Spickzettel („Cheat Sheet“): Versionsverwaltung mit Git (in Windows PowerShell / PowerShell Core)

Autor: Dr. Holger Schwichtenberg (www.IT-Visions.de)

www.IT-Visions.de
Dr. Holger Schwichtenberg

Installation

```
https://git-scm.com # Git-Website
# Git Credentials Manager for Windows (Anmeldung u.a. für Azure DevOps)
https://github.com/Microsoft/Git-Credential-Manager-for-Windows
# Git-Erweiterungen für PowerShell aus PowerShell-Gallery installieren
Install-Module posh-git -Scope CurrentUser -Force
# Git-Erweiterungen für PowerShell aktivieren in dieser Konsoleninstanz
Import-Module posh-git
```

Allgemeine Informationen

```
# Git-Versionsnummer ausgeben
git version # auch: git --version
# Liste der Hilfethemen
git help
# HTML-Hilfe zu einem Thema zeigen, z.B. Tags
git help tag
```

Globale Einstellungen

```
# Benutzerdaten global einstellen
git config --global user.name "HolgerSchwichtenberg"
git config --global user.email "dev@IT-Visions.de"
# Setze VSCode als Git-Editor (für Commit-Kommentare u.a.)
git config --global core.editor "code --wait"
# Ausschalten der Git-Warnung bez. LF und CR
git config --global core.autocrlf true
# Konfliktstile "3-Wege" (mit Anzeige des gemeinsamen Vorgängers)
git config merge.conflictstyle diff3
# Seitenweises Blättern ausschalten
git config --global core.pager cat
# Alias definieren: git lg für git log mit Parametern
git config --global alias.lg 'log --oneline --graph --decorate -n 10'
# Alle globalen Git-Einstellungen auflisten
git config --list --global
```

Lokales Repository beginnen

```
# Repository im aktuellen Verzeichnis anlegen
git init
# lokale Konfiguration nur für dieses Repository
git config merge.conflictstyle merge #kein 3-Wege-Diff
# lokale Git-Einstellungen auflisten
git config --list --local
# für .NET- und .NET Core-Projekte geeignete .gitignore-Datei herunterladen
iwr -Uri "https://www.gitignore.io/api/aspnetcore" -OutFile ".gitignore"
# LF durch CRLF ersetzen, damit Git sich nicht beschwert
(gc ".gitignore" -Raw).Replace("`n","`r`n") | Set-Content ".gitignore" -Force
```

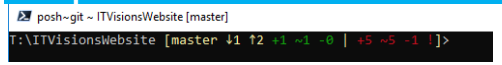
Format für .gitignore

```
log*.txt Alle Dateien mit diesem Muster ausschließen
/bin Alle Dateien im Ordner /bin ausschließen
```

Dateien hinzufügen und ändern

```
# Alle Dateien ins Staging
git add * #oder git add .
# Einzelne neue oder geänderte Datei ins Staging hinzufügen
git add ./Readme.txt
# Datei ins Staging, auch wenn sie durch .gitignore erfasst ist
git add -f ./lib/ITVisionsUtilLib.dll
# Verzeichnis ins Staging hinzufügen (mit allen enthaltenen Dateien)
git add ./Util/Network/HTTP
# Status anzeigen
git status
# Commit des Inhalts von Staging
git commit -m "Mein guter Commit-Kommentar"
# Commit ohne vorheriges Staging (nur für geänderte und gelöschte Dateien!)
git commit -a -m "Bugfix" #-a == --all, -m == --message
# Erweitern des vorherigen Commit um weitere Änderungen
git commit -a --amend -m "Bugfix"
# Eine Datei temporär aus der Änderungsverfolgung herausnehmen
git update-index --assume-unchanged ./Program.cs
# Eine Datei zurück in die Änderungsverfolgung
git update-index --no-assume-unchanged ./Program.cs
# Änderungsverfolgung für alle Dateien reaktivieren
git update-index --really-refresh
# Nur Teile einer Datei ins Staging (interaktive Festlegung)
git add ./Program.cs -p
```

Prompt-Informationen



↓ Remote Commits gegenüber lokal ↑ lokale Commits gegenüber Remote
\+ neue ~ geänderte – gelöschte ! Konflikte Grün Staging Rot Workspace

Dateien umbenennen, verschieben, löschen

```
# Datei umbenennen
git mv urheberrechte.html copyright.html
# Datei verschieben
git mv copyright.txt hilfedateien/copyright.html
# Datei löschen
git rm hilfedateien/copyright.html
# Ordner mit Dateien löschen
git rm hilfedateien -r
```

Änderungen darstellen

```
# Liste der geänderten Dateien
git ls-files -m
# Alle Änderungen für alle Dateien
git diff
# Änderungen nur für eine Datei
git diff ./Program.cs
# Nur Namen und Status der geänderten Dateien
git diff --name-status
# Wer hat wann was geändert?
git blame ./Program.cs
# Details zum letzten Commit / vorvorletzten Commit / bestimmten Commit
git show bzw. git show HEAD~2 bzw. git show 650af40
# Zeige Git Repository Browser GUI
gitk
```

Versionsgeschichte anzeigen

```
# Versionsgeschichte, mehrzeilig
git log
# Versionsgeschichte einzeilig
git log --oneline
# Versionsgeschichte einzeilig mit Branch-Darstellung und Refs (Head, Tags, Remotes), auf letzte zehn Commits begrenzt
git log --oneline --graph --decorate -n 10
# Versionsgeschichte individuell formatiert
git log --pretty=format:"%t %h - %an, %ar: %s"
# Suche Commit mit einem Textteil im Kommentar
git log --grep "Aenderung" --oneline
# Nur Commits zwischen beiden Commit-ID einzeilig auflisten
git log 650af40^..7d4bd60 --oneline
# Zeige alle, unerreichbare Commits (nach einem Reset etc.)
git reflog --all
# Zeige, welche Benutzer die letzten 50 Commits gespeichert haben
git shortlog ...HEAD~50 -s --email
# Zeige Benutzerstatistik mit den einzelnen Commits seit Tag v1.1
git shortlog ...v1.1 --email
```

Platzhalter bei git log

%T Tree Hash lang (SHA-1) %t Tree Hash kurz
git%H Commit Hash lang %h Commit Hash kurz
%an Autorenname %ae E-Mail des Autors
%ad Datum des Autors %ar Datum relativ %s Commit-Kommentar
%cn Committer-Name %ce E-Mail des Committers
%cd Datum des Committers %cr Datum relativ

Verweise auf Commit

HEAD letzter Commit
HEAD^ und HEAD~1 vorletzter Commit
HEAD~5 fünf Commit zurück
...HEAD~20 Die letzten 20 Commits
...650af40 oder 650af40.. von der Commit-ID bis zum aktuellen Commit
650af40 bestimmter Commit anhand der ID
650af40..7d4bd60 Alle nach der ersten Commit-ID bis zur zweiten
650af40^..7d4bd60 Alle von der ersten Commit-ID bis zur zweiten
Anderer Text Tag oder Branchname

Suche

```
# Suche alle Dateien, in denen ein Text vorkommt
git grep --line-number "Console.WriteLine"
# oder rein mit PowerShell-Befehlen:
dir -r | sls "Console.WriteLine"
```

Branching (Verzweigen)

```
# Neuen Branch erstellen auf Basis des aktuellen Workspaces inkl. Staging
git branch Feature1
# Liste der Branches
git branch
# Liste der Branches mit letztem Commit
git branch -v
# Zu Branch "Feature1" wechseln
git checkout Feature1
# Branch wechseln, auch wenn dadurch nicht commitete Änderungen im aktuellen Branch verloren gehen
git checkout Feature1 -f
# Branch erstellen und direkt dorthin wechseln
git checkout -b Feature2
# Name des aktuellen Branches
$branch = (git branch | sls "\* (.*)").Matches[0].Groups[1].Value
# Branch löschen
git branch -d Feature1
# Branch löschen, auch wenn er nicht merged ist!
git branch -D Feature1 # oder -d --force
# Alle Branches auflisten, die nicht merged sind
git branch --no-merged
# Löschen aller Branches, die inaktiv und merged sind und deren Name mit "Feature" beginnt
git branch --merged | sls " (Feature.*)" | % { git branch -D
$_.Matches[0].Groups[1] }
# Löschen aller Branches, außer dem aktuellen
git branch | sls " (.*)" | % { git branch -D $_.Matches[0].Groups[1] }
```

Spickzettel („Cheat Sheet“): Versionsverwaltung mit Git (in Windows PowerShell / PowerShell Core)

Autor: Dr. Holger Schwichtenberg (www.IT-Visions.de)

www.IT-Visions.de
Dr. Holger Schwichtenberg

Merging, Rebasing und Cherry-Picking

```
# Merge aus Feature1 nach master
git checkout master
git merge Feature1
# Merge aus Feature1, bei dem im Konfliktfall immer master gewinnt
git merge -s ours Feature1
# Rebase aus Feature1 nach master
git rebase Feature1
# Rebase aus Feature1, bei dem im Konfliktfall Feature1 gewinnt
git rebase Feature1 -s ours # andere octopus, ours, recursive, resolve, subtree
# Cherry-Picking: Nur diese Commit-ID aus anderem Branch
git cherry-pick 650af40
# Cherry-Picking: Alle Commits von / bis zur genannten Commit-ID
git cherry-pick 7d4bd60^..$7d4bd60
# Cherry-Picking: Alle Commits bis zur genannten Commit-ID
git cherry-pick ...$7d4bd60
```

Konfliktlösung bei Merging, Rebasing, Cherry-Picking

```
# Nach der manuellen Lösung des Konflikts im Editor
git add ./Program.cs
git commit -a -m "Merge aus Feature1, Konflikt gelöst"
# Konfliktlösung abbrechen (zurück zu letztem Commit)
git merge --abort # bzw. git rebase --abort bzw. git cherry-pick --abort
```

Undo (Rückgängig machen)

```
# Workspace zurück auf Stand Staging (wenn leer: HEAD) für alle Dateien
git checkout .
# Workspace zurück auf Stand Staging (wenn leer: HEAD) für eine Datei
git checkout -- ./Program.cs
# Workspace zurück auf Stand HEAD (Staging löschen) für alle Dateien
git checkout HEAD .
# Workspace zurück auf Stand HEAD (Staging löschen) für eine Datei
git checkout HEAD ./Program.cs
# Workspace zurück auf Stand zwei vor HEAD für eine Datei
git checkout HEAD~2 ./Program.cs
# Setze HEAD zwei Schritte zurück
git reset --soft HEAD~2
# Setze HEAD und Staging zwei Schritte zurück
git reset HEAD~2 # optional: --mixed
# Setze HEAD und Staging für eine Datei auf bestimmten Commit zurück
git reset 650af40 ./Program.cs
# Setze HEAD, Staging und Workspace zwei Schritte zurück (für alle Dateien)
git reset --hard HEAD~2
# Setze HEAD, Staging und Workspace auf einen bestimmten Commit zurück
git reset --hard 650af40
```

Lokales Repository veröffentlichen

```
# Azure DevOps als erstes Remote-Repository verbinden
git remote add AZURE https://dev.azure.com/Organisation/Projekt/_git/Repo
# GitHub als zweites Remote-Repository verbinden
git remote add GITHUB https://github.com/Konto/Repo.git
# Liste der Remote Repositories mit Endpunkten
git remote -v
# Details zu einem Remote-Repository
git remote show AZURE
# Push für einen bestimmten Branch mit Setzen des Upstream-Branches
git push AZURE master --set-upstream
# Push für alle Branches, setzt auch Upstream-Branches
git push AZURE --all --set-upstream
# Push für aktuellen Branch zum definierten Upstream-Branch
git push
# Push für einen bestimmten Branch zu einem bestimmten Remote
git push AZURE master
# Push eines bestimmten Branches erzwingen, Konflikte ignorieren
git push -f AZURE master
# Push für bestimmten Branch zu anderem Remote
git push GITHUB master
# Ändern des Upstreams für einen Branch
git branch -u AZURE /master
# Branch-Details inkl. Upstream-Branches und Remote-Tracking-Branches
git branch -vv -a
# Änderungen von Remote-Repository in Remote-Tracking-Branch holen
git fetch AZURE bzw. git fetch AZURE bzw. git fetch GITHUB
# Wechseln in den Remote-Tracking-Branch: Was hat sich im Remote getan?
git checkout GITHUB/master
git log --oneline --graph --decorate
# pull = fetch + merge
git pull AZURE # für aktuellen Upstream-Branch
git pull GITHUB master # für andere Remote-Branches
# Verbindung zu einem Remote entfernen bzw. zu allen Remotes löschen
git remote rm GITHUB bzw. git remote | % { git remote rm $_ }
```

Bestehendes Repository klonen

```
# Klonen mit Standardremotename "origin"
git clone https://dev.azure.com/Organisation/Projekt/_git/Repo
# Klonen mit Remotename "GITHUB", ausführlich
git clone https://dev.azure.com/Organisation/Projekt/_git/Repo -v --progress
-o GITHUB $cloneDir
# Anlegen eines lokalen Branches für alle Remote-Branches
git branch -a | ? { $_ -like "*remotes*" -and $_ -notlike "*HEAD*" }| % { git
branch --track ${remote#origin/} $_.Trim() }
```

Große Repositories mit Virtual File System for Git

```
https://github.com/Microsoft/VFSForGit
# Klonen mit VFS for Git
gvfs clone https://dev.azure.com/Organisation/Projekt/_git/Repo
# Verbindung lösen
gvfs unmount
```

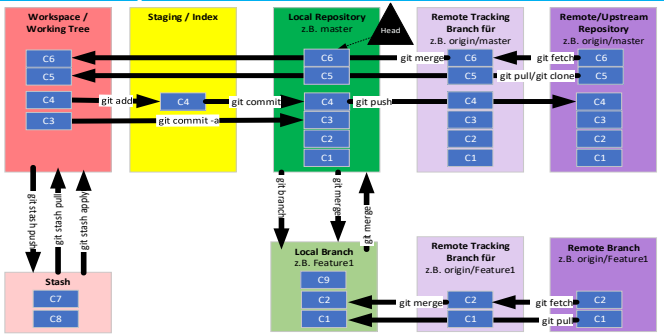
Tags (Etiketten)

```
# Tag für letzten Commit vergeben inklusive Tag-Nachricht
git tag "v1.1" -m "Release-Version 1.1"
# Tag für vorvorletzten Commit vergeben
git tag -a "v1.1" Head~2 -m "Release-Version 1.1"
# Tag verschieben auf vorletzten Commit (auch mit Commit-ID möglich!)
git tag -a "v1.1" Head~1 -m "Release-Version 1.1" -f
# Log mit Tags ausgeben (--decorate)
git log --oneline --graph --decorate
# Liste aller Tags inklusive Tag-Nachricht ausgeben
git tag -l -n
# Liste aller Tags mit v1. ausgeben
git tag -l "v1.*"
# Zeige Commit-IDs zu allen Tags
git show-ref --tags --abbrev
# Ein Tag zum Upstream-Branch übertragen
git push AZURE v1.1
# Alle Tags zum Upstream-Branch übertragen
git push AZURE --tags
# Ein Tag lokal löschen
git tag -d "v1.1"
# Löschen aller lokalen Tags
git tag -d $(git tag -l)
# Ein Tag in Remote löschen
git push AZURE :refs/tags/v1.1
```

Stashing (Bunkern)

```
# Änderungen im Bunker ablegen
git stash push -m "ErsterVersuch"
# Liste der Versionen im Bunker
git stash list
# Letzte Version aus Bunker holen
git stash pop
# Zweites Element aus Bunker holen
git stash pop 'stash@{1}'
# Erstes Bunker-Element anwenden, aber im Bunker belassen
git stash apply 'stash@{0}'
# Bunkereintrag anhand des Namens suchen und anwenden
git stash apply ((git stash list | ? { $_ -like "*ZweiterVersuch*" }) -split ":")[0]
# Letzten Bunkereintrag löschen ohne ihn anzuwenden
git stash drop
# Alle Bunkereinträge löschen
git stash clear
```

Git-Datenspeicher



Dateien aufräumen

```
# Löschen aller Dateien und Ordner, die nicht unter Versionskontrolle sind
git clean  -f -d -x
# Löschen nur der per .gitignore ausgenommenen Dateien und Ordner
git clean  -f -d -X
# Abschalten, dass -f bei clean erforderlich ist
git config --global clean.requireForce false
# Löschen nicht mehr erreichbarer Objekte, die älter als zwei Wochen sind
git gc
# Sofortiges Löschen aller nicht mehr erreichbaren Objekte
git reflog expire --expire-unreachable=now --all
git gc --prune=now
```

Über den Autor

Dr. Holger Schwichtenberg gehört zu den bekanntesten Experten für Webtechniken und .NET in Deutschland. Er hat zahlreiche Fachbücher veröffentlicht und spricht regelmäßig auf Fachkonferenzen. Sie können ihn und seine Kollegen für Entwicklungsarbeiten, Schulungen, Beratungen und Coaching buchen. E-Mail: **buero@IT-Visions.de**
Website: **www.IT-Visions.de** Weblog: **www.dotnet-doktor.de**