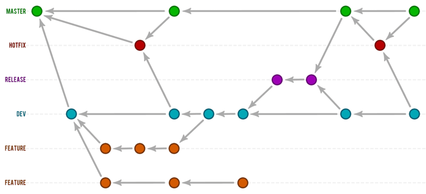
GIT BRANCHING & RELEASE STRATEGY



There are several branching strategies out there but none more popular than Vincent Driessen's GitFlow model. His model allows for the usual mainline master and dev branches, and also feature, release and hotfix branches. This may be overkill for some, but if you have more than a few developers working on the same team and on several different features of the same codebase, it can help you to hang on to your sanity (what little you have left anyway).

The original blog post above is an excellent reference, but I wanted to provide a more detailed workflow. The examples below employ quite a verbose usage of the various Git commands for clarity's sake. For example I always include source/destination branches (local and remote) when pulling, pushing, merging, etc. If you are certain your branch tracking is setup correctly, you can of course use the shorter form commands. Personally, I don't like to leave things to chance so I would rather be explicit and confident of the expected outcome.

LET'S MEET THE PLAYERS (BRANCHES)

MASTER

The master branch is automatically created when you create a new Git repository. Master branch should only store code that is 'release' ready. Deployment to live/production is only ever done from a tag created on this branch.

DEV

The dev branch is sometimes referred to as the 'integration' branch. It begins life as a branch from master. All developed code (from feature branches) is eventually merged into this branch. Dev branch reflects the current state of 'developed' code but not necessarily release ready (a release may require several features).

FEATURE

Feature branches provide an isolated environment to develop an application feature without the risk of potentially harmful changes either from you or other developers affecting each others code. This means you can experiment and even ditch your feature without any impact on other development. Feature branches are always created from, and eventually merged back into dev branch. Once merged, they can be deleted as all your feature code is now in dev branch.

RELEASE

Release branches allow us to take a snapshot of the code at any given time and effectively freeze it from any other development. When any finished features branches have been merged back into dev branch, tested and are ready for production, we can checkout a release branch from dev. On occasion the release may need to be 'polished' with some minor changes (no new functionality here!). When complete, the release branch can be merged into master, tagged with a release number i.e. 3.2, tested and deployed. Don't forget to also merge the release branch back into dev to add any release tweaks that might have been made. As with feature branches, release branches have a limited existence, they can be deleted once merged with master and dev branches.

HOTFIX / MAINTENANCE

So your new feature passed all the tests you created, was included in a release and is now out in the wild on your live website. Then boom! You notice a bug that wasn't captured by your tests. Depending on the severity, you may need to roll back your production environment to the previous release tag. Whatever you do though, you're going to have to fix that bug. This is where we create a hotfix branch from master to implement our bug fix. Once complete we can merge the hotfix into master, tag it with a release number i.e. 3.2.1, test and deploy. As with release branches, we also need to merge the hotfix into dev branch to fix the same bug there.

FROM FEATURE TO PRODUCTION

Feature branches are always branched from dev.

```
$ git checkout -b my_feature_branch origin/dev
```

Any work completed is then committed to your local working copy of the feature branch. You can of course also push this branch to origin if anyone else needs to work on elements of this feature.

```
$ git push origin my_feature_branch
```

They can then checkout their own working copy of the feature branch.

```
$ git checkout -b my_feature_branch origin/my_feature_branch
```

As with any branch, after you have committed your changes but before pushing it to origin, you must pull from origin to ensure all code is merged correctly and fix any merge conflicts that might occur.

```
$ git commit -m"my awesome feature changes"
$ git pull origin my_feature_branch
...
... if any merge conflicts, fix and commit again
...
$ git push origin my_feature_branch
```

When a feature is complete, it must be merged back into dev. Note that if dev branch has been altered since the feature was checked out, you may have some merge conflicts to fix. These must resolved and committed.

```
$ git checkout dev
$ git pull origin dev
$ git merge --no-ff my_feature_branch
$ git push origin dev
```

Note the use of the **--no-ff** option. If dev had not changed since we checked out the feature branch, Git could simply fast forward when merging. This however results in a linear history with no recollection of a feature branch ever existing. By telling Git to not allow a fast forward merge, a commit object is created and the history of the feature branch is retained.

Once the feature branch has been successfully merged, it can be deleted both locally and on origin as it is no longer required. Note you may have to change branch as you can't delete the branch you currently have checked out.

```
$ git checkout dev
$ git branch -D my_feature_branch
$ git push origin :my_feature_branch
```

If you are using Git 1.7.0 or above you can also use the following alternative to delete remote branches:

```
$ git push origin --delete my_feature_branch
```

When you have all the required features for your next release merged into dev and tested, you are ready to create a release branch.

```
$ git checkout -b release_3.2 origin/dev
```

Any minor tweaks can be made and committed to this release branch. There shouldn't be a need to push this branch to origin unless several developers need to collaborate on the tweaks. Remember the rule here is no new functionality. When the release is complete and tested, we can merge it into dev and master branches. We can also delete the release branch once merged.

```
$ git checkout dev
$ git pull origin dev
$ git merge --no-ff release_3.2
$ git push origin dev
$ git checkout master
$ git pull origin master
$ git merge --no-ff release_3.2
$ git push origin master
$ git branch -D release_3.2
```

Now let's cut a tag on master.

```
$ git tag -a v3.2 -m"Release 3.2"
$ git push origin master --tags
```

This tag can now be tested and ultimately deployed to production.

Well done, give yourself a pat on the back. You've done a great ... what's that you say? Someone has reported a bug? I don't think so, not in my ... oh yeah, look at that! So we now have a bug on our live application. Best get working on a hotfix!

When you're done, as with release branches, you must merge the hotfix into both dev and master branches then delete it. We can then cut a new tag for testing and deployment.

```
$ git checkout -b  hotfix_3.2.1 origin/master
...
... various magic performed and committed here
...
$ git checkout dev
$ git pull origin dev
$ git merge --no-ff hotfix_3.2.1
$ git push origin dev
$ git checkout master
$ git pull origin master
$ git merge --no-ff hotfix_3.2.1
$ git push origin master
$ git branch -D hotfix_3.2.1
$ git tag -a v3.2.1 -m"Hotfix Release 3.2.1"
$ git push origin master --tags
```

Now rinse and repeat!

Check out my Git cheat sheet for a quick reference guide to the most used commands.