**Git Branch Management**

#### 

#### **Main Branches**

At the core, the central repo will hold three main branches :

* master
* dev
* pre-release(testing)

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.

When the source code in the dev branch reaches a stable point and is ready to be released, all of the changes should be merged into origin/pre-release, and then go through a bugfix-only QA process to get the branch ready for a production release.

Therefore, each time when changes are merged back into master, this is a new production release **(version/tag)**. This is very strict, as we automatically build and roll-out software to production servers every time there is a commit on master with latest build.

**Supporting Branches**

Next to the main branches, our development model uses a variety of supporting branches to ease tracking of features, to assist in quickly fixing live production problems, and to identify parallel development among team members. Unlike the main branches, these branches are temporary.

The different types of branches we may use are:

* Feature branches (Enhancement or Error or Bug)
* Hotfix branches (Released Version code fixes)

Each of these branches have a specific purpose and are bound to strict rules as to which branches may be their originating branch (created by which branch) and which branches must be their merge targets.

##### **Feature Branches**

branch off from: develop

Must merge back into: develop

Branch origin is anything except master.

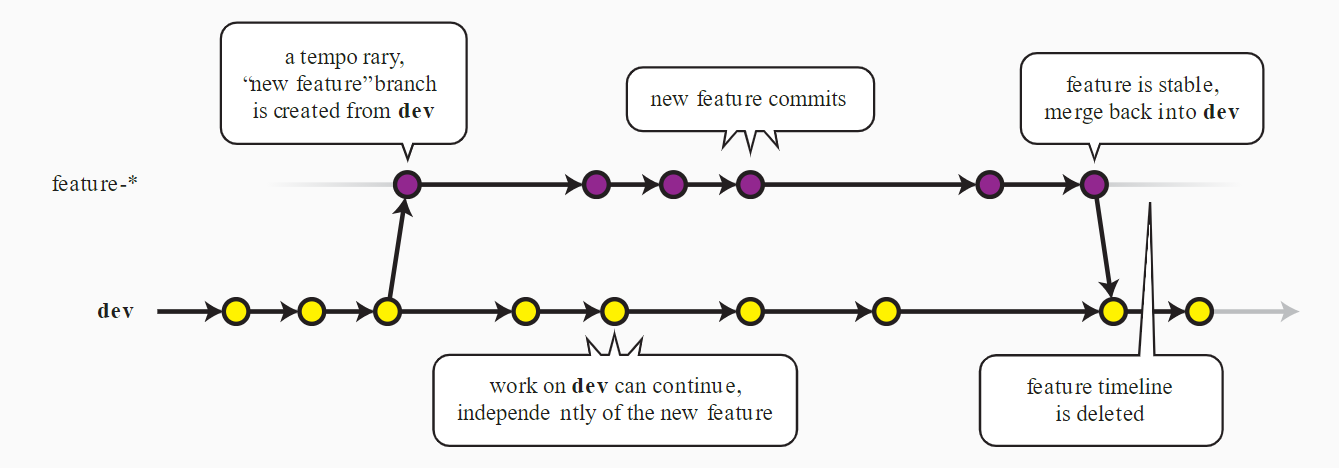
Why because the feature is always, we develop for dev branch of pre-release branch.

Branch naming convention:

(If it is feature write as feature-name-\*, If it is pre-release testing error write as bug-ticketNumber\*)

Feature 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).

Note: Feature branches typically exist in developer repos only, not in origin.



Feature branching commands

###### **Creating a feature branch**

When starting work on a new feature, branch created from the develop branch.

git checkout -b <feature branch name> <from branch name>

$ git checkout -b feature-new-name dev

Switched to a new branch "feature-new-name"

Update the feature branch, making commits as needed

**Incorporating a finished feature on dev**

Finished features may be merged into the dev branch to definitely add them to the upcoming release:

$ git checkout dev

Switched to branch 'dev'

$ git merge feature-new-name

Updating ea1b82a..05e9557(Summary of changes)

$ git push origin dev

The 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 together added the feature. By not doing this, the feature's commit history would be intermingled throughout the commit history of dev, and reverting a whole feature (i.e. a group of commits) would be a headache.

###### **Deleting a feature branch**

Now that we've merged the feature into dev, the feature branch may be removed:

$ git branch -d feature-new-name

Deleted branch feature-new-name (was <deletedId>).

##### **Hotfix Branches**

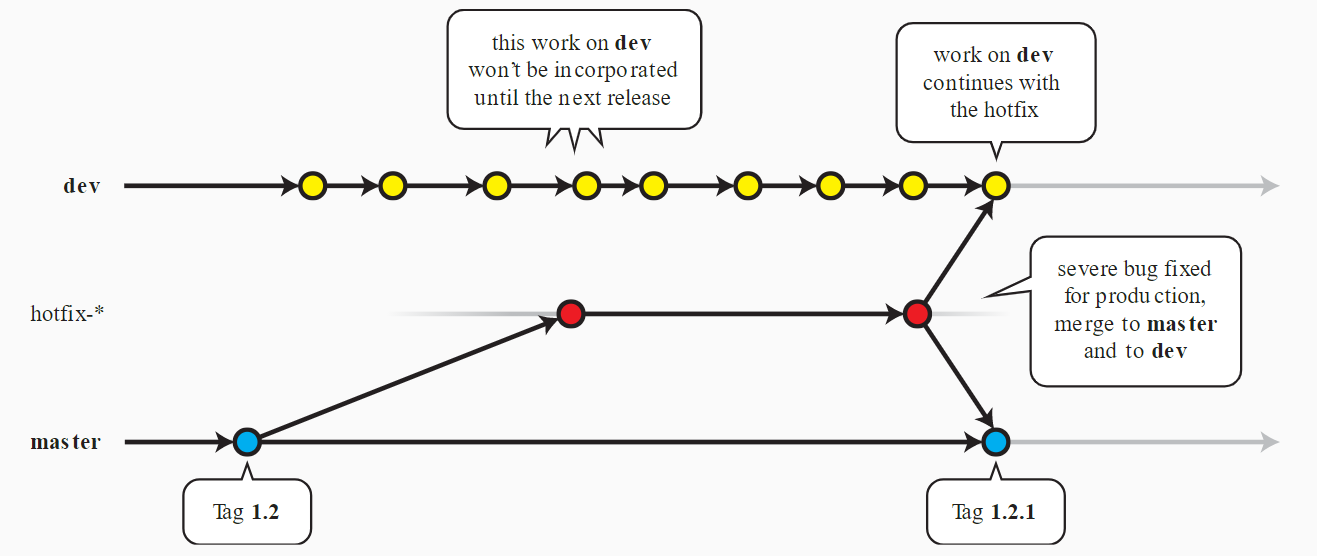
branch off from: master

Must merge back into: develop and master

Branch naming convention: hotfix-version-ticketNumber

Hotfix branches are very much like the pre-release branch in that we are also meant to prepare for a new production release, even if unplanned. we 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 dev branch) can continue, while another person is preparing a quick production fix.



hotfix branching commands

**Creating the hotfix branch**

Hotfix branches are created from the master branch. For example, say version 3.5.0 is the current production release running live and causing troubles due to a severe bug. But changes on dev aren't stable yet. We may then branch off a hotfix branch and start fixing the problem.

$ git checkout -b hotfix-3.5.0-7007 master

Switched to a new branch "hotfix-3.5.0"

*Note: Don’t forget to bump the version number after branching off, If required!*

Then, fix the bug and commit the fix in one or more separate commits.

$ git commit -m "Fixed severe production problem"

[hotfix-3.5.1 abbe5d6] Fixed severe production problem5 files changed, 32 insertions(+), 17 deletions(-)Finishing a hotfix branch

**Incorporating the hotfix branch:**

When finished, the bugfix needs to be merged back into master, but also needs to be merged back into dev, in order to safeguard that the bugfix is included in the next release as well. This is completely similar for pre-release branch if needed.

First, update master and tag the release.

$ git checkout master

Switched to branch 'master'

$ git merge hotfix-3.5.1

Merge made by recursive. (Summary of changes)

$ git tag -a 3.5.1

Next, include the bugfix in develop, too:

$ git checkout dev

Switched to branch 'dev'

$ git merge hotfix-3.5.1

Merge made by recursive.(Summary of changes)

The one exception to the rule here is that, when pre-release is being worked on, the hotfix changes need to be merged into that branch, instead of dev. Back-merging the bugfix into the pre-release branch will eventually result in the bugfix being merged into dev too when pre-release is finished. (If work in dev immediately requires this bugfix and cannot wait for pre-release to be finished, you may safely merge the bugfix into dev now already as well.)

**Deleting a hotfix branch**

Finally, remove the temporary branch:

$ git branch -d hotfix-3.5.1

Deleted branch hotfix-3.5.1 (was abbe5d6).

Bumpping version:

$ git checkout -b hotfix-3.5.1 master

Switched to a new branch "hotfix-3.5.1"

$ ./bump-version.sh 3.5.2

Files modified successfully, version bumped to 3.5.2.

$ git commit -a -m "Bumped version number to 3.5.2"

[hotfix-3.5.2 41e61bb] Bumped version number to 3.5.2 1 files changed, 1 insertions(+), 1 deletions(-)

**GIT Commands:**

$ git init (To initialize folder as git repository)

$ git remote add origin <url> (To add the remote repository url)

$ git remote set-url origin <url> (To change the remote repository url)

$ git add . (To add all the files to git)

$ git commit -am “commit message” (-am used for to add and commit the files)

**Tag related Commands:**

$ git tag (List the available tags)

$ git tag -n (List the available tags with discription)

$ git tag -l v1 (v1 pattern tags will display)

$ git tag -l --sort=-version:refname <pattern>

$ git describe --tags (Latest tag will display)

$ git fetch --tags

$ git tag <tagName> (To create tag)

$ git tag <tag\_name> <commit\_sha> (To create tag for particular commit ID)

$ git tag <tag\_name> HEAD (for the last commit)

$ git tag <tag\_name> HEAD~1 (for the commit before HEAD)

$ git tag <tag\_name> HEAD~2 (for two commits before HEAD)

$ git tag v0.1.0 develop (Tags HEAD of 'develop' branch)

$ git push --tags (To push the created tags)

$ git tag -d <tagname> (To delete the Tag)

**Create branch using tags:**

$ git checkout –track -b <branchname> (To enter into the branch)

$ syntax: git checkout tags<tagname> -b <branch-name> (or) git checkout -b <branch-name> <tag>

$ git checkout tags/v1.0 -b v1.0-branch

git log --oneline --graph ( To check the HEAD point for both created by tag and created branch)

git rm --cached . (To clear the git cache)

git rm -r --cached . (To clear the cache recursively)

**Upstream branch commands:**

$ git push -u <remote> <branch> (To push remote branch)

$ git push --set-upstream <remote> <branch> (To push remote branch)

$ git branch -vv (To see the tracking branch and To check the HEAD of the branches)

$ git diff masterbranch..featurebranch (To get the differences b/w master and feature branch)

$ git rm <file> (To remove the file)

$ git commit -am "Deleted the file from the git repository"

$ git push

$ git ls-tree -r master ( To see the files tree in git)

$ git rm -r <folder> (To delete folder)

$ git checkout HEAD -- .gitignore (To reset the .gitignore file to HEAD)

$ git branch -u origin/<branchName> (to track the local branch to remote)

$ git config credential.helper store (To save the credentials)

$ git branch -m (new branch name)

$ git branch -d branchname (to delete branch)

$ git branch -dr orgin/branchname (To delete branch from local repository)

$ git push origin --delete branchname ( To delete branch from remote repository)

$ git reset --hard origin/master

**Revert Commands**

$ git revert (To revert origin/master)

$ git reset --hard origin/master (To reset branch to origin/master)

$ git reset --soft HEAD~1 (To undo last git commit)

$ git reset --hard HEAD (Going back to HEAD)

$ git reset --hard HEAD^ (Going back to the commit before HEAD)

$ git reset --hard HEAD~2 (Going back two commits before HEAD)

$ git push -f origin master

This is never a recommended use of git. This will overwrite changes on the remote. Only do this if you know 100% that your local changes should be pushed to the remote master.