

Git kennenlernen

Anfänger

Erste Schritte

Zusammenarbeit

Synchronisierung

Einen Pull Request erstellen

Arbeiten mit Branches

Workflows vergleichen

Zentraler Workflow

Feature Branch Workflow

Git-flow-Workflow

Workflows verzweigen

Migration zu Git

Fortgeschrittene



Workflows vergleichen

Ein Git-Workflow ist eine Rezeptur oder Empfehlung zur Verwendung von Git, die eine konsistente und produktive Arbeitsweise ermöglichen soll. Git-Workflows ermutigen die Benutzer, Git effektiv und konsistent zu nutzen. Git lässt sich für das Management von Änderungen sehr flexibel einsetzen. Aufgrund dieser Flexibilität gibt es keinen standardisierten Prozess für die Interaktion mit Git. Wenn ein Team an einem Projekt in Git arbeitet, ist es wichtig, dass alle Teammitglieder für Änderungen dieselbe Methode anwenden. Um dies sicherzustellen, sollte das Team sich auf einen bestimmten Git-Workflow einigen. Es gibt verschiedene veröffentlichte Git-Workflows, die für dein Team möglicherweise gut geeignet sind. Im Folgenden behandeln wir einige dieser Workflow-Optionen.

Bei der Vielzahl möglicher Workflows ist es nicht leicht, einen Anfang zu finden, wenn man Git in der Arbeitsumgebung implementieren möchte. Diese Seite gibt einen Überblick über die gängigsten Git-Workflows für Software-Teams und bietet so einen ersten Ansatzpunkt.

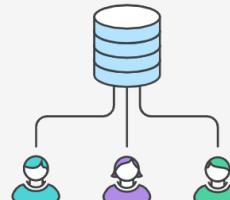
Beachte im Folgenden, dass diese Workflows mehr als Richtlinien und weniger als konkrete Regeln dienen sollen. Wir möchten dir Möglichkeiten aufzeigen, sodass du die Vorteile der verschiedenen Workflows so nutzen kannst, wie es zu deinen individuellen Anforderungen passt.

Wie sieht ein erfolgreicher Git-Workflow aus?

Wenn du einen Workflow für deinen Team auswählst, musst du dabei unbedingt die Teamkultur berücksichtigen. Du möchtest mit dem Workflow die Effizienz deines Teams steigern und nicht die Produktivität behindern. Du solltest bei der Bewertung eines Git-Workflows u. a. die folgenden Punkte beachten:

- Lässt sich der Workflow entsprechend der Teamgröße skalieren?
- Lassen sich mit diesem Workflow Fehler leicht rückgängig machen?
- Macht der Workflow die Arbeit für das Team unnötig komplizierter?

Zentraler Workflow



Der zentralisierte Workflow ist hervorragend für Teams geeignet, die von SVN migrieren. Ganz wie in Subversion arbeitest du auch im zentralisierten Workflow mit einem zentralen Repository, in dem alle Änderungen am Projekt gespeichert werden. Dabei heißt der standardmäßige Entwicklungs-Branch nicht `trunk`, sondern `master`. In ihm werden sämtliche Änderungen committet. Außer dem `master` werden in diesem Workflow keine weiteren Branches benötigt.

Die Umstellung auf ein verteiltes Versionskontrollsystem mag kompliziert erscheinen. Du kannst Git jedoch nutzen, ohne deinen

Möchtest du Git kennenlernen?

Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Möchtest du Git kennenlernen?

Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

aktuellen Workflow verändern zu müssen. Dein Team kann Projekte exakt so entwickeln wie mit Subversion.

Gegenüber SVN hat Git für den Entwicklungs-Workflow jedoch eine ganze Reihe von Vorteilen. Zunächst hat jeder Entwickler eine eigene lokale Kopie des gesamten Projekts zur Verfügung. In dieser isolierten Umgebung kann er unabhängig von seinen Kollegen arbeiten, ohne Rücksicht auf deren Änderungen am Projekt zu nehmen. Commits werden dem lokalen Repository hinzugefügt und Upstream-Änderungen erst einbezogen, wenn er dazu bereit ist.

Zum anderen erhältst du Zugriff auf das solide Branching- und Merging-Modell von Git. Im Gegensatz zu SVN sind Git-Banches darauf ausgelegt, einen störungssicheren Mechanismus zur Code-Integration und zum Teilen von Änderungen zwischen verschiedenen Repositories zu ermöglichen. Der zentralisierte Workflow nutzt wie andere Workflows auch ein serverseitig gehostetes Remote-Repository, in dem die Entwickler Pushes und Pulls durchführen. Im Gegensatz zu den anderen Workflows gibt es beim zentralisierten Workflow allerdings keine definierten Pull-Request- oder Forking-Muster.

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Wie es funktioniert

Jeder Entwickler klonst zunächst das zentrale Repository. Anschließend bearbeitet er die Dateien in seiner lokalen Kopie des Projekts und committet seine Änderungen wie in SVN. Der einzige Unterschied ist, dass die neuen Commits lokal gespeichert werden, vollständig isoliert vom zentralen Repository. Eine Upstream-Synchronisierung wird erst dann durchgeführt, wenn der Entwickler es für angebracht hält.

Sollen die Änderungen im offiziellen Projekt veröffentlicht werden, pusht der Entwickler seinen lokalen master-Branch in das zentrale Repository. Dieser Vorgang entspricht `svn commit`, fügt jedoch alle lokalen Commits hinzu, die nicht bereits im zentralen master-Branch abgelegt sind.

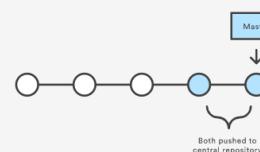
Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Anlegen des zentralen Repositorys



Central Repository
Local Repository



Zunächst muss jemand das zentrale Repository auf einem Server anlegen. Handelt es sich um ein neues Projekt, kannst ein leeres Repository anlegen. Andernfalls muss du ein vorhandenes Git- oder SVN-Repository importieren.

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Zentrale Repositorys sollten immer Bare-Repositorys sein (sie sollten kein Arbeitsverzeichnis haben). Du kannst sie folgendermaßen erstellen:

```
ssh user@host git init --bare /path/to/repo.git
```

Gib dabei einen gültigen SSH-Benutzernamen für user, die Domäne oder die IP-Adresse deines Servers für host und den gewünschten Speicherort deines Repositorys für `/path/to/repo.git`. Es ist Konvention, die Erweiterung `.git` an den Repository-Namen anzuhängen, um das Repository als Bare-Repository zu kennzeichnen.

Gehostete zentrale Repositorys

Zentrale Repositorys werden oft über Git-Hosting-Services von Drittanbietern wie [Bitbucket Cloud](#) oder [Bitbucket Server](#) erstellt. Das Anlegen eines Bare-Repositorys wie oben erläutert wird von deinem Hosting-Service übernommen. Der Hosting-Service stellt anschließend eine Adresse für dieses zentrale Repository bereit, über die du von deinem lokalen Repository aus darauf zugreifen kannst.

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Klonen des zentralen Repositorys

Im nächsten Schritt erstellt jeder Entwickler eine lokale Kopie des gesamten Projekts. Dazu dient der Befehl `git clone`:

```
git clone ssh://user@host/path/to/repo.git
```

Wird ein Repository geklont, fügt Git automatisch eine Verknüpfung mit dem Namen `origin` hinzu, die auf das Ursprungs-Repository verweist. So kannst du später mit diesem Repository interagieren.

Änderungen vornehmen und committen

Nachdem das Repository lokal geklont wurde, kann ein Entwickler mit dem standardmäßigen Commit-Prozess in Git – bearbeiten, auf die Staging-Ebene verschieben und committen – Änderungen vornehmen. Falls du mit der Staging-Umgebung nicht vertraut bist, solltest du wissen, dass sie eine Möglichkeit bietet, einen Commit vorzubereiten, ohne dabei jede einzelne Änderung dem Arbeitsverzeichnis hinzufügen zu müssen. So kannst du, auch wenn du viele lokale Änderungen vorgenommen hast, sehr fokussierte Commits erstellen.

```
git status # View the state of the repo
git add <some-file> # Stage a file
git commit # Commit a file</some-file>
```

Möchtest du Git kennenzulernen?
Sieh dir dieses interaktive Tutorial an.

Jetzt loslegen

Du erinnerst dich sicher, dass mit diesen Befehlen lokale Commits erstellt werden. Daher kann John diesen Prozess so oft wiederholen wie er will, ohne sich darum kümmern zu müssen, was im zentralen Repository passiert. Das kann sehr nützlich sein bei umfangreichen Features, die in einfachere und kleinere Teile zerlegt werden müssen.

Pushen von neuen Commits in das zentrale Repository

Nachdem neue Änderungen in das lokale Repository committet wurden, müssen diese Änderungen gepusht werden, um sie mit den anderen Entwicklern, die an diesem Projekt mitarbeiten, zu teilen.

```
git push origin master
```

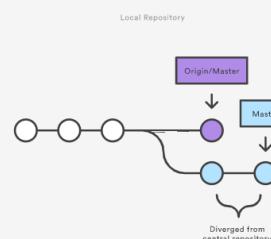
Möchtest du Git kennenzulernen?
Sieh dir dieses interaktive Tutorial an.

Jetzt loslegen

Mit diesem Befehl werden die neu committeten Änderungen in das zentrale Repository gepusht. Beim Pushen von Änderungen in das zentrale Repository ist es möglich, dass Updates von anderen Entwicklern zuvor gepusht wurden und Code enthalten, der mit den vorgesehenen Push-Updates in Konflikt gerät. Git gibt in solchen Fällen eine Konfliktmeldung aus. Dies bedeutet, dass zuerst `git pull` ausgeführt werden muss. Im folgenden Abschnitt gehen wir auf dieses Konfliktszenario noch genauer ein.

Konflikte beheben

Das zentrale Repository stellt das offizielle Projekt dar. Deshalb sollte dessen Commit-Verlauf als unveränderlich betrachtet werden. Wenn die lokalen Commits eines Entwicklers vom zentralen Repository abweichen, verhindert Git, dass Änderungen an ihnen verschoben werden, um das Überschreiben offizieller Commits zu verhindern.



Möchtest du Git kennenzulernen?
Sieh dir dieses interaktive Tutorial an.

Jetzt loslegen

Bevor der Entwickler ein Feature veröffentlichen kann, muss er die aktuellen zentralen Commits abrufen und seine Änderungen voranstellen. Er könnte also sagen: "Ich möchte auf der Arbeit der anderen aufbauen und meine Änderungen hinzufügen." Als Ergebnis entsteht ein absolut linearer Verlauf, genau wie in herkömmlichen SVN-Workflows.

Sollten lokale Änderungen in direktem Konflikt mit Upstream-Commits stehen, stoppt Git den Rebase-Prozess. Du hast dann die Möglichkeit, die Konflikte manuell zu lösen. Ein Vorteil von Git: Mit den Befehlen `git status` und `git add` kannst du nicht nur

Commits erzeugen, sondern auch Merge-Konflikte lösen. Neue Entwickler können ihre Merges auf diese Weise unkompliziert selbst verwalten. Sollten sie doch auf Probleme stoßen, können sie ebenso einfach den gesamten Rebase abbrechen und es nochmals versuchen (oder sich Unterstützung holen).

Beispiel

Sehen wir uns nun ein allgemeines Beispiel an, wie sich die Zusammenarbeit eines typischen kleinen Teams mit diesem Workflow gestaltet. Wir zeigen, wie zwei Entwickler – nennen wir sie John und Mary – an separaten Features arbeiten und über ein zentrales Repository ihre Beiträge teilen.

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

John arbeitet an seinem Feature



John kann in seinem lokalen Repository mit dem standardmäßigen Commit-Prozess in Git – bearbeiten, auf die Staging-Ebene verschieben und committen – Features entwickeln.

Du erinnerst dich sicher, dass mit diesen Befehlen lokale Commits erstellt werden. Daher kann John diesen Prozess so oft wiederholen wie er will, ohne sich darum kümmern zu müssen, was im zentralen Repository passiert.

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Mary arbeitet an ihrem Feature



Mary arbeitet parallel an einem eigenen Feature in ihrem lokalen Repository und nutzt dazu denselben Prozess nach dem Muster "Bearbeitung/Staging/Commit". Ganz wie John kann sie dabei das zentrale Repository vollständig ignorieren. Auch die Änderungen, die John in seinem lokalen Repository vornimmt, sind für sie *vollkommen unwichtig*, da alle lokalen Repositorys *privat* sind.

John veröffentlicht sein Feature



Sobald John sein Feature abgeschlossen hat, sollte er seine lokalen Commits auf dem zentralen Repository veröffentlichen, damit andere Teammitglieder darauf zugreifen können. Dies kann er über den Befehl `git push` erreichen:

```
git push origin master
```

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Denke daran, dass es sich bei `origin` um die Remote-Verbindung zum zentralen Repository handelt, das mit Git erstellt wurde, als John es geklont hat. Das `master`-Argument weist Git an, den `master`-Branch von `origin` so aussehen zu lassen wie seinen lokalen `master`-Branch. Da das zentrale Repository nicht mehr aktualisiert wurde, seit John es geklont hat, wird es keine Konflikte geben und der Push-Vorgang erwartungsgemäß verlaufen.

Mary versucht ihr Feature zu veröffentlichen



Möchtest du Git kennenlernen?
Sieh dir dieses

[Jetzt loslegen](#)

Sehen wir uns mal an, was passiert, wenn Mary ihr Feature versucht zu verschieben, nachdem John seine Änderungen in das zentrale Repository verschoben hat. Sie kann haargenau denselben Befehl zum Verschieben nutzen:

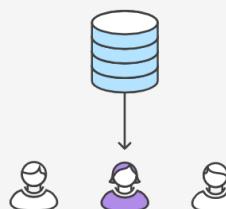
```
git push origin master
```

Doch ihr lokaler Verlauf weicht vom zentralen Repository ab. Deshalb lehnt Git den Request ab und zeigt eine längere Fehlermeldung an:

```
error: failed to push some refs to '/path/to/repo.git'
hint: Updates were rejected because the tip of your current branch
hint: has conflicts with its remote counterpart. Merge the remote changes
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push'
```

So wird Mary daran gehindert, offizielle Commits zu überschreiben. Sie muss die Aktualisierungen von John in ihr Repository verschieben, ihre lokalen Änderungen implementieren und es noch einmal versuchen.

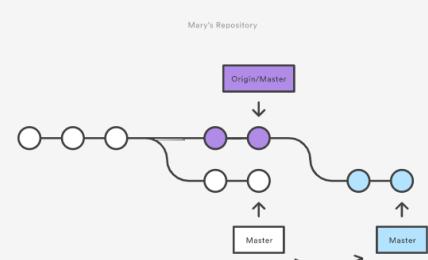
Mary stellt Johns Commit(s) ihre Änderungen voran.



Mary kann `git pull` verwenden, um Upstream-Änderungen in ihr Repository zu integrieren. Dieser Befehl ist `svn update` ähnlich – damit wird der gesamte Upstream-Commit-Verlauf in Marys lokales Repository gepulkt und versucht, ihn in ihre lokalen Commits zu integrieren:

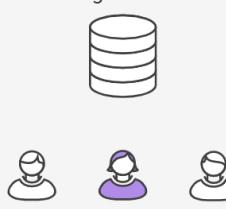
```
git pull --rebase origin master
```

Die Option `--rebase` weist Git an, alle Commits von Mary zur Spitze des `master`-Branches zu verschieben, nachdem dieser mit den Änderungen aus dem zentralen Repository synchronisiert wurde, wie unten gezeigt:



Das Verschieben würde auch ohne diese Option funktionieren. Dann würdest du aber jedes Mal, wenn jemand eine Synchronisierung mit dem zentralen Repository durchführen muss, einen überflüssigen "Merge-Commit" erzeugen. Für diesen Workflow eignet sich das Rebasing immer besser als ein Merge-Commit.

Mary löst einen Merge-Konflikt



Das Rebasing erfolgt, indem jeder lokale Commit einer nach dem anderen an den aktualisierten `master`-Branch übertragen wird. Das bedeutet, dass du Merge-Konflikte für jeden einzelnen Commit erfassst, anstatt alle gleichzeitig in einem massiven

Möchtest du Git kennenlernen?

Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Möchtest du Git kennenlernen?

Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Möchtest du Git kennenlernen?

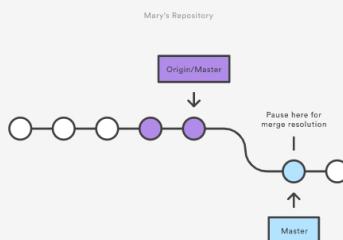
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Merge-Commit zu lösen. Dadurch bleiben deine Commits weitestgehend fokussiert, was zu einem sauberen Projektverlauf führt. Das macht es wiederum wesentlich einfacher herauszufinden, wo Bugs entstanden sind und ob Änderungen mit minimalen Auswirkungen auf das Projekt zurückgesetzt werden können.

Wenn Mary und John an nicht ähnlichen Features arbeiten, sind Konflikte durch den Rebasing-Prozess unwahrscheinlich. Kommt es dennoch zu Konflikten, hält Git das Rebasing bei dem aktuellen Commit an und gibt folgende Meldung zusammen mit einigen relevanten Anweisungen aus:

```
CONFLICT (content): Merge conflict in <some-file>
```



Das Tolle an Git ist, dass *jeder* seine eigenen Merge-Konflikte lösen kann. In unserem Beispiel würde Mary einfach den Befehl `git status` ausführen, um festzustellen, wo das Problem liegt. In Konflikt stehende Dateien werden im Abschnitt mit nicht zusammengeführt Pfaden angezeigt:

```
# Unmerged paths:
# (use "git reset HEAD <some-file>..." to unstage)
# (use "git add/rm <some-file>..." as appropriate to m
# both modified: <some-file>
```

Danach wird sie die Datei(en) entsprechend ihren Anforderungen bearbeiten. Wenn sie mit dem Ergebnis zufrieden ist, kann sie die Datei(en) wie üblich stagieren, den Rest erledigt dann `git rebase`:

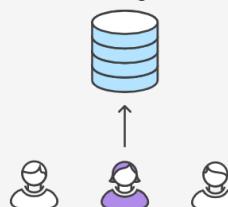
```
git add <some-file>
git rebase --continue
```

Und das war's schon. Git fährt mit dem nächsten Commit fort und wiederholt den Prozess für alle anderen Commits, die Konflikte erzeugen.

Wenn du an dieser Stelle feststellst, dass du keine Ahnung hast, worum es geht, keine Sorge. Führe einfach den folgenden Befehl aus und du wirst wieder dort landen, wo du angefangen hast:

```
git rebase --abort
```

Mary veröffentlicht erfolgreich ihr Feature



Nachdem Mary die Synchronisierung mit dem zentralen Repository abgeschlossen hat, kann sie ihre Änderungen erfolgreich veröffentlichen:

```
git push origin master
```

Wie geht es weiter?

Wie du siehst, kann man mit ein paar Git-Befehlen die herkömmliche Entwicklungsumgebung einer Subversion replizieren. Das ist gut, wenn Teams die Umstellung von SVN durchführen wollen. Doch dabei kommen die Möglichkeiten, die Git zur verteilten Zusammenarbeit bietet, nicht zum Einsatz.

Der zentralisierte Workflow eignet sich gut für kleine Teams. Der oben beschriebene Konfliktlösungsprozess kann bei größeren Teams zu Frustrationen führen. Wenn dein Team mit dem

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Möchtest du Git

teams zu erzielen kann. Wenn dein Team mit dem zentralisierten Workflow einigermaßen vertraut ist, aber lieber seine Zusammenarbeit optimieren möchte, solltest du auf jeden Fall einmal die Vorteile des [Feature-Branch-Workflow](#) ansehen. Wenn jedem Feature ein isolierter Branch zugewiesen wird, können ausführliche Diskussionen über neue Ergänzungen angestoßen werden, bevor diese in das offizielle Projekt integriert werden.

kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Weitere geläufige Workflows

Der zentralisierte Workflow ist im Wesentlichen ein Baustein für andere Git-Workflows. Die meisten beliebten Git-Workflows verfügen über ein zentralisiertes Repository in irgendeiner Form, von dem die einzelnen Entwickler pullen und in das sie pullen. Im Folgenden sprechen wir kurz einige andere beliebte Git-Workflows an. Diese erweiterten Workflows bieten starker spezialisierte Muster für das Management von Branches für die Feature-Entwicklung, von Hotfixes und von Releases.

Feature Branching

Feature-Branching ist eine logische Erweiterung des zentralisierten Workflows. Die Grundidee hinter dem [Feature-Branch-Workflow](#) ist, dass die gesamte Feature-Entwicklung in einem dedizierten Branch und nicht in einem `master`-Branch stattfinden sollte. Diese Einkapselung erleichtert mehreren Entwicklern die Arbeit an einem bestimmten Feature, ohne dabei die Haupt-Codebasis zu stören. Außerdem wird der `master`-Branch dadurch niemals beschädigten Code enthalten, was für Continuous-Integration-Umgebungen ein immenser Vorteil ist.

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Git-flow-Workflow

Der [Git-flow-Workflow](#) wurde erstmals 2010 in einem hoch angesehenen Blogbeitrag von [Vincent Driessen](#) auf [nvie](#) veröffentlicht. Der Git-flow-Workflow definiert ein strenges Branching-Modell, das um den Release des Projekts konzipiert wurde. Dieser Workflow fügt keine neuen Konzepte oder Befehle hinzu, die über das für den Feature-Branch-Workflow erforderliche hinausgehen. Stattdessen ordnet er verschiedenen Branches äußerst spezifische Rollen zu und definiert, wie und wann diese interagieren sollen.

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Workflows verzweigen

Der [Forking-Workflow](#) unterscheidet sich vollkommen von den in diesem Tutorial besprochenen Workflows. Anstatt ein einzelnes serverseitiges Repository als zentrale Codebasis zu verwenden, bietet er jedem Entwickler ein serverseitiges Repository. Jeder Beteiligte arbeitet also nicht mit einem sondern zwei Git-Repositories: einem privaten, lokalen und einem öffentlichen auf Serverseite.

Leitlinien

Den einen Git-Workflow, der für alle Situationen geeignet ist, gibt es nicht. Wie zuvor erläutert ist es wichtig, einen Git-Workflow zu entwickeln, der die Produktivität deines Teams steigern kann. Er sollte nicht nur zur Teamkultur, sondern auch zur allgemeinen Unternehmenskultur passen. Git-Features wie Branches und Tags sollten den Release-Zeitplan deines Unternehmens ergänzen. Wenn dein Team [Projektmanagementsoftware](#) zum Verfolgen von [Aufgaben](#) nutzt, solltest du Branches einsetzen, die mit den in Arbeit befindlichen Aufgaben übereinstimmen. Darüber hinaus solltest du bei der Auswahl eines Workflows einige Leitlinien beachten:

Möchtest du Git kennenlernen?
Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)

Kurzlebige Branches

Je länger ein Branch separat vom Produktions-Branch existiert, desto höher wird das Risiko, dass es zu Merge-Konflikten und Deployment-Problemen kommt. Kurzlebige Branches sorgen für sauberere Merges und Deployments.

Einfaches, möglichst seltenes Rückgängigmachen von Änderungen

Es ist wichtig, dass der verwendete Workflow proaktiv dabei hilft,

merges zu verninaern, die rückgangig gemacht werden müssen.
Ein Workflow, bei dem ein Branch getestet wird, bevor ein Merge
in den master-Branch möglich ist, wäre ein Beispiel hierfür.
Allerdings wird es trotzdem Fälle geben, in denen ein
Rückgängigmachen unvermeidlich ist. Daher ist es von Vorteil,
wenn der Workflow dies auf einfache Weise ermöglicht, ohne dass
andere Teammitglieder in ihrer Arbeit gestört werden.

Anpassung an den Release-Zeitplan

Der Workflow sollte zum Softwareentwicklungszyklus deines
Unternehmens passen. Wenn du mehrmals am Tag veröffentlichtst,
sollte dein master-Branch stabil gehalten werden. Wenn dein
Release-Plan weniger dicht ist, wären für dich Git-Tags zum
Taggen eines Branch zu einer Version eine Überlegung wert.

Zusammenfassung

In diesem Artikel haben wir Git-Workflows erläutert. Wir haben
uns einen zentralisierten Workflow mit praktischen Beispielen
genauer angesehen. Aufbauend auf den zentralisierten
Workflows haben wir weitere spezialisierte Workflows behandelt.
Hier noch einmal eine Zusammenfassung der wichtigsten Punkte
dieses Artikels:

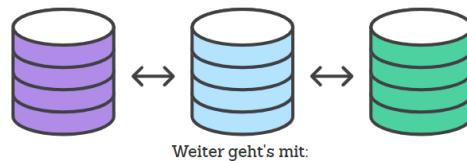
- Es gibt keinen allgemeingültig anwendbaren Git-Workflow.
- Der Workflow sollte einfach sein und die Produktivität deines
Teams steigern.
- Der Workflow sollte entsprechend deinen
Unternehmensanforderungen gestaltet werden.

Wenn du einen weiteren Git-Workflow kennenlernen möchtest,
dann wirf einen Blick in unsere umfassende Dokumentation
zum [Feature-Branch-Workflow](#).

Möchtest du Git kennenlernen?

Sieh dir dieses interaktive Tutorial an.

[Jetzt loslegen](#)



Feature Branch Workflow
Feature Branch Workflow