Git workflow

Introduction

Working with git can be hard, frustrating and can make you suicidal. To prevent this and to take full advantage of the source control management capabilities of git it's useful to have a workflow. A lot of people use "A successful Git branching model" aka GitFlow for their git workflow. As Lars van de Kerkhof, and people from github point out, this workflow is pretty complicated and it's easy to make mistakes with (especially with all the merging going on!). So, we developed an alternative, more simple workflow which also keeps your history clean.

"Github flow" principles :

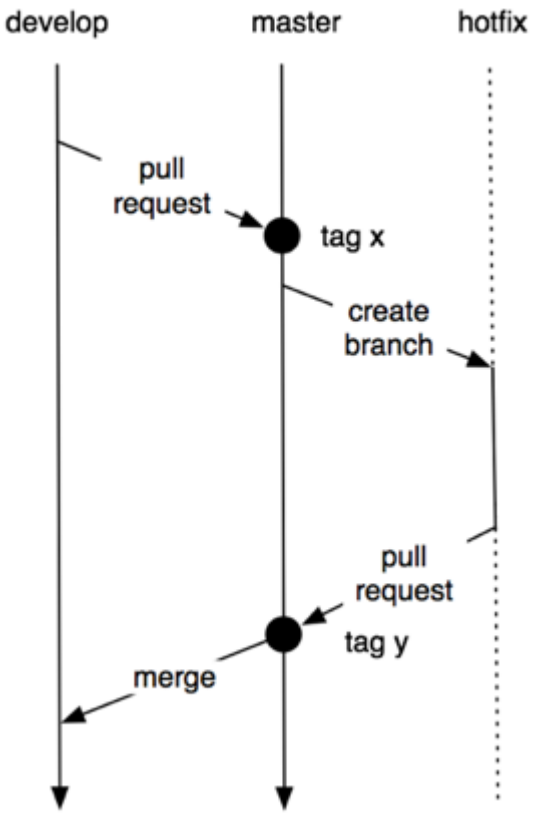
- You have a main repository with two branches : master and develop (*)
- The master branch is your current production branch
- The develop branch is your current development branch
- (*) You can also have just one branch 'master' (or develop), this is sufficient for most of the time
- You develop your new features in feature branches
- ***A feature branch always contains just one user story / issue / task!***
- You rebase your branch often with the main repo upstream development branch
- You always send pull requests when you are ready developing

Let's look at this in more detail.

Main repository

The main repository can have two branches, master and develop. When features are merged into develop and the decision is made to release a new version, you send a pull request to the master branch. (How to add new features to the develop branch is explained later). You tag the release in the master branch so you can, for example, automate package building. You apply hotfixes to the master branch, and you should not forget to merge them as well in the develop branch (this is done with git merge).

Note that you can also have just one 'master' branch and tag the releases there. Most of the time this is sufficient and easier to maintain. The advantage is that you don't need to run the tests again, and that you want to release more often (which is a good thing!).



Main repo development workflow

First checkout the repo you want to work on :

```
$ git clone git@github.com:USER/REPO
```

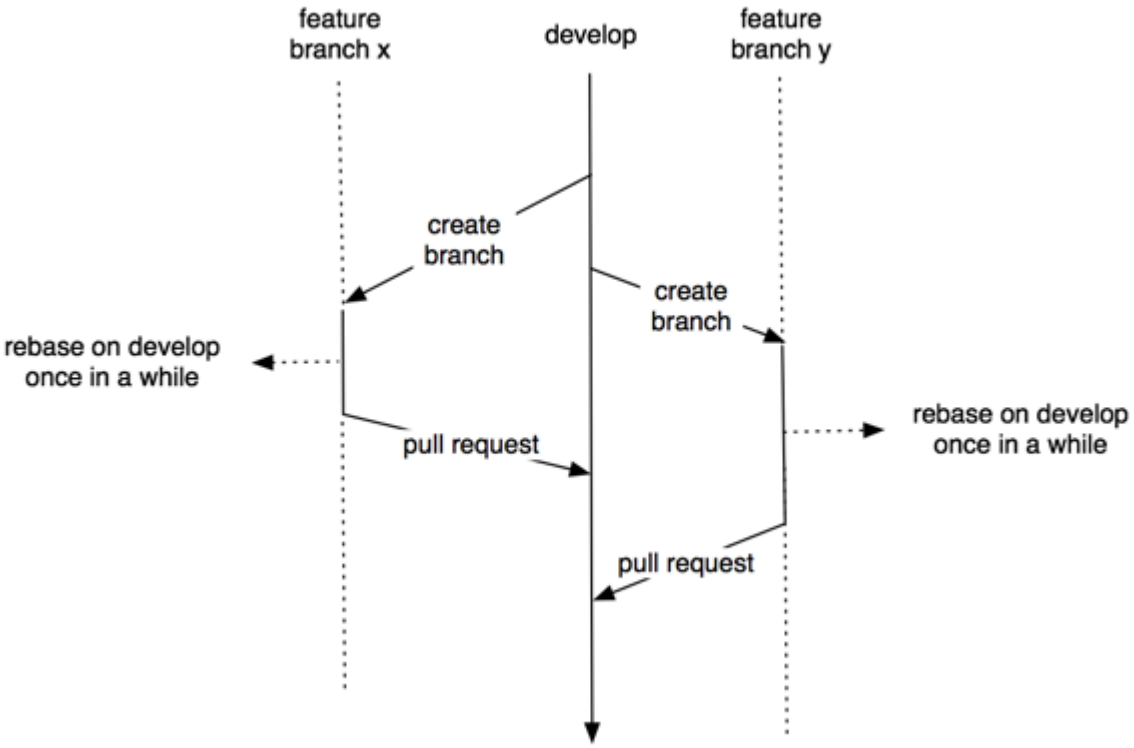
You can list all available branches like this :

```
$ git branch -r
origin/HEAD -> origin/master
origin/develop
origin/featurex
origin/featurey
```

Show your current branch like this (develop if you just cloned) :

```
$ git branch
* develop
featurex
featurey
master
```

The workflow looks like this :



Ok, let's create a new feature branch and check it out immediately :

```
$ git checkout develop && git pull --ff-only
$ git checkout -b featurez
$ git branch
develop
featurex
featurey
* featurez
master
```

Ok, you developed a while on your feature, you committed your changes and now you want to push it to github, and track the branch (newer versions of git do this automatically).

```
$ <do something>...
$ git commit -va
$ git push origin featurez
$ git branch --set-upstream featurez origin/featurez
```

Once in a while, when there is a lot going on in the develop branch, you want to rebase on develop. Rebase means: "rewind all your local commits, apply commits that are remotely available and after that it will apply your local commits again". (So it's almost if you created a branch later in time). The advantage is that your commit history will be clean and lineair and that you will immediately see the conflicts (and you should solve them.) This way, the reviewer of your pull request does not see commits which are already in the develop branch, See also http://elentok.blogspot.nl/2012/01/git-rebase-example.html

First take a look what changed, to be sure. We need to use git fetch (not git pull!) to retrieve the upstream changes:

```
$ git fetch
```

Now we can take a look at the log :

```
$ git log --graph --all
* commit f6bf039df4f7da878f657d575dee5f3bc32114e2
| Author: Martijn Jacobs
| Date:   Mon Dec 10 13:45:34 2015 +0100
|
|     Added test2.txt
|
| * commit c9f4e0d3e831ca3e675ba0cd08c1970668335560
| | Author: Martijn Jacobs
| | Date:   Mon Dec 10 13:45:12 2015 +0100
| |
| |     Added test4.txt
| |
| * commit 3e99ccafb6347b4a3b592712b15b39b8ce945be1
|/  Author: Martijn Jacobs
|   Date:   Mon Dec 10 13:45:02 2015 +0100
|
|       Added test3.txt
|
* commit 3671f190ecee0f2751791631b801458813792aaa
| Author: Martijn Jacobs
| Date:   Mon Dec 10 13:44:11 2015 +0100
|
|     Added test1.txt
|
* commit 919975638e4d9d10a87980a2541bb347c3e13955
  Author: Martijn Jacobs
  Date:   Mon Dec 10 12:43:48 2015 +0000

      Initial commit
```

We see here that the develop branch has one commit already there "Added test2.txt", while we added test3.txt and test4.txt in our feature branch. (Normally you would see pull request merges 99% of the time but this is easier for the example). So we are going to rebase now :

```
$ git rebase origin/develop
```

if we now take a look at the log we see an nice chronological log as the commits in the develop branch are also applied on our feature branch.

Now let's take a look at the history :

```
$ git log --graph --all

    * commit bbc51e4f784b4f0ba367ce70ef752546618cf051
| Author: Martijn Jacobs
| Date:   Mon Dec 10 13:45:12 2015 +0100
|
|     Added test4.txt
|
* commit 68bcc2bb4d62d9e61016bc3e154f7f045c027aee
| Author: Martijn Jacobs
| Date:   Mon Dec 10 13:45:02 2015 +0100
|
|     Added test3.txt
|
* commit f6bf039df4f7da878f657d575dee5f3bc32114e2
| Author: Martijn Jacobs
| Date:   Mon Dec 10 13:45:34 2015 +0100
|
|     Added test2.txt
|
* commit 3671f190ecee0f2751791631b801458813792aaa
| Author: Martijn Jacobs
| Date:   Mon Dec 10 13:44:11 2015 +0100
```

```
|
|     Added test1.txt
|
* commit 919975638e4d9d10a87980a2541bb347c3e13955
  Author: Martijn Jacobs
  Date:   Mon Dec 10 12:43:48 2015 +0000

  Initial commit
```

Our upstream featurez branch is now not in sync anymore with our local featurez branch, so we have to delete it remotely first, and repush it again. This is because we rebased our branch :

```
$ git push origin :featurez
$ git push origin featurez
```

You can also force push your branch if you like, instead of deleting it first:

```
$ git push origin featurez --force
```

When you are done developing in your feature branch, you can send a pull request in github so someone else can review your code and merge it into the develop branch.

After the pull request has been merged, the history of the develop branch also looks nice and clean, so first fetch the changes again:

```
$ git fetch
```

Show the history:

```
$ git log --graph --all
*   commit 653983d28f57e1231c559c1ead37b360124c0486
|\  Merge: f6bf039 bbc51e4
| | Author: Martijn Jacobs
| | Date:   Mon Dec 10 12:58:40 2015 +0000
| |
| |     Merge pull request #1 from martijnjacobs/featurez
| |
| |     Featurez
| |
| * commit bbc51e4f784b4f0ba367ce70ef752546618cf051
| | Author: Martijn Jacobs
| | Date:   Mon Dec 10 13:45:12 2015 +0100
| |
| |     Added test4.txt
| |
| * commit 68bcc2bb4d62d9e61016bc3e154f7f045c027aee
|/  Author: Martijn Jacobs
|   Date:   Mon Dec 10 13:45:02 2015 +0100
|
|       Added test3.txt
|
* commit f6bf039df4f7da878f657d575dee5f3bc32114e2
| Author: Martijn Jacobs
| Date:   Mon Dec 10 13:45:34 2015 +0100
|
|     Added test2.txt
|
* commit 3671f190ecee0f2751791631b801458813792aaa
| Author: Martijn Jacobs
| Date:   Mon Dec 10 13:44:11 2015 +0100
|
|     Added test1.txt
|
* commit 919975638e4d9d10a87980a2541bb347c3e13955
```

```
  Author: Martijn Jacobs
  Date:   Mon Dec 10 12:43:48 2015 +0000
      Initial commit
```

Make sure that you delete your feature branch after the pull request was approved to keep the repo clean (if the reviewer did not do so:

```
$ git push origin :featurez
```

Then switch back to develop

```
$ git checkout develop
```

Also delete your local branch :

```
$ git branch -D featurez
```

Update your local develop branch

```
$ git pull --ff-only
```

Hotfixes

When you have a separate develop and master branch and you need a hotfix, first create a hotfix branch of master. If you have just one main branch, just act if it's a feature branch. For now an example with a separate master and develop branch:

```
$ git checkout master && git pull --ff-only
$ git checkout -b hotfix
<do your work>
$ git commit -va
$ git push origin hotfix
```

Then send a pull request to the master branch. When it's accepted, your hotfix should also be merged with develop. There are several approaches for this. One approach is to merge the master branch back into develop:

```
$ git checkout develop && git pull --ff-only
$ git merge master
$ git commit -va
$ git push origin develop
```

Experimenting with your user story / feature branch

There are cases where you are working on a user story and you want to experiment with a couple of code implementations. At this point you should use the power of Git and create a new branch based on your feature branch. By creating a new branch you can keep your feature branch clean of test code and prevent merging issues when someone else is also working on the same feature branch.

Remote changes to your feature branch

When there is more than one developer working on the same feature branch, you will get some remote changes to it and you need to rebase your feature branch. Those changes are important to your experimental branch as well, so you need to rebase your experimental branch as well.

Squashing experimental commits

When you are experimenting in that new branch, it's very likely that you will do a lot of commits. When you're going to merge your experimental branch into your feature branch, you might not want to add all those messages to your feature branch. This is a point where Git has some more cool stuff for you, namely "squashing commits".

NOTE: ***You should NOT squash commits that exist remotely, except when you do this on purpose in your feature branch. However, NEVER SQUASH COMMITS THAT EXIST IN MASTER/DEVELOP REMOTELY!***

If you check the differences again, you will now get something like this:

```
$ git log develop..HEAD --oneline --decorate
e98fb96 (HEAD, experimental) Some more experimenting
d792832 testing some experimental code
bdd1f6e (origin/feature-x, feature-x) Remote change
c12aa6c changes to some_file and another change
```

Lets say we want to squash the experimental commits (`d792832` and `e98fb96`) into one nice commit. Here is how you do that:

```
$ git rebase -i HEAD~2
```

Note: *When using `HEAD~2`, you tell Git to take the last 2 commits starting with HEAD.*

Your default Git editor will now open and show you something like this:

```
pick d792832 testing some experimental code
pick e98fb96 Some more experimenting

# Rebase bdd1f6e..e98fb96 onto bdd1f6e
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

This is the place where you tell Git what commits you want to combine. We want to combine both commits into one, so we change the top 2 lines into this:

```
pick d792832 testing some experimental code
squash e98fb96 Some more experimenting
```

When you now save this and exit the editor, another editor window will appear and asks you to tell it what the new commit message should be:

```
# This is a combination of 2 commits.
# The first commit's message is:

testing some experimental code

# This is the 2nd commit message:
```

```
Some more experimenting

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# Not currently on any branch.
# You are currently editing a commit during a rebase.
#
# Changes to be committed:
#   (use "git reset HEAD^1 <file>..." to unstage)
#
#       modified:   another_file.txt
#       modified:   some_file.txt
#
```

Change the text into this to create just 1 message for both commits:

```
# This is a combination of 2 commits.
# The first commit's message is:

Added some real cool code that does some real magic!

#testing some experimental code

# This is the 2nd commit message:

#Some more experimenting

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# Not currently on any branch.
# You are currently editing a commit during a rebase.
#
# Changes to be committed:
#   (use "git reset HEAD^1 <file>..." to unstage)
#
#       modified:   another_file.txt
#       modified:   some_file.txt
#
```

Your terminal will now show something like the following:

```
[detached HEAD 311f4e7] Added some real cool code that does some real magic!
 2 files changed, 2 insertions(+)
Successfully rebased and updated refs/heads/experimental.
```

Now you will have 1 nice commit with your experimental code ready for merging into the main feature branch. You can check this by using `git log` again:

```
$ git log develop..HEAD --oneline --decorate
311f4e7 (HEAD, experimental) Added some real cool code that does some real magic!
bdd1f6e (origin/feature-x, feature-x) Remote change
c12aa6c changes to some_file and another change
```

So now you're done experimenting and want to put the code into your main feature branch. This can be simply achieved by merging the `experimental` branch into your feature branch:

```
$ git checkout feature-x
Switched to branch 'feature-x'
$ git merge experimental
Updating bdd1f6e..311f4e7
Fast-forward
 another_file.txt | 1 +
 some_file.txt    | 1 +
 2 files changed, 2 insertions(+)
```

The only thing that remains is to push your code to remote branch and you're good to go! :)