TUTORIAL: HOW TO MANAGE GIT BRANCHES USING GITFLOW

Each project has its owr
set of technical
challenges, but most ha
similar versioning needs. A branching and tagging model used for versioning must in some way facilitate future development, quality assurance, releasing, and release maintenance. It incumbent upon project leaders to maintain a consistent process to address these concerns and it is beneficial to a company to maintain a level of consistency between projects to foste cooperation between teams – such as between a project team and operations – and to redu the impact of engineering team changes. To encourage compliance with a reasonable standard and to help it become part of a team's culture, a process should have an obvious a immediate value for team members. Gitflow reduces the time it takes to understand and consistently follow one reasonable process. In day-to-day development, it also saves keystrokes while keeping things clean.

Gitflow is a collection of command line git extensions to automate a branching strategy introduced by a 2010 blog post entitled 'A successful Git branching model' by Vincent Driessen – http://nvie.com/posts/a-successful-git-branching-model. While the model utilize standard git practices, it is well defined, simple, and meets the predominant concerns of development and release cycles with finesse. The process – especially its management of releases and hotfixes – is worth consideration even if you do not adopt the gitflow tools.

Gitflow commands encourage discipline and consistency by simplifying branching and merging in their particular way, but they still use standard git branches, tags, and commits, retaining each developer's ability to use the full power of git outside of this process at their discretion. Manual branching and merging can still be done to facilitate any process, such a for user acceptance testing. Two branches are permanent: "master", which reflects a release or to-be-released product, and "develop", which includes completed features for the next release. You could name these branches as you please when setting up gitflow, but the less explanation is required for other developers if you use the defaults. Branches track features, releases, and hot-fixes to releases as well with standardized directory naming such as "feature/". These branches are temporary and live only until they are merged.

To practice the gitflow process, follow the steps outlined below. A script containing all of these commands and the resulting log from a successful run can be found at https://gist.github.com/4251497.

Install Gitflow

Gitflow is installed locally and operates upon a normal git repository. Each developer will ne to install and configure it. On Macs, it is most easily installed using Homebrew. Install Homebrew from http://mxcl.github.com/homebrew, and then run "brew install git-flow". For alternate installation instructions, see https://github.com/nvie/gitflow/wiki/Installation.

These commands will create a new git repository and create the default branches used by gitflow.

$ git init gitflow-tutorial
$ cd gitflow-tutorial/
$ git flow init -d

Develop a feature

These steps will create a feature branch, introduce a feature, merge it back to the develop branch, and finally delete the feature branch.

Make a feature branch

$ git flow feature start awesome-thing
# Switched to a new branch ‘feature/awesome-thing’

Commit a change

$ touch file1.txt
$ git add .
$ git commit -m ‘Implemented an awesome thing.’

Merge commits from develop and upstream

This step can be safely skipped in our tutorial, but is quite important when working on multiple features or with a remote repository. You should use rebase on the feature branch t amend your commits instead of merging. This allows you to keep the graph simple and assures that your commits will remain atomic instead of introducing changes into a merge commit, which could make it more difficult to understand, cherry-pick, or revert later. You should only rebase temporary branches such as feature branches if you wish to otherwise preserve branch history.

$ git rebase develop
# Manage conflicts by updating your commits if needed.

Merge the feature

You have already handled merge conflicts by rebasing, so when you finish the feature there should be no further conflicts.

$ git flow feature finish awesome-thing
# The feature is merged into develop.
# Switched to branch ‘develop’

Create a release

These steps will create a release branch from the develop branch, update the version, merge all to the master branch, create a tag, merge back to the develop branch, and finally delete th

release branch. Code pushed to the master branch should be considered ready for release. I you are developing a web or enterprise mobile application, you might configure external tool such as Heroku to automatically deploy your application when it is pushed.

Start the release

$ git flow release start v1.0
# Switched to branch ‘release/v1.0’

Perform release specific changes

You may need to update environment configurations, plists, or to perform other changes to support a release. Any changes you make in this branch will be merged into both the develo and master branches after completion, and a release tag will be created upon master.

$ echo ‘v1.0’ > version.txt
$ git add .
$ git commit -m ‘Added version file and set initial v1.0 version.’

Complete the release

$ git flow release finish -m ‘v1.0’ v1.0
# The release is merged into master and develop.
# Switched to branch ‘develop’.

Hotfix the release

Hotfixing should be done to fix issues with a release. The process is almost identical to the release process although the hotfix branch is created from the master branch to reflect a change to the last release.

$ git flow hotfix start v1.0.1
# Switched to branch 'hotfix/v1.0.1'.
$ echo 'v1.0.1' > version.txt
$ git add .
$ git commit -m 'Updated version file to v1.0.1.'
$ git flow hotfix finish -m 'v1.0.1' v1.0.1

Work with a Remote Repository

When using gitflow with a remote repository, each developer still must initialize it locally by executing "git flow init".

Pull and merge

Before starting a feature or release, "git pull develop" or use the "-F" option if you want to wo with the latest code. Before rebasing your feature branch as described above, "git pull develop". You are unlikely to need to manage any conflicts when pulling the develop branch

because you have rebased each feature branch with the latest code before merging. Likewis before starting a hotfix, “git pull master” or use the “-F” option to fetch.

Share branches

Share feature, release, and hotfix branches between developers by sharing them as remote branches.

$ git flow feature publish my-feature

When merging a remote branch, make sure to clean it up on the remote side as well with “-F’

$ git flow feature finish my-feature -F

The remote endpoint can even be set, allowing gitflow to fit naturally into fork/pull-request processes.

Result

Examine the resulting git graph using a tool like SourceTree. You’ll see that all of the temporary branches we have created were deleted after merging, and we are left with only th permanent branches: develop and master.

DEVELOP BRANCH

The development branch has seen a lot of action. We branched off of it to implement a feature, merged that branch, and then branched and merged again to support a release. Finally, we merged a hotfix originating from master.

MASTER BRANCH

The master branch is simple in comparison, because the changes for the release were all merged at one time. Notice that there is one merge per tag, making it trivial to see the list of commits for each version, or to view all changes files for the entire merge.

The tutorial outlined above is simplified, but should get you comfortable enough to start usi Gitflow with your projects today. Refer to Vincent Driessen's original blog post (http://nvie.com/posts/a-successful-git-branching-model) and project site (https://github.com/nvie/gitflow) for more information, including installation instructions, command line arguments, and interesting project forks.