Contents

[Document purpose 1](#_Toc522869252)

[Overall model 2](#_Toc522869253)

[Main branches 2](#_Toc522869254)

[Supporting branches 3](#_Toc522869255)

[Feature branches 4](#_Toc522869256)

[Release branches 5](#_Toc522869257)

[Hotfix branches 6](#_Toc522869258)

[Create pull request 6](#_Toc522869259)

# Document purpose

A GIT Workflow is a recipe or recommendation on how to use GIT to accomplish work in a consistent and productive manner. GIT workflows encourage users to leverage GIT effectively and consistently. GIT offers a lot of flexibility in how users manage changes.

When working with a team on a GIT managed project, it’s important to assure that all team members agree on how the flow of changes will be applied. To ensure the team is on the same page, all team members need to follow this guide and perform GIT actions according to it.

Original GitFlow guide: https://nvie.com/posts/a-successful-git-branching-model/

# Overall model



Overall model

Main branches



Master and develop branches collaboration

The central repo holds two main branches with an infinite lifetime:

Master

Develop

We consider origin/master to be the main branch where the source code of HEAD always reflects a production-ready state.

We consider origin/develop to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release. Some would call this the “integration branch”. This is where any automatic nightly builds are built from.

When the source code in the develop branch reaches a stable point and is ready to be released, all of the changes should be merged back into master somehow and then tagged with a release number. In the text that follows we’ll discuss in more detail how to do this.

Therefore, each time when changes are merged back into the master, a new production release is created by definition.

# Supporting branches

Except the main branches (master and develop) our development model uses a variety of supporting branches to aid parallel development among team members, simplify feature tracking, prepare for production releases and to assist in quickly fixing live production problems. Unlike the main branches, these branches always have a limited life time, since they will, eventually, be removed.

The different types of branches we may use are:

Feature branches

Release branches

Hotfix branches

Each of these branches has a specific purpose and is bound to strict rules: as to which branch may be their originating branch and which branch must be their merge targets. We will walk through them in a minute.

# Feature branches



Feature branch concept

May branch off from:

Develop

Must merge back into:

Develop

Branch naming convention:

Anything except master, develop, release-\*, or hotfix-\*

Feature branches (or sometimes called topic branches) are used to develop new features for the upcoming or a distant future release. When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into develop (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment).

Feature branches typically exist only in the developer repositories, not in origin.

The --no-ff flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that were included in the development of that feature. Compare:

-no-ff flag explanation

In the latter case, it is impossible to see from the GIT history which of the commit objects together have implemented a feature—you would have to manually read all the log messages. Reverting the whole feature (i.e. a group of commits), is a true headache in the latter situation, whereas it is easily done if the --no-ff flag was used.

It will create a few more (empty) commit objects, but the gain is much bigger than the cost.

# Release branches

May branch off from:

develop

Must merge back into:

develop and master

Branch naming convention:

release-\*

Release branches support preparation of a new production release. They allow for last-minute dotting of i’s and crossing t’s. Furthermore, they allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.). By doing all of this work on a release branch, the develop branch is cleared to receive features for the next big release.

The key moment to branch off a new release branch from develop is when develop (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to develop at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

It is exactly at the start of a release branch that the upcoming release gets assigned a version number—not any earlier. Up until that moment, the develop branch reflected changes for the “next release”, but it is unclear whether that “next release” will eventually become 0.3 or 1.0, until the release branch is started. That decision is made on the start of the release branch and is carried out by the project’s rules on version number bumping.

# Hotfix branches



Hotfix branch example

May branch off from:

master

Must merge back into:

develop and master

Branch naming convention:

hotfix-\*

Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

The essence is that work of team members (on the develop branch) can continue, while another person is preparing a quick production fix.

# Create pull request

Before every merging of two branches, we need to perform pull request. Ultimate goal of pull request is to merge a branches into project.

It enable team communication related to the work of the branch. It also provide notifications to team members and they need to do feedback, code review and approval of the content.

You can also open a pull request in the middle of feature implementation. If you are stuck with some implementation, you can ask team members for comments and help.

When reviewer review pull request, they can edit, decline, approve, merge or comment on this request.

When all reviewers approve request, request are closed and we can merge that branches.