Archiv

Diese Attribute sind nicht mit Unix-Dateirechten gleichzusetzen. Git versioniert sie im Normalfall nicht.

Ein schreibgeschütztes Attribut kann jedoch lokale Git-Operationen stören, etwa beim Auschecken, Zurücksetzen oder Wechseln eines Branches. Wenn Git eine Datei nicht überschreiben kann, prüfe zunächst die Windows-Eigenschaften der Datei.

In PowerShell:

```
Get-Item .\config\settings.php | Select-Object Name, Attributes
```

Ein schreibgeschütztes Attribut lässt sich entfernen:

```
attrib -R .\config\settings.php
```

Für ein gesamtes Verzeichnis einschließlich Unterordnern:

```
attrib -R .\temp\* /S /D
```

> ⚠ Entferne Attribute nur in Projektordnern, deren Inhalt du kennst. Systemordner und fremde Verzeichnisse sollten nicht pauschal verändert werden.

NTFS-Berechtigungen bei Zugriffsproblemen prüfen

Wenn Git Fehler wie „Permission denied“, „Access is denied“ oder „unable to unlink“ meldet, liegt die Ursache häufig nicht bei Git selbst. Häufige Gründe sind:

- Eine Datei ist in PhpStorm, einem Terminal oder einem anderen Programm gesperrt.
- Ein Virenscanner untersucht oder blockiert die Datei.
- Der Projektordner gehört einem anderen Windows-Benutzer.
- Das Repository liegt in einem geschützten Ordner.

- Der Ordner befindet sich in OneDrive, einem Netzlaufwerk oder einem synchronisierten Unternehmensspeicher.
- Die NTFS-Berechtigungen erlauben keinen Schreibzugriff.

Zeige die Berechtigungen eines Ordners an:

```
Get-Acl C:\src\git-kurs | Format-List
```

Alternativ:

```
icacls C:\src\git-kurs
```

Für persönliche Entwicklungsprojekte ist ein lokaler Ordner wie `C:\src` meist robuster als:

```
C:\Program Files
```

```
C:\Windows
```

```
C:\Users\Name\OneDrive\Dokumente
```

Geschützte Systemordner und synchronisierte Ordner erhöhen die Wahrscheinlichkeit für Berechtigungs-, Sperr- und Synchronisationsprobleme.

Dateisperren unter Windows erkennen

Windows erlaubt es Anwendungen häufig, Dateien exklusiv zu sperren. Das unterscheidet sich von vielen Linux-Workflows, bei denen Dateien während der Bearbeitung leichter ersetzt oder gelöscht werden können.

Wenn Git eine Datei nicht löschen oder überschreiben kann:

1. Schließe geöffnete Editoren, Vorschauprogramme und Terminals.
2. Beende gegebenenfalls PhpStorm vollständig.
3. Prüfe laufende PHP-, Node.js-, Composer- oder Testprozesse.
4. Warte kurz, falls Microsoft Defender die Datei prüft.
5. Wiederhole erst danach den Git-Befehl.

Bei hartnäckigen Sperren kann der Windows-Ressourcenmonitor helfen:

1. Öffne den Task-Manager.
2. Wähle **Leistung**.
3. Öffne den **Ressourcenmonitor**.
4. Wechsle zu **CPU**.
5. Suche im Bereich **Zugeordnete Handles** nach dem Dateinamen.

So lässt sich feststellen, welcher Prozess eine Datei oder ein Verzeichnis verwendet.

PhpStorm und Dateirechte

PhpStorm zeigt Änderungen an Dateiinhalten und Git-Metadaten im Commit-Werkzeugfenster an. Eine Modusänderung kann dort als eigener Änderungstyp erscheinen.

Vor einem Commit solltest du daher nicht nur die Dateinamen, sondern auch die Diff-Ansicht kontrollieren:

1. Öffne das Werkzeugfenster **Commit**.
2. Wähle die geänderte Datei.
3. Prüfe, ob neben Inhaltsänderungen auch eine Modusänderung vorliegt.
4. Übernimm Rechteänderungen nur, wenn sie fachlich gewollt sind.

Für Shell-Skripte und Linux-CLI-Werkzeuge ist die Kommandozeile oft eindeutiger:

```
git diff --summary
```

```
git ls-tree HEAD -- scripts/deploy.sh
```

PhpStorm kann Windows-NTFS-Berechtigungen nicht als plattformübergreifende Git-Regel verwalten. Die maßgebliche Information für das Repository bleibt der Git-Dateimodus.

Empfohlene Git-Konfiguration für einen Windows-Entwicklungsrechner

Für eine typische lokale Entwicklungsumgebung unter Windows 11 sind diese Einstellungen ein sinnvoller Ausgangspunkt:

```
git config --global core.longpaths true
git config --global core.filemode false
```

Prüfe beide Werte:

```
git config --global --get core.longpaths
git config --global --get core.filemode
```

Erwartete Ausgabe:

```
true
false
```

Diese Konfiguration bedeutet:

- Git akzeptiert lange Pfade, soweit Windows und die verwendeten Werkzeuge dies unterstützen.
- Git meldet lokale Änderungen am Ausführungsbit nicht unnötig als Änderung.
- Bereits im Repository gespeicherte Ausführungsrechte bleiben erhalten.
- Bewusste Rechteänderungen können weiterhin mit `git update-index --chmod` vorgenommen werden.

Diagnoseablauf bei Problemen

Wenn ein Checkout, Pull oder Branch-Wechsel auf Windows scheitert, arbeite strukturiert statt Befehle mehrfach zu wiederholen.

Bei einem Pfadlängenfehler

1. Prüfe die Git-Einstellung:
   ```
   git config --get core.longpaths
   ```
2. Aktiviere sie bei Bedarf:
   ```
   git config --global core.longpaths true
   ```
3. Prüfe, ob lange Pfade in Windows aktiviert sind.

4. Klone oder verschiebe das Repository in einen kurzen Pfad, etwa:

```
C:\src\projekt
```

5. Prüfe besonders lange Pfade mit PowerShell.
6. Aktualisiere bei Bedarf Git for Windows und PhpStorm.

Bei einem Rechte- oder Zugriffsfehler

1. Prüfe den Repository-Zustand:

```
git status
```

2. Schließe Programme, die Dateien sperren könnten.
3. Prüfe, ob das Projekt in OneDrive oder einem geschützten Ordner liegt.
4. Prüfe NTFS-Berechtigungen:

```
icacls C:\src\projekt
```

5. Prüfe schreibgeschützte Attribute:

```
Get-Item .\betroffene-datei | Select-Object Name, Attributes
```

6. Prüfe bei reinen Modusänderungen:

```
git diff --summary
```

7. Setze bei Bedarf das Ausführungsbit gezielt über Git statt über Windows-Dateieigenschaften.

Praxisübung: Umgebung absichern

Lege ein kurzes lokales Arbeitsverzeichnis an:

```
New-Item -ItemType Directory -Path C:\src -Force
```

Wechsle dorthin:

```
Set-Location C:\src
```

Aktiviere die relevanten Git-Einstellungen:

```
git config --global core.longpaths true
git config --global core.filemode false
```

Prüfe die Konfiguration einschließlich ihrer Herkunft:

```
git config --show-origin --get-all core.longpaths
git config --show-origin --get-all core.filemode
```

Erstelle ein kleines Test-Repository:

```
mkdir git-windows-test
cd git-windows-test
git init
```

Lege ein Skript an:

```
echo echo Deployment > deploy.sh
```

Mache es bewusst ausführbar:

```
git add deploy.sh
git update-index --chmod=+x deploy.sh
git commit -m "chore: add executable deployment script"
```

Prüfe abschließend den gespeicherten Dateimodus:

```
git ls-tree HEAD deploy.sh
```

Die Ausgabe sollte mit `100755` beginnen.

Merkpunkte

- **Kurze Projektpfade** sind die zuverlässigste Vorbeugung gegen Pfadprobleme.
- Aktiviere lange Pfade sowohl in **Windows** als auch in **Git** mit `core.longpaths`.
- Verwende für lokale Projekte vorzugsweise Pfade wie `C:\src\projekt`.
- Git speichert keine vollständigen Windows-NTFS-Berechtigungen.
- Für Git ist vor allem das Unix-Ausführungsbit relevant: `100644` oder `100755`.
- Setze Ausführungsrechte für plattformübergreifende Skripte bewusst mit `git update-index --chmod`.
- Mit `core.filemode false` vermeidest du auf Windows unnötige lokale Modusänderungen.

- Bei „Access denied“ sind Dateisperren, Virenscanner, OneDrive oder NTFS-Berechtigungen oft die eigentliche Ursache.

Revision #1
Created 2026-07-25 11:13:47 UTC by art10m
Updated 2026-07-25 11:13:47 UTC by art10m