Git-Workflows - Der Gitflow-Workflow

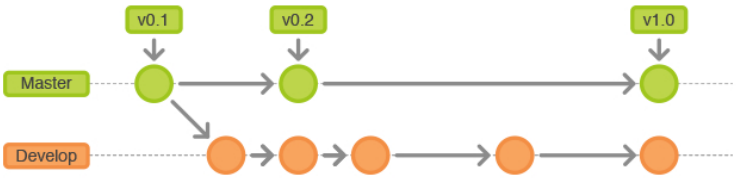
Dieser Workflow definiert ein striktes Branching-Modell, das auf die nahtlose Auslieferung von Releases abzielt. Er ist sicherlich komplexer als der Feature-Branch-Workflow, bietet aber einen robusten Rahmen für das Management großer Projekte.

Der Gitflow-Workflow

Der Workflow sieht keine neuen Konzepte oder Befehle vor, weist verschiedenen Branches jedoch strikte Rollen zu, die festlegen, wie und wann sie interagieren sollten. Natürlich bietet auch der Gitflow-Workflow alle Vorteile des Feature-Branch-Modells: Pull-Requests, isolierte Experimente und eine effizientere Zusammenarbeit.

Der Gitflow-Workflow nutzt ebenfalls eine zentrale Repository als Kommunikations-Drehkreuz für alle Entwickler. Und wie bei den anderen Workflows arbeiten die Entwickler lokal und pushen Branches in die zentrale Repository. Der Unterschied liegt in der Branch-Struktur des Projekts.

Statt eines einzelnen Master-Branchs sieht dieser Workflow zwei Branches vor, die die History des Projekts abbilden. Der Master-Branch enthält die offizielle Release-Historie, der Develop-Branch dient als Integrations-Branch für Features. Es ist zudem üblich, alle Commits in den Master-Branch mit Versionsnummern zu taggen.



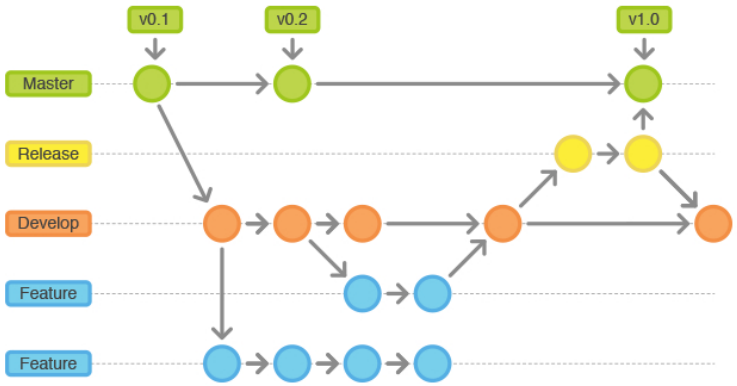
Der Rest des Workflows dreht sich um die Unterschiede zwischen diesen beiden Branches.

Feature-Branches

Jedes neue Feature sollte in seinem eigenen Branch entwickelt werden, der für Backups und aus Gründen der Zusammenarbeit in die zentrale Repository gepusht werden kann. Doch statt Branches auf Basis des Master-Branchs zu erzeugen, wird der Develop-Branch als Quelle genutzt. Wenn ein Feature fertig ist, wird es zurück in diesen Develop-Branch gemergt. Dieser Workflow sieht vor, dass Features niemals direkt mit dem Master-Branch interagieren.

Die Kombination von Feature-Branches und dem Develop-Branch entspricht soweit dem Feature-Branch-Workflow. Doch der Gitflow-Prozess geht darüber hinaus.

Release-Branches



Wenn der Develop-Branch genügend Features für ein Release enthält (oder sich ein vordefinierter Release-Termin nähert), wird vom Develop-Branch ein Release-Branch geforkt. Damit beginnt der nächste Release-Zyklus; neue Features sollten ab diesem Punkt nicht mehr hinzugefügt werden, sondern nur Bugfixes und ähnliche Release-orientierte Änderungen. Ist es zur Auslieferung bereit, wird das Release in den Master-Branch gemergt und mit einer Versionsnummer getaggt. Zusätzlich sollte es zurück in den Develop-Branch gemergt werden, der sich weiterentwickelt haben könnte, seitdem das Release initiiert wurde.

Die Nutzung eines dedizierten Branches zur Vorbereitung von Releases ermöglicht es, dass ein Team das aktuelle Release feinschleift, während das andere Team weiter an Features für das nächste Release arbeitet. Auf diese Weise lassen sich auch bestimmte Entwicklungsphasen sehr gut definieren. (Beispielsweise kann man problemlos sagen “Diese Woche bereiten wie Version 4.0 vor” und sieht dies auch tatsächlich in der Struktur der Repository.)

Konventionen:
Branchen von:

```
develop
```

Mergen in:

```
master
```

Naming-Konvention:

```
release-*
```

oder

```
release/*
```

Maintenance-Branches

Maintenance- oder Hotfix-Branches eignen sich für das Patchen von Produktiv-Releases. Ein solcher Branch ist der einzige, der direkt vom Master geforkt wird. Sobald die Probleme gefixt sind, wird er sowohl in den Master- als auch in den Develop-Branch (oder den aktuellen Release-Branch) gemergt; der Master wird mit einer aktualisierten Versionsnummer getaggt.

Durch eine solche dedizierte Entwicklungslinie für Bugfixes kann ein Team Probleme des Produktiv-Releases beheben, ohne den Rest des Workflows zu unterbrechen oder auf den nächsten Release-Zyklus warten zu müssen. Maintenance-Branches sind sozusagen Ad-hoc-Release-Branches, die direkt mit dem Master interagieren.

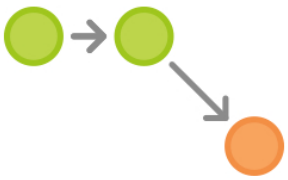
Dieser Workflow sollte einen guten Eindruck von den Potenzialen vermitteln, die lokale Repositories, das Push-Pull-Muster und das Branching- und Merging-Modell von Git bieten. Doch auch der Gitflow-Workflow ist lediglich ein Beispiel für einen effizienten Entwicklungsprozess mit Git und kein unveränderliches Reglement, das unbedingt strikt befolgt werden muss. Es ist kein Problem, einige Aspekte eines Workflows zu adaptieren und andere nicht zu berücksichtigen. Git passt sich Anforderungen und Arbeitsmethoden an.

Weitere Infos

- Stash von Atlassian für Git-Teams - Lizenzen, Preise, Beratung, Schulung, Integration
- Stash: Release Notes und die wichtigsten Features
- Stash: Demo und Einführung

Beispiel

Einen Develop-Branch erzeugen



Ausgehend von der Annahme, dass bereits eine zentrale Repository aufgesetzt wurde, besteht der erste Schritt darin, den obligatorischen Master- um einen Develop-Branch zu ergänzen. Hier erzeugt der Entwickler einen leeren Develop-Branch lokal und pusht in zum Server:

```
git branch develop
```

```
git push -u origin develop
```

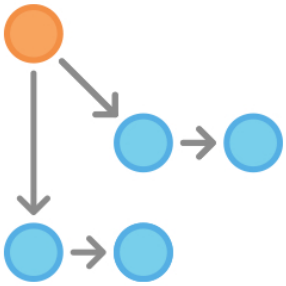
Die Branch wird die komplette Versions-Historie des Projekts enthalten, während der Master eine verkürzte History abbildet. Andere Entwickler sollten die zentrale Repository nun klonen und einen Tracking-Branch für den Develop-Branch erzeugen:

```
git clonessh://user@host/path/to/repo.git
```

```
git checkout -b develop origin/develop
```

Nun hat jeder eine lokale Kopie der historischen Branches aufgesetzt.

Entwickler A und B beginnen neue Features



Das Beispiel beginnt mit Entwickler A und Entwickler B, die an separaten Features arbeiten. Beide benötigen separate Branches, die statt auf dem Master- auf dem Develop-Branch basieren:

```
git checkout -b some-feature develop
```

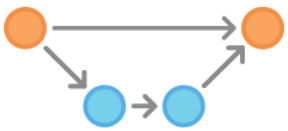
Beide fügen den Feature-Branches Commits nach dem gewohnte Vorgehen hinzu: Edit, stage, commit.

```
git status
```

```
git add
```

```
git commit
```

Entwickler A stellt sein Feature fertig

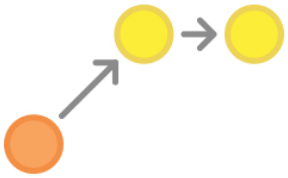


Nach mehreren Commits ist Entwickler A der Meinung, dass das Feature fertig ist. Falls sein Team mit Pull-Requests arbeitet, wäre dies ein guter Zeitpunkt, eine aufzusetzen und daraum zu bitten, das Feature in den Develop-Branch zu mergen. Andernfalls kann er es wie folgt in seinen lokalen Develop-Branch mergen und in die zentrale Repository pushen:

```
git pull origin develop
git checkout develop
git merge some-feature
git push
git branch -d some-feature
```

Der erste Befehl stellt sicher, dass der Develop-Branch aktuell ist, bevor versucht wird, das Feature in diesen zu mergen. Konflikte werden auf die gleiche Weise wie beim zentralisierten Workflow aufgelöst.

Entwickler A beginnt mit der Vorbereitung eines Releases



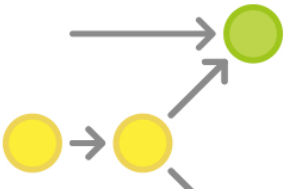
Während Entwickler B nach wie vor an seinem Feature arbeitet, beginnt Entwickler A damit, das erste offizielle Release des Projekts vorzubereiten. Wie bei der Feature-Entwicklung nutzt er einen neuen Branch, um die Release-Vorbereitungen abzukapseln. In diesem Schritt wird auch die Versionsnummer des Releases vergeben:

```
git checkout -b release-0.1 develop
```

Dieser Branch ist dazu da, das Release feinzuschleifen, alles zu testen, die Dokumentation zu aktualisieren und alle weiteren Vorbereitungen für die anstehende Auslieferung zu treffen.

Sobald Entwickler A den Branch erstellt und ihn in die zentrale Repository pusht, ist er für Features geschlossen. Jede Funktion, die nicht bereits im Develop-Branch ist, wird für den nächsten Release-Zyklus aufgeschoben.

Entwickler A stellt das Release fertig



Ist das Release bereit zur Auslieferung, mergt Entwickler A es in den Master- und in den Develop-Branch und löscht anschließend den Release-Branch. Es ist wichtig, zurück in den Develop-Branch zu mergen, da der Release-Branch eventuell kritische Updates enthält, die für neue Features vorhanden sein müssen. Und auch hier gilt: Wenn das Team von Entwickler A Code-Reviews nutzt, ist hier ein sehr guter Zeitpunkt für eine Pull-Request.

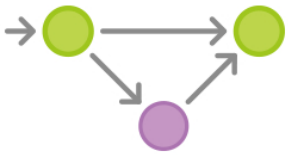
```
git checkout master
git merge release-0.1
git push
git checkout develop
git merge release-0.1
git push
git branch -d release-0.1
```

Release-Branches fungieren als Puffer zwischen der Feature-Entwicklung (Develop-Branch) und öffentlichen Releases (Master-Branch). Wann immer etwas in den Master-Branch gemergt wird, sollte der Commit getaggt werden, um ihn leicht referenzieren zu können:

```
git tag -a 0.1 -m "initial public release" master
git push --tags
```

Git bringt diverse Hooks mit. Das sind Scripte, die ausgeführt werden, wann immer ein bestimmtes Ereignis in der Repository passiert. Diese Hooks können konfiguriert werden, dass automatisch ein öffentliches Release gebaut wird, wann immer der Master-Branch in die zentrale Repository oder wann immer ein Tag gepusht wird.

Endnutzer entdeckt einen Bug



Nach der Auslieferung des Releases macht sich Entwickler A daran, mit Entwickler B Features für das nächste Release zu bauen – bis ein Endnutzer ein Ticket anlegt, in dem er einen Bug im aktuellen Produktiv-Release berichtet. Um diesen Bug zu beseitigen, erzeugt Entwickler A (oder auch Entwickler 😎) einen Branch auf Basis des Master-Branchs, löst das Problem mit so vielen Commits wie erforderlich und mergt den Branch direkt in den Master zurück.

```
git checkout -b issue-#001 master
# Bug wird gefixt
git checkout master
git merge issue-#001
git push
```

Wie Release-Branches enthalten auch Wartungs-Branches wichtige Aktualisierungen, die auch im Develop-Branch enthalten sein müssen. Entwickler A muss also auch diesen Merge ausführen. Anschließend kann er den Branch gerne löschen:

```
git checkout develop
git merge issue-#001
git push
git branch -d issue-#001
```


//SEIBERT/MEDIA GmbH

LuisenForum
Kirchgasse 6
65185 Wiesbaden

T. 0611-205 70-0
F. 0611-205 70-70

info@seibert-media.net
Unsere Website
Unser Weblog