git workflow Documentation

Release 0.0.1

Angie Mutava

Nov 16, 2017

Contents:

<<<<<<< HEAD Welcome to git workflow's documentation! ==

- genindex
- modindex
- search

CHAPTER 1

Introduction

Git workflow is a very broad topic. It's a recipe on how to use git and accomplish work in your workplace. It's very vital to establish or learn the teams' git workflow in any organisation or in your own team. There are different approaches to git workflow. Git is very flexible and therefore need to choose the approach you feel fit to use for your team. While working on a git managed project it is key to have the team agree on the flow. To have the team on the same page is vital to productivity.

From the above introduction it's certain that there is no concrete way for git workflow. Any team is suitable to use whichever combination they feel best suites them. In regard to this we will delve into different approaches of git workflow and feel free to combine any to best suite your team.

1.1 What is a successful git workflow?

When choosing a git workflow it's key to ensure the workflow enhances your teams' productivity. Some key things to consider when choosing a git workflow.

- Scalability - Will the workflow allow for team scale.
- Ease of use - Does is allow team to easily correct errors and mistakes.

CHAPTER 2

Centralized Git workflow

This workflow uses a central repository as the entry point of all the changes to the project. The default and only branch is the master branch. Each member of the team commits to this branch. Using git enables each developer to have local copy of the entire project.This way the developers have the freedom to make commits locally without ever worrying about the upstream changes. This makes them commit their work upstream when they feel it is good enough. The centralized workflow has no defined pattern for forking and branching.

How it works

A central repository is created and each developer clones this repository using the git clone (url) command. When you clone a repository git adds a shortcut to the parent repository called origin, this helps you when you want to reference the parent repository.

Making changes

Each team member adds their feature and commits locally on their local copy. A synchronization is needed from all the team members before changes are pushed upstream. Changes can be made using the standard git commit process edit, stage and commit. The staging area is a way of preparing your work before pushing it to the working directory. This process can be repeated before the substantial work done on a feature is pushed to the central repository. Common git commands in this stage.

```
git status # check the state of the repo.
```

```
git add <some-file> # add a file to the staging area.
```

```
git commit # commits the file.
```

Synchronizing with the central repository

After all is done on the local repo the developer ought to push the changes to the central repository to share their work with other developers in the project.

```
git push origin master
```

This command pushes the changes made to the central repository. Having many people working on the same repository is bound to bring conflicts. In the case where a developers pushes changes upstream and another colleague had their worked pushed, the later developer will have conflicts since they didn't pull the changes made by the prior dev. In this scenario a git pull will be key before pushing code upstream.

Merge conflicts

Before any developer can publish their feature, they need to fetch the central commits and rebase their changes.If the local commits differ with the upstream conflicts: git pauses the rebasing process and allows you to manually resolve the conflicts. Git uses the same git status and git add commands for generating commits and resolving merge conflicts. This enables developers to have control over their own merges. If the case gets worse git enables the users to abort the entire process.

A case example to this workflow can be found here

Rebasing

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

The –rebase option tells git to move all the later developers commits to the tip of the master branch after synchronising it with the changes from the central repository. A pull would still work but a merge commit would be needed everytime you synchronize with the central repository. In this workflow it's better rebasing than generating a merge commit. Rebasing works by transfering each local commit one at a time to the central repository. This means you will catch the conflicts per each commit instead of resolving all of them in a merge commit. This will help maintain a clean project history. It also helps locate bugs and where they were introduced and hence makes it easy to roll back changes without affecting the entire project.

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

In the case where a commit generated an error during rebase git flags the above message and the necessary instructions. The developer can then run the git status command to see where the error is. Conflicting files will appear in the unmerged paths.

```
    Unmerged paths:

    (use "git reset HEAD <some-file>..." to unstage)

(use "git add/rm <some-file>..." as appropriate to mark resolution)

both modified:  <some-file>
```

After the developer corrects her work then she can stage the changes using the git add command and continue with the rebase.

```
git add <some-file>

git rebase --continue
```

Git will move on to the next commit and repeat the process for any other commits that generate conflicts. If as a developer you don't know what is going on then there is need to revert the entire process to where you started. The command for this is:

```
git rebase --abort
```

If all is well then the developer can comfortably push their changes.

```
git push origin master
```

The centralized workflow is key for small teams but cannot work well in a distributed setting.

CHAPTER 3

Feature Branching.

This is a feature branching of the centralized workflow. This means that all the feature work takes place in a branch instead of the master branch. This encapsulation makes it easy for all the developers to work on a feature without affecting the codebase. This also means the master branch should not contain broken code which enhances continous integration. This approach encourages the use of pull requests which is key in collaborative teams since it enhances code reviews. After the developers review the code and the developer involved makes the changes requested then merging is done.

How it works.

Feature branching just like the name suggests involves creating a branch for each feature instead of working from the master branch. Feature branches should have descriptive names. The features can be classified into:

- feature
- chore
- bug

In regard to this the name of the branches should have the head as ft, bg or ch to stand for the highlighted features respectively. The second part should be a descriptive 3 word name that tells more on what the branch achieves. The last part is the story id of the task. The feature branches should be pushed to central repository. This gives other developers a chance to check your work and give feedback. This is also key to monitor each individual's local commits.

All the feature branches are created off the latest code in the project.

```
git checkout master

git fetch origin

git reset --hard origin/master
```

This switches to the branch master, fetches the latest changes/commits and resets the repo to match the latest version.

```
git checkout -b new-feature
```

This creates a new branch based on the master -b tells git to create the branch if none exists.

```
git status

git add <some-file>
git commit
```

The above commands enable you to edit, stage and commit code as many times as you wish.

```
git push -u origin new-feature
```

It is good to push the feature branch online. This gives a chance to the other team players review your work and see what you have been upto. After getting feedback, push your changes online and await a merge of the work to the central repository. In other situations you might have merge conflicts. In case of this solve the merge conflicts and merge the PR.

Pull Requests

Once a developer creates a feature branch pushing it to the central repository is not all they have to do. They will need to have their work featured in the central repository. The only way this can be done is by raising a pull request. This gives a chance to the other developers to view your work before it becomes part of the main codebase. One the reviews are done the developer can rebase from the central branch to get the chnages made by other developers. If there are merge conflicts then they can resolve them and merge the PR.

Other git workflows are repo based and therefore can leverage on this workflow and incorporate it.

CHAPTER 4

Gitflow Workflow.

Defines a strict branching model designed around the project release.This workflow doesn't add any new concepts or commands beyond what's required for the Feature Branch Workflow. It assigns very specific roles to different branches and defines how and when they should interact.

In addition to using the feature branching, it uses feature branches for preparing, maintaining and recording releases.

How it works

It defines clearly what branches need to be set up and how to merge them together. This workflow has two key branches the develop and master branches. The master branch contains the project release whereas the develop branch is the intergration branch for features. The default way is to create a develop branch that compliments the master branch.

```
git branch develop

git push -u origin develop
```

This branch will contain the history of the project.Other developers should now clone the central repository and create a tracking branch for develop. Feature branching in Gitflow workflow. Instead of branching from master feature branches branch from develop.

```
git checkout develop

git checkout -b feature_branch
```

By checking out the develop branch means we will create branches from the develop branch.

```
git checkout develop

git merge feature_branch
```

After finiching on the feature branch you checkout from develop and merge your branch.Once develop has acquired enough features for a release, you fork a release branch off of develop. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it's ready to ship, the release branch gets merged into master and tagged with a version number. In addition, it should be merged back into develop, which may have progressed since the release was initiated.

Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. A new release branch can be created using the following methods.

```
git checkout develop
```

```
git checkout -b release/0.1.0
```

Maintenance or “hotfix” branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they’re based on master instead of develop. This is the only branch that should fork directly off of master. As soon as the fix is complete, it should be merged into both master and develop (or the current release branch), and master should be tagged with an updated version number.

CHAPTER 5

Forking Workflow.

Instead of using a single server-side repository to act as the “central” codebase, it gives every developer a server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one. This workflow is popular in open source projects. The main advantage with this workflow is that changes can be intergrated without everyone having to push their work to a central repository. Developers push their own server side work and only the main developer can push to their local repository.

How it works

This workflow works by having each developer fork the main repository and have their own copy. They do not directly clone the main repository. This works as their public repository. The developer then clones the forked repository and uses it as their local copy. When they make a commit it is updated on their local repository which in return they create a pull request to the main repository to inform the owner that a new feature is ready to be intergrated.

- A developer ‘forks’ an ‘official’ server-side repository. This creates their own server-side copy.
- The new server-side copy is cloned to their local system.
- A Git remote path for the ‘official’ repository is added to the local clone.
- A new local feature branch is created.
- The developer makes changes on the new branch.
- New commits are created for the changes.
- The branch gets pushed to the developer’s own server-side copy.
- The developer opens a pull request from the new branch to the ‘official’ repository.
- The pull request gets approved for merge and is merged into the original server-side repository

To integrate the feature into the official codebase, the maintainer pulls the contributor’s changes into their local repository, checks to make sure it doesn’t break the project, merges it into their local master branch, then pushes the master branch to the official repository on the server. The contribution is now part of the project, and other developers should pull from the official repository to synchronize their local repositories.

The only difference is how those branches get shared by the different workflows. In the Forking Workflow, they are pulled into another developer’s local repository, while in the Feature Branch and Gitflow Workflows they are pushed to the official repository.

Whereas other Git workflows use a single origin remote that points to the central repository, the Forking Workflow requires two remotes—one for the official repository, and one for the developer's personal server-side repository.

```
git remote add upstream link_to _repo
```

The Forking Workflow helps a maintainer of a project open up the repository to contributions from any developer without having to manually manage authorization settings for each individual contributor.

CHAPTER 6

Popular Git commands and how they are used.

- git config –global user.name “Sam Smith”

  git config –global user.email sam@example.com : It configures the author name and email to be used with the commits.
- git init: Initializes a git repository locally.
- git clone username@host/path_to_repo : Creates a copy of the remote repository to your local machine.
- git add * : adds all the unstaged files to the staging area.
- git add <filename> : adds a specific unstaged file to the staging area.
- git commit -m “Commit message”: commits changes to the head of the local repository instead of the remote.
- git commit -a :adds all the files that are in the staging area and all any file changed since then.
- git push origin master : pushes the changes made to the remote repository.
- git status: list all the files you have made changes to.
- git remote add origin <server> : adds the a link to the remote repository to be able to make changes to it.
- git remote -v : list all the currently configured remote repositories.
- git checkout -b <branchname> : creates a branch and switches to it.
- git checkout <branchname> : switches in between branches.
- git branch : lists all the branches in your repository and tells you the branch you are currently working on.
- git branch -a: List all remote or local branches
- git branch -d <branchname> : deletes the feature branch.
- git push origin <branchname> : pushes your branch to remote repository.
- git push –all origin : Pushes all your local branches to the remote repository.
- git push origin : <branchname> : Deletes a branch in your remote repository.
- git pull : fetch and merge changes in your remote repository to your working repository.

- git merge <branchname> : To merge work in another branch to your working branch.
- git rm : Removes files from your index and your working directory so they will not be tracked.
- git diff: Generates patch files or statistics of differences between paths or files in your git repository, or your index or your working directory.
- git archive : Creates a tar or zip file including the contents of a single tree from your repository.
- git gc: Garbage collector for your repository. Optimizes your repository.
- git fsck: Does an integrity check of the Git file system, identifying corrupted objects.
- git prune : Removes objects that are no longer pointed to by any object in any reachable branch.

Updates from remote repository

- git diff : Views all the merge conflict.
- git diff –base <filename> : view the conflicts against a base file.
- git diff <sourcebranch> <targetbranch> : previews changes before merging.
- git add <filename> : After resolving the merge conflicts mark the file.
- git tag 1.0.0 <commitID> : You can use a a tag to mark a significant change like a release.
- git push –tags origin : Pushes all the local tags to the remote repository.
- git log : It lists all the commits.

undoing local changes

- git checkout – <filename> : replaces the changes in your local tree to the local content in the head.
- git fetch origin, git reset –hard origin/master : Instead, to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it, do this.
- git reset –hard HEAD : Resets your index and working directory to the state of your last commit.
- git grep “folder” : searches the working directory.