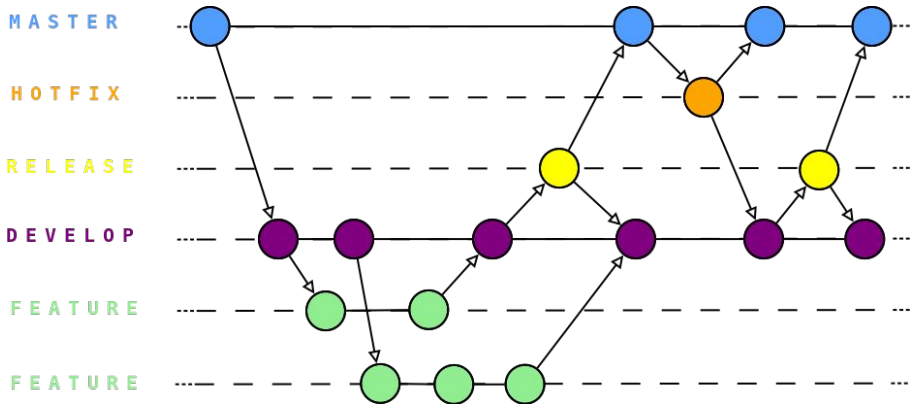
Created by : Zahra
2 Jun 2019

General Structure

Branches

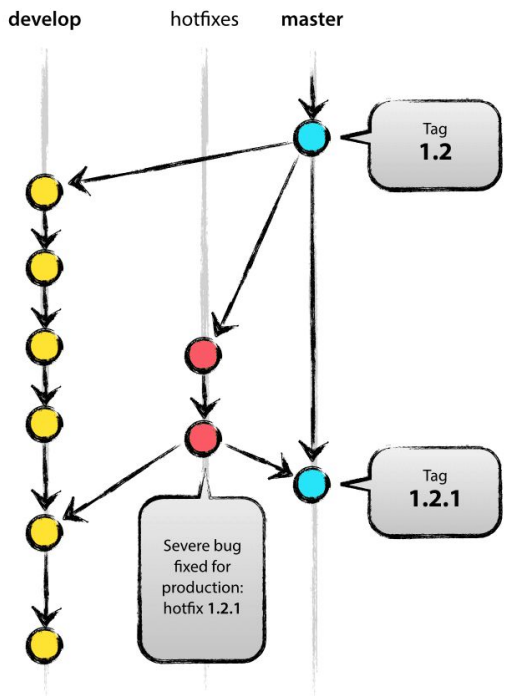
- Master
- Dev
- features/story_id-description



* Hotfix Branch

Hotfix Branches. 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 .

The ***hotfix*** *branches* are cut from master *branch* and merged back to master and then to develop *branch*.



* Release Branch

Working with **git release branches**. **Release branches** contain production ready new features and bug fixes that come from stable develop **branch**. In most cases, master **branch** is always behind develop **branch** because development goes on develop **branch**.

No new code ,there are only some configs

* Debug Branch

They are all about fixing bugs and solving problems of the code

preferably Have 2 distinct repositories:

Back-End

- Master
- Dev
- features/story_id-description

Front-End

- Master
- Dev
- features/story_id-description

Specially if you are using **REST API** which works completely separated

Notice…

We only have clean commits on master

So what are the clean commits?!

1- Independently meaningful

2- Doesn’t cause master not to compile

3- Full explanation of the work which is done

4- Code Documents exist

A Good Commit

A good commit test should be an answer to this

If applied, this commit will <commit message>.

e.g:

“Add Profile page”

Not:(they are False)

Added .. Adding … etc

A Good Commit

And Also CAPITALIZE the first letter

Better to Say the reason you are doing though:

e.g:

Add “serializer” method for verification of something (to verify sth)

A Good Commit

Avoid generic messages

For example:(They are bad)

- ❑ **Fix minor**
- ❑ **Fix this**
- ❑ **Fix stuff**
- ❑ **Fix something**

Another notation:(Not recommended)

feat => A new feature added

fix => fix a problem in code

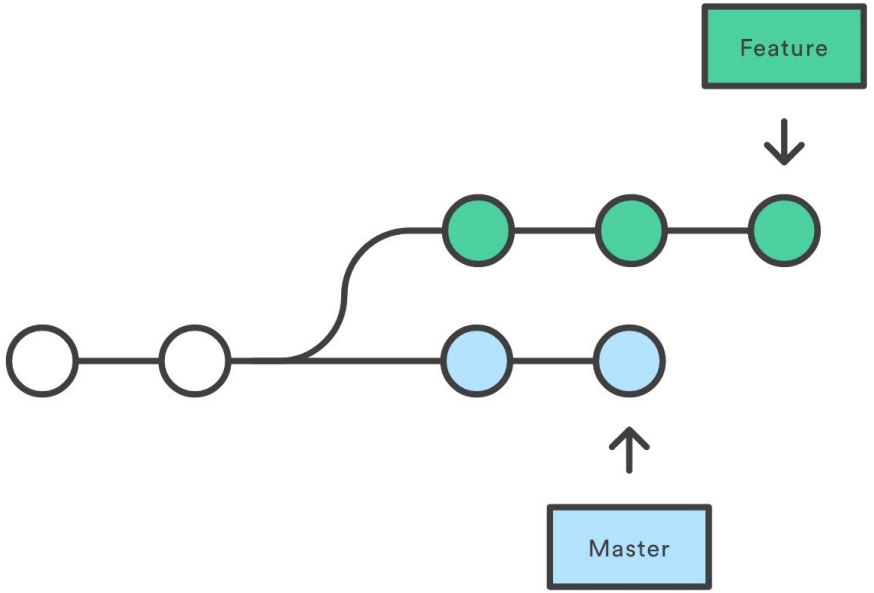
etc …

Format :

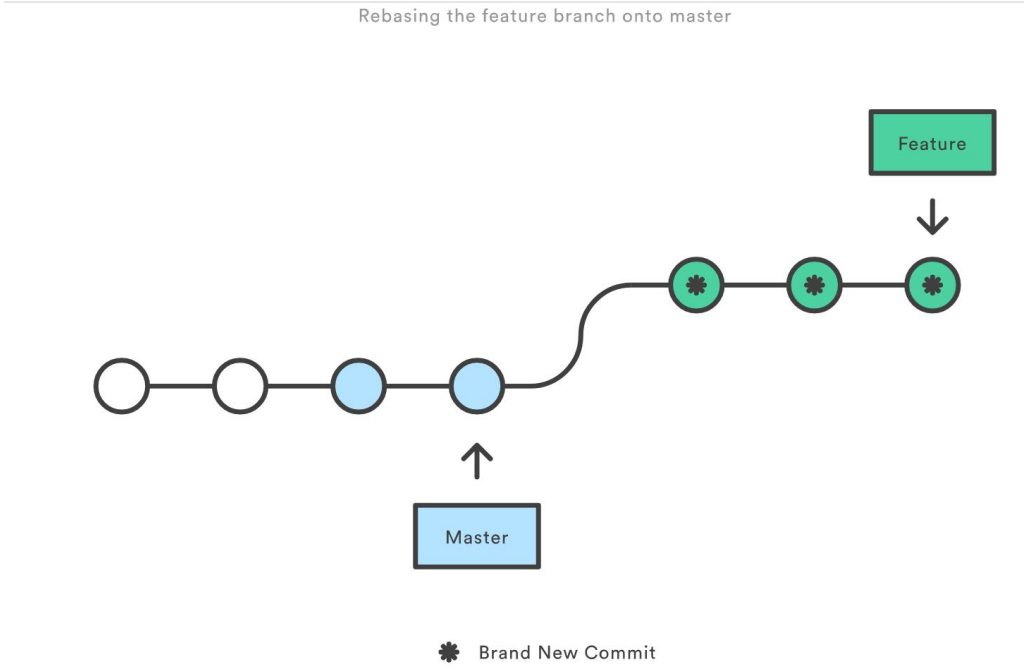
“feat: Implements save recent card”

Rebase and Merge

A forked commit history

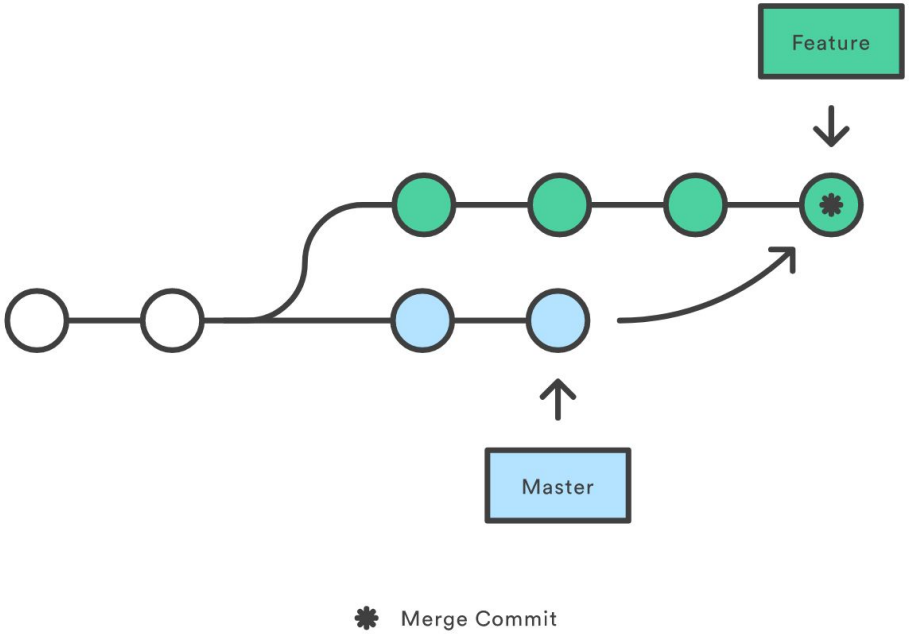


Rebase and Merge



Rebase and Merge

Merging master into the feature branch



Git Reset

To review, **git reset** is a powerful command that is used to undo local changes to the state of a **Git** repo.

Git reset operates on "The Three Trees of **Git**". These trees are the Commit History (HEAD), the Staging Index, and the Working Directory.

Git Reset

Undo Add

```
$ edit                                  (1)

$ git add frotz.c filfre.c

$ mailx                                 (2)

$ git reset                             (3)

$ git pull git://info.example.com/ nitfol  (4)
```

Git Reset

Undo a commit and redo

This is most often done when you remembered what you just committed is incomplete, or you misspelled your commit message, or both. Leaves working tree as it was before "reset".

Git Reset

Undo commits permanently

```
$ git commit ...

$ git reset --hard HEAD~3
```

Do not do this if you have already given these commits to somebody else

Git Reset

Undo a merge or pull

- **Try to update from the upstream resulted in a lot of conflicts; you were not ready to spend a lot of time merging right now, so you decide to do that later.**

Example

$ git pull **(1)**

Auto-merging nitfol

CONFLICT (content): Merge conflict in nitfol

Automatic merge failed; fix conflicts and then commit the result.

$ git reset --hard **(2)**

$ git pull . topic/branch **(3)**

Updating from 41223... to 13134...

Fast-forward

$ git reset --hard ORIG_HEAD **(4)**

Deleting commits(N last commits)

To **remove** the last **commit** from **git**, you can simply run **git** reset --hard HEAD^ If you are **removing** multiple **commits** from the top, you can run **git** reset --hard HEAD~2 to **remove** the last two **commits**. You can increase the number to **remove** even more **commits**.

Deleting commits (Desired commit)

If you need to delete more than just the last commit there are two methods you can use.

- The first is using **rebase** this will allow you to remove one or more consecutive commits.
- The other is **cherry-pick** which allows you to remove non consecutive commits.

Git Stash

- Use **git stash** when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the HEAD commit.

EXAMPLES

Pulling into a dirty tree

When you are in the middle of something, you learn that there are upstream changes that are possibly relevant to what you are doing. When your local changes do not conflict with the changes in the upstream, a simple `git pull` will let you move forward.

However, there are cases in which your local changes do conflict with the upstream changes, and `git pull` refuses to overwrite your changes. In such a case, you can stash your changes away, perform a pull, and then unstash, like this:

```
$ git pull
 ...
file foobar not up to date, cannot merge.
$ git stash
$ git pull
$ git stash pop
```

Interrupted workflow

When you are in the middle of something, your boss comes in and demands that you fix something immediately. Traditionally, you would make a commit to a temporary branch to store your changes away, and return to your original branch to make the emergency fix, like this:

```
# ... hack hack hack ...
$ git checkout -b my_wip
$ git commit -a -m "WIP"
$ git checkout master
$ edit emergency fix
$ git commit -a -m "Fix in a hurry"
$ git checkout my_wip
$ git reset --soft HEAD^
# ... continue hacking ...
```

You can use *git stash* to simplify the above, like this:

```
# ... hack hack hack ...
$ git stash
$ edit emergency fix
$ git commit -a -m "Fix in a hurry"
$ git stash pop
# ... continue hacking ...
```

How to clean up a messy branch?

Search About

- git rebase --interactive
- git merge --squash

Pull Req/Conflict/reviewer/policy