# Git The Big Picture



## Workflow

### 1. Main Branches

Es gibt verschiedene Varianten, ein Git-Projekt zu beginnen. Der folgende Beispielworkflow beginnt mit zwei Main Branches:

- Master Branch (***master***)
- Development Branch (***development***)

Beide Main Branches sind von unbegrenzter Dauer.
Die initiale Produktversion startet auf dem Branch *master* und wird im Branch *development* reflektiert. Im Branch *master* wird der Sourcecode von HEAD stets im produktionsfertigen Status angezeigt.

Im Branch *development* wird der Sourcecode von HEAD stets mit den letzten Änderungen für das nächste Release angezeigt.


**Gut zu wissen!**

Eine andere, durchaus verbreitete Workflowvariante wäre, dass konstant auf dem Master Branch gearbeitet wird. Die hier gezeigte Variante trägt durch ihre Aufteilung aber zum besseren Verständnis der einzelnen Workflowschritte bei.

### 2. Releases

Wenn der Sourcecode in *development* einen stabilen Status erreicht hat und bereit ist, ausgeliefert zu werden, wird er mittels eines Merge in den Branch *master* überführt und mit einer Releasenummer ausgezeichnet.

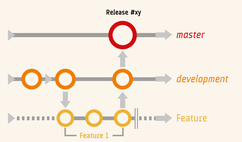
IMMER wenn Änderungen in den Branch *master* gemergt werden, haben wir per Definition ein neues Produktionsrelease (dazu mehr in „Einen Release Branch erstellen").


### 3. Supporting Branches

Supporting Branches erleichtern das parallele Arbeiten der Teammitglieder. Man unterscheidet Feature Branch, Release Branch und Hotfix Branch.

### Feature Branch

Der Feature Branch dient dem Tracking der einzelnen Features: Es gibt u. U. mehrere parallele Feature Branches (jeweils einen Branch pro Feature). Ein Feature Branch existiert, solange das Feature in der Entwicklungsphase ist. Dann wird das Feature in den Branch *development* überführt (merge), sodass es in das nächste Release eingefügt werden kann. Das Feature wird dann wiederum vom Branch *development* in den Branch *master* gemergt.




**Gut zu wissen!**

Branch Naming Convention bei Feature Branches: Alles ist erlaubt, außer *master*, *development*, *release-** und *hotfix-**. Bei der Verwendung spezifischer Software, wie etwa JIRA, erfolgt die Benennung der Branches üblicherweise nach den Issue-IDs.

### Release Branch *(release-*)*

Der Release Branch dient der Vorbereitung für das Production-Release: Kurzfristige Überprüfungen des Releasecodes werden auf einen Release Branch gelegt. Hier können schnelle Bug Fixes (geringerer Form) gemacht und Metadaten für das nächste Release vorbereitet werden. Somit bleibt der Branch *development* frei, um weiterhin Features zu mergen. Der Release Branch wird nach Abschluss eines Fixes/der Vorbereitung aufgelöst. Zuvor wird er sowohl in *development* als auch in *master* gemergt.

### Einen Release Branch erstellen

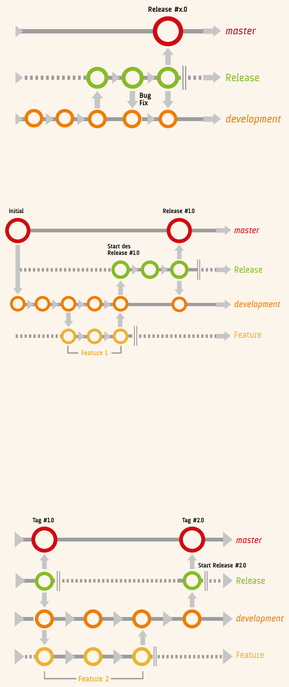
Der Release Branch zweigt vom Branch *development* ab, sobald *development* kurz vor einem neuen Release steht. Zu diesem Zeitpunkt müssen alle Features für das nächste anstehende Release in den Branch *development* gemergt werden.

Der Start eines Release Branch bedeutet automatisch die Vergabe der Versionsnummer des bevorstehenden Release (also z. B. #1.0, 2.0. ... ). Davor reflektierte der Branch *development* zwar bereits die Änderungen des kommenden Release, es ist aber bis dahin noch unklar, ob es sich um eine Version #0.x oder #x.0 handeln wird (vgl. dazu Punkt 2).

**Checkliste**
Bevor ein Release Branch in ein neues Release münden kann:

- ☑ Mergen Sie den Release Branch in den Branch *master* (zur Erinnerung: Jeder Commit ist per Definition ein neues Release!).
- ☑ Vergeben Sie einen Tag (z. B.: 1.0) für den Commit als zukünftige Referenz auf diese Version.
- ☑ Mergen Sie den Release Branch auch in den Branch *development*, damit künftige Releases ebenfalls die im Release Branch getätigten Bug Fixes enthalten können.
- ☑ Entfernen Sie den Release Branch.

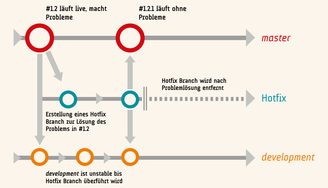
Vor dem finalen Rollout werden keine größeren neuen Features mehr gemergt – sollte dennoch gerade das nächste Feature in den releasefähigen Status übergehen, wird es im Branch *development* gemergt und dort bis zum nächsten großen Release vorgehalten.




### Hotfix Branch *(hotfixes-*)*

Der Hotfix Branch hilft dabei, schnell und live Produktionsprobleme zu fixen. Er ist dem Release Branch ähnlich, wird aber unvorhergesehen gebraucht. Denn er entsteht aus der Notwendigkeit, sofort zu handeln, wenn der Zustand der produktiv laufenden Version unerwünschte Merkmale aufweist (z. B. critical bugs in der Production-Version, die sofort gefixt werden müssen). Der Hotfix Branch zweigt vom Branch *master*, oder besser gesagt, von dem Tag auf dem Branch *master*, das die Produktionsversion markiert.

Die schnellen Production Fixes werden auf dem Hotfix Branch erledigt, damit zur gleichen Zeit auf dem Branch *development* weitergearbeitet werden kann. Der Hotfix Branch wird in *master* und *development* überführt. Danach wird der Hotfix Branch wieder gelöscht.




**Ausnahme!**

Wenn parallel auch noch ein Release Branch aktiv ist, wird der Hotfix Branch in der Regel nur in diesen gemergt. Mit dem Release Branch werden die Fixes dann in den Master gemergt. Werden die im Hotfix Branch getätigten Production Fixes besonders akut benötigt, wird sowohl in den Release Branch als auch in den Development Branch gemergt.

### Der komplette Git-Workflow




## Commands

### 1) Lifecycle einer Git-Datei

**untracked:** noch nicht versioniert
**unmodified:** versioniert, aber noch nicht verändert
**modified:** verändert, aber noch nicht im Stage
**staged:** kann committet werden

### 2) Ein lokales Repository besteht aus:

**Working Directory/Arbeitskopie:** erste Station für ausgecheckte Dateien
**Staging Area/Index:** zweite Station, enthält die ausgecheckten und bearbeiteten Dateien, die im nächsten Schritt zu einem Branch in einem Repository hinzugefügt werden sollen
**HEAD:** verweist auf den letzten Commit im aktuellen Branch

### 3) Git Commands

**Neues lokales Repository von bestehender Arbeitskopie anlegen**

```
git init
```

**Neues Repository ohne Arbeitskopie anlegen**

```
git init --bare
```

#### Repository klonen und Arbeitskopie auschecken

**Arbeitskopie erstellen**

```
git clone /pfad/.../... [zielpfad]
```

**Bei Verwendung eines entfernten Repositorys**

```
git clone [protocol://][benutzername@host:]/pfad/.../... [zielpfad]

// Es genügt, hier einen gültigen URL anzugeben.
// Benutzername, Zielpfad und das Protokoll sind optional.

// Mögliche Werte für das Protokoll: http:, https:, ssh
// oder git
```

#### Add & Commit

**Änderungen der Staging Area/Index hinzufügen**

```
git add <dateiname>
git add *
git add . // Rekursiv, nimmt alle Verzeichnisse und
          // Unterverzeichnisse mit in den Index
```

**Bestätigen der Änderungen in der Staging Area/Index**

```
git commit -m "Commit-Nachricht"
git commit -am "Commit-Nachricht" // -a fügt alle
// Änderungen an Dateien hinzu, die bereits getrackt
// werden. Damit spart man sich ein git add <dateiname>
```

—> Änderung in HEAD

**Änderungen in entfernte Repository hochladen**

```
git push origin master
```

*master* kann durch beliebigen anderen existierenden Branch im *origin* ersetzt werden.

**Lokales Repository ist nicht von entferntem geklont worden, soll aber mit einem anderen Repository verbunden werden**

```
git remote add origin <repository-url>
```

#### Branching Commands

**Zu einem neuen Branch (feature_branch) wechseln**

```
git checkout –b feature_branch
```

**Zurück zum Master**

```
git checkout master
```

**Neuen Branch löschen**

```
git branch –d feature_branch
```

#### Mergen und Aktualisieren

**Neueste Änderungen in aktuellen Branch des lokalen Repositorys übernehmen**

```
git pull origin master
```

**Branch mit einem anderen Branch zusammenführen**

```
git merge <branch>
```

**Zusammenführen von Änderungen**

```
git add <dateiname>
```

**Differenzen zwischen Branches anzeigen**

```
git diff <quellbranch> <zielbranch>
```

**Änderungen der Staging Area/Index im Vergleich zum letzten Commit anzeigen**

```
git diff
```

#### Tagging

**Neuen Tag erstellen (z. B. v0.1)**

```
git tag v0.1 [<commit-ID>]
```

**Neuen kommentierten Tag erstellen (z. B. v0.1)**

```
git tag -a v0.1 -m 'Meine Version 0.1'
```

**Liste der referenzierbaren Commit-IDs**

```
git log
```

#### Änderungen rückgängig machen

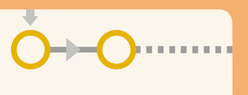
**Lokale Änderungen auf letzten Stand in HEAD zurücksetzen**

```
git checkout -- <dateiname>
```

**Änderungen komplett verwerfen**

Zum Thema „Änderungen verwerfen" finden Sie in der Sektion 10 Profitipps umfangreiche Tipps von Mario Konrad.

## 10 Profitipps



**Bjørn Stachmann,** etracker GmbH
**Feature Branches reviewen**

Wenn man im Team mit Feature Branches arbeitet, möchte man oft wissen, was auf den einzelnen Branches passiert ist. *Fast-Forward Merges** erschweren das leider. Git erspart sich dabei ein Merge Commit, wenn es genügt, den Branch-Zeiger auf ein nachfolgendes Commit vorzurücken. Der Nachteil: Man sieht nicht mehr, woher die Änderungen gekommen sind oder ob überhaupt ein Merge stattgefunden hat. Ich empfehle deshalb *Fast Forwards* abzuschalten:

```
git config --global merge.ff false
```

Die *First-Parent History*, bei der man immer nur den ersten Vorgänger eines Commits betrachtet, zeigt uns, welche Änderungen per Commit direkt auf dem Branch getätigt und welche per Merge von anderen Branches dazugeholt wurden:

```
git log --first-parent master..feature
```

Bei Reviews möchte man oft nur jene Änderungen betrachten, die direkt auf diesen Feature Branches durchgeführt wurden. Dann blendet man die Merges aus:

```
git log --first-parent --no-merges
                              master..feature
```

**Tobias Bayer,** inovex
**Tipp für Kommandozeilen-Aficionados**

Ich habe diese beiden Aliasse in meiner *config*:

```
graph = log --graph
            --pretty=format:'%Cred%h%Creset
      -%C(yellow)%d%Creset %s %Cgreen(%cr)
 %C(bold blue)<%an>%Creset' --abbrev-commit
                            --date=relative

daily = log --since '1 day ago' --oneline
            --author tobias.bayer@inovex.de
```

Der erste zeigt einen schönen Graphen auf der Kommandozeile und der zweite sagt mir, was ich im Daily zu erzählen habe. 😉

**Tim Berglund,** GitHub
**Git Log**

Das Command *git log* bringt eine einschüchternd große Anzahl an Optionen mit sich. Viele Git-User reagieren auf diese Komplexität, indem sie nur die einfachste Einsatzmöglichkeit in Anspruch nehmen. Andere tauschen die Kommandozeile gegen grafische Tools ein, sobald sie sich eine komplexe Repository History anzeigen lassen wollen. Obwohl nichts Verkehrtes daran ist, ein Git-GUI zu verwenden – mir gefallen **http://mac.github.com** und **http://windows.github.com** besonders gut –, sollten Git-User in der Lage sein, die History von der Kommandozeile aus zu sehen, wenn ihnen das lieber ist. Es gibt vier Kommandozeilen-Log-Schalter, die dies ermöglichen:

```
git log --graph --oneline --decorate -all
```

*--graph* sorgt dafür, dass der Log den Repository-Graphen mit ASCII-Art zeichnet, was besser funktioniert, als man vermuten würde. *--oneline* bricht die Ausgabe jedes Commits auf eine einzige Zeile mit verkürztem Commit Hash herunter. *--decorate* versieht jeden Commit mit jedem verfügbaren Branch- oder Tag-Namen. *--all* sorgt dafür, dass der Log die Commits aller Branches anzeigt, nicht nur die des aktuellen.

Dieses Command ist natürlich etwas sperrig. Man sollte also ggf. einen Alias erzeugen – siehe dazu folgenden Tipp von Artur Speth.

**Artur Speth,** Microsoft
**Aliasse**

Aliasse sind nützlich, wenn man Befehle in Git abkürzen möchte. Zum Beispiel bekommt man mit dem Befehl *git diff --cached* die *staged*-Änderungen. Dafür kann ich mir einen Alias einrichten.

Mit *git config --global alias.staged 'diff --cached'* habe ich nun einen einfachen Zugriff auf den Befehl mittels *git staged*.

**Christian Janz ,** BridgingIT
**Änderungen mit „git stash" parken**

Wer kennt das nicht? Während der Entwicklung eines neuen Features muss dringend ein Fehler gefixt werden. Was aber passiert mit den gemachten Änderungen? Hier schafft *git stash* Abhilfe: Damit wird der aktuelle Zustand von Arbeitsverzeichnis und Index gesichert und das Arbeitsverzeichnis auf den HEAD Commit zurückgesetzt. Von dort kann nun in den Release Branch gewechselt werden, um dort den Fehler zu beheben. Danach integriert *git stash pop* die anfangs gemachten Änderungen für das neue Feature wieder in das Arbeitsverzeichnis. Eclipse unterstützt dieses Feature auch seit EGit 2.0.

**Dominik Schadow,** BridgingIT
**Ein Git Repository in ein anderes kopieren**

Mit wenigen Schritten lässt sich ein Git Repository inkl. Historie und Autoren in ein bereits vorhandenes kopieren. Der folgende Tipp zeigt dies mit dem *master* Branch eines Repositorys:

- Das Ziel-Repository normal klonen oder erstellen
- **git remote add -f [remote name] [repo url]** integriert das zweite Repository
- **git merge -s ours --no-commit *[remote name]*/master** führt Änderungen ohne Commit zusammen, wobei im Konfliktfall das Ziel-Repository gewinnt
- **git read-tree --prefix=*[local path]* -u *[remote name]*/master**. integriert alle Commits in den angegebenen Pfad
- **git commit** und **git push** Änderungen übertragen

**Michael Johann,** Eco Novum GmbH
**Statusanzeige im Prompt**

Auf welchem Branch befinde ich mich, und welchen Status hat er? Ist das Arbeitsverzeichnis des aktuellen Branches zwischenzeitlich verändert („dirty") worden? Diese Frage taucht bei einem Entwickler unter Nutzung eines Versionskontrollsystems genauso häufig auf wie der Befehl „ls" *(list)* in einer Shell, um sich den Inhalt des Dateisystems anzeigen zu lassen. Die folgenden Zeilen, eingebunden in das Shell-Skript (z. B.: *~/.bash_profile)*, halten uns immer bequem auf dem Laufenden:

```
function git_deleted {
  [[ $(git status 2> /dev/null | grep
                deleted:) != "" ]] && echo "-"
}
function git_added {
  [[ $(git status 2> /dev/null | grep
       "Untracked files:") != "" ]] && echo '+'
}
function git_modified {
  [[ $(git status 2> /dev/null | grep
                modified:) != "" ]] && echo "*"
}
function git_dirty {
  echo "$(git_added)$(git_modified)$(git_
                                    deleted)"
}
function git_branch {
  git branch --no-color 2> /dev/null | sed -e
  '/^[^*]/d' -e "s/* \(.*\)/[\1$(git_dirty)]/"
}
export PS1='\w$(git_branch)$ '
```

Im Wesentlichen zeigen die Funktionen drei Zustände an:

- Wurden Dateien gelöscht? Dann wird ein Minuszeichen ans Ende des Prompts angehängt.
- Wurden Dateien verändert? In diesem Fall zeigt ein Sternsymbol die Änderung an.
- Wurden Dateien hinzugefügt? Der Status wird dann durch ein Pluszeichen signalisiert.

Sollten Sie ein Linux betreiben, das deutschsprachige Ausgaben erzeugt, sollten Sie die Zeichenketten bei den *grep*-Befehlen entsprechend durch die zu erwartenden Worte ersetzen.

**Martin Dilger,** Freiberuflicher Softwareconsultant und -trainer
**Rebase**

```
"git rebase -i HEAD~<Anzahl commits>"
```

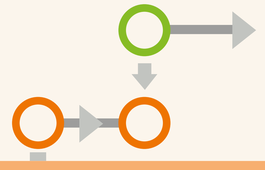
Mit einem Interactive Rebase lässt sich wortwörtlich Geschichte schreiben. Git ermöglicht es, bereits getätigte Commits im Nachhinein (solange sie noch nicht gepusht sind) zu bearbeiten, zusammenzufassen, zu vereinfachen und zu veredeln. *HEAD* bezeichnet den zuletzt getätigten Commit auf einem Branch.

*~ <Anzahl Commits>* beschreibt die Anzahl an Commits, die vom obersten Commit aus bearbeitet werden sollen.

```
pick 1369148 commit 1
r 35cffe6 commit 2
s 126b8b3 commit 3
f 16gdb8s commit 4
```

- Die Standardauswahl ist *pick* oder *p*. Sie bewirkt, dass Commit 1 unverändert bleibt.
- Mit *reword* oder *r* lässt sich im Nachhinein die Commit Message verändern.
- Mit *squash* oder s wie bei Commit 3 würde er nach dem Rebase mit Commit 2 verschmelzen. Seine Commit Message bleibt erhalten.
- Von mir persönlich mit Abstand am häufigsten verwendet wird *fixup* oder *f* wie Commit 4. Dieser Commit würde vollständig mit Commit 3 verschmelzen. Git arbeitet von unten nach oben.

Interactive Rebase ist das Tool der Wahl, um Commits sauber zu strukturieren.



**Mario Konrad,** bbv Software Services
**Git Reset**

Haben Sie in Ihrem Repository einen Commit, den Sie löschen möchten, oder wollen Sie den HEAD auf einen bestimmten Commit zurücksetzen, so können Sie mit Git die Geschichte Ihres Repositorys nachträglich verändern.

Die letzten zwei Commits aus dem Repository unwiderruflich löschen:

```
git reset --hard HEAD~2
```

Auf die gleiche Weise ist es auch möglich, den HEAD auf einen bestimmten Commit zu legen:

```
git reset --hard
      1c22a705ad6018fd75ae71ea5fbb592274e32ef0
```

Wurde beim letzten Commit etwas vergessen oder es hat sich bei der Commit Message ein Fehler eingeschlichen, ist *git-reset* sehr hilfreich:

```
git reset --soft HEAD^
...
git commit -a -m "commit message"
```

Sie möchten den letzten *pull* oder *merge* rückgängig machen? Kein Problem:

```
git pull
...
git reset --hard
```

Haben Sie eine Änderung bereits in ein remote Repository gepusht, dann müssen Sie den Reset des HEADs mit *force* auf das remote Repository pushen:

```
git reset --hard HEAD^
git push --force origin master
```

Bei Ihrer aktuellen Arbeit wollen Sie Änderungen eines Files verwerfen, nicht jedoch die Arbeit an anderen Files beeinträchtigen. Diese Aktion wird nicht mit *git-reset* durchgeführt, sondern mit *checkout*:

```
git checkout -- file.txt
```

Achtung: Seien Sie vorsichtig mit *git-reset*, Sie können so Ihr Repository permanent beschädigen.

**René Preißl,** eToSquare
**Subtree-Befehl**

Submodule bedeuten meist viele fehleranfällige Schritte. Da kommt der *subtree*-Befehl wesentlich einfacher daher. Auch mit ihm kann man ein externes Repository einbinden, z. B. den Branch *master* des HTML5-Boilerplate-Projekts:

```
git subtree add --prefix html5-boilerplate \
                --squash https://github.com/
          h5bp/html5-boilerplate.git master
```

Im Gegensatz zu Submodulen werden dabei aber alle Dateien in das aktuelle Repository importiert. Das weitere Arbeiten bedarf keiner besonderen Schritte. Das Aktualisieren auf einen neueren Stand ist ein einzelner Befehl:

```
git subtree pull --prefix html5-boilerplate \
                 --squash https://github.com/
           h5bp/html5-boilerplate.git master
```

Änderungen im Unterprojekt können auch wieder in das externe Repository zurückübertragen werden (*split*-Befehl).

Mein Tipp: Vermeiden Sie Submodule und nehmen Sie besser den *subtree*-Befehl.



Grafische Umsetzung: meat* – concept and design

Inhalt und Konzeption: Software & Support Media GmbH • Mitarbeit: Michael Johann