Git and GitHub

Kubilay Çiçek

July 2018

Contents

1) About Git
   a) What is Git
   b) The Three States Of Files
2) Things To Do First
   a) İdentity And Editor
   b) Checking The Settings
   c) Getting Help
3) The Fundamental Commands For Git
   a) ~ init
   b) ~ add
   c) ~ commit
   d) ~ clone
   e) Brief on those 4 commands above
   f) ~ status
   g) A Simple Example
   h) ~ ignore
   i) ~ rm (remove)
   j) ~ checkout
   k) ~ remote
   l) ~ fetch, ~ merge and ~ pull
   m) ~ push
4) General Mechanism Of Git And The General İdea Of How The Project Process İn Git Mechanism
   a) A Brief Of Branch, Master, Head And Checkout Concept And Processes
   b) An example of mechanism
   c) Merge conflicts
   d) Branch management
   e) Rebasing
5) Workflows of Git
   a) Centralized workflow
   b) Integration-Manager Workflow
   c) Dictator and Lieutenants Workflows

1) About git

a) What is Git ?

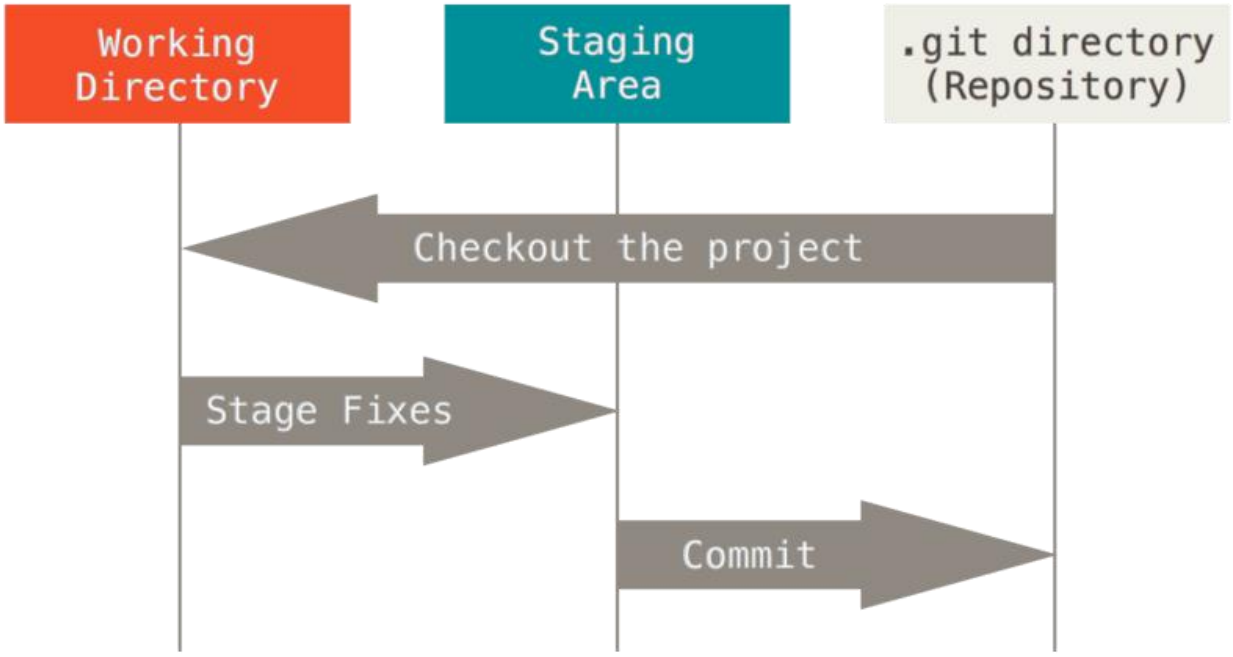
Git is basically VSC (Version-Control method of choice). Means that you can copy your work in another file or/and in another name and work on it without taking the risk losing what you have at first. So that, after you finished your enhancement you can delete the older one or change it to the newer one. Or apart from all those if you just did not like new one, you can always turn back to the first work.

Git and its program GitHub is making the process easier and less complicated for us. Because the chance of lost in the files and in the versions of the project is getting higher depending on the number of the people and the issues. The difference of Git from other VSC's it does not create any other file, just create new work which is connected to the former step. By this way you can see the project tree, and where your work is connected to (branch).

b) The Three States Of Files

Git has three main states that your files can reside in: *committed*, *modified*, and *staged*:

- Committed means that the data is safely stored in your local database.
- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.



The basic Git workflow goes something like this:

1. You modify files in your working directory.
2. You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

If a particular version of a file is in the Git directory, it is *committed*. If it has been modified and was added to the staging area, it is *staged*. And if it was changed since it was checked out but has not been staged, it is *modified*.

2) Things to do first

a) Identity and editor

To be able to use the git database you should get your user name and e-mail address. For all the works and operation you do Git saves them to be able to check your work or contribution.

```
$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com
```

To be able to use the original Git terminal you should select your text editor. You can use your own text editor like notepad++ or like emacs that comes with Git setup.

```
$ git config --global core.editor emacs
```

```
$ git config --global core.editor "'C:/Program
Files/Notepad++/notepad++.exe' -multiInst -nosession"
```

b) Check the settings

You can check your settings like user name, e-mail, colors.. anytime.

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
...
```

c) Getting help

The main command is:

```
$ git help <verb>
```

But you can get spesific help like:

```
$ git add -h
usage: git add [<options>] [--] <pathspec>...


-i, --interactive     interactive picking
-p, --patch           select hunks interactively
-e, --edit            edit current diff and apply ...
```

3) The Fundamental Commands For Git

a) ~ init

```
$ git init
```

To use Git you must connect to main file to the Git database. You can do that either using the command above or just while you are in the file make right click and select Git Bash Here.

b) ~ add

```
$ git add <file>
```

Mainly used to upload files of your work to the local Git file that you created with ~init command. This command is the first step for tracking the work.

c) ~ commit

```
$ git commit <file>
```

Now your files are tracked after you enter this command.

d) ~ clone

```
$ git clone https://github.com/libgit2/libgit2
```

If you want to get a copy of an existing Git repository, you should use this command. After that, you have the whole repository in your local file. But the important thing is that even you clone a repository, that does not means you are now in and working on that repository. Means only that you have and can enter or contribute.

e) Brief on those 4 commands above

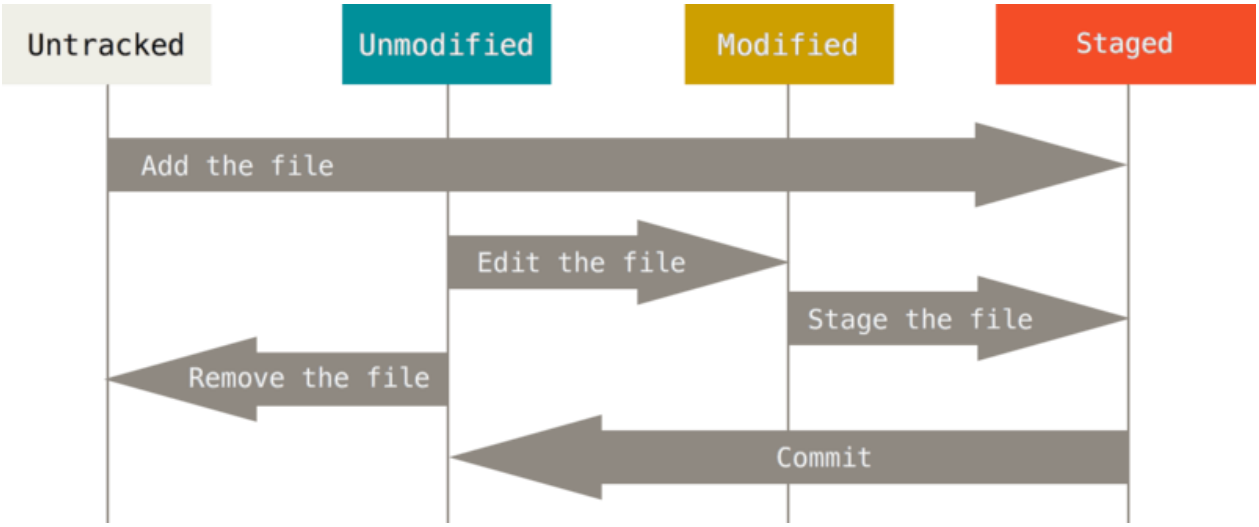
You are going to use all those 4 commands above. But they are as simple as they are dangerous.

At this point, you should have a Git repository on your local machine, and a checkout or *working copy* of all of its files in front of you. Typically, you'll want to start making changes and committing snapshots of those changes into your repository each time the project reaches a state you want to record.

Remember that each file in your working directory can be in one of two states: *tracked* or *untracked*. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.

Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven't edited anything.

As you edit files, Git sees them as modified, because you've changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.



f) ~ status

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

With this command, you can learn your current branch, your tracked or untracked files, modified or unmodified files etc. anytime you want. You should check every time before commit a file or work.

g) A Simple Example

Let's say you have a new file. Start with adding a README.

```
$ git add README
```

Now, README is now tracked and staged to be committed. But let's check our situation.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README
```

As we see that, README is just a staged not committed.

Next we have changed a file which is that repository and it is ‘CONTRIBUTING.md’. Let’s check again.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)
    modified:   CONTRIBUTING.md
```

README is still the same. But we edited the CONTRUBUTING.md. So it is still tracked, but also modified and not staged. Now:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README
    modified:   CONTRIBUTING.md
```

They are ready to commit. But let’s change CONTRUBUTING.md again. And check the status.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README
    modified:   CONTRIBUTING.md
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

There are 2 modified CONTRUBUTING.md. The upper one is our first edited one. And it is staged because we did it. But the second one is not staged because we changed it. If we commit the file before we staged the second change, we are going to lose the last edit. Now, for the last time use the ~add command to stage it to commit.

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README
    modified:   CONTRIBUTING.md
```

Now we staged and can commit the latest version.

h) ~ ignore

```
$ cat .gitignore
*.[oa]
*~
```

Sometimes there are a lot of files to add to repository. So we put '*' at the end of the add command. But we do not want some file types to commit. For these purpose we can use ignore command.

i) ~ rm (remove)

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
```

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

    deleted:    PROJECTS.md
```

You can also remove a file with this command.

j) ~ checkout

~ checkout command is very dangerous command because when you use that command without commit, it is gone for good.

Let's say that you did some changes in CONTRUBUTING.md.

```
$ git status

On branch master

Changes to be committed:

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

    renamed:    README.md -> README

Changes not staged for commit:

  (use "git add <file>..." to update what will be committed)

  (use "git checkout -- <file>..." to discard changes in working
directory)

    modified:   CONTRIBUTING.md
```

But you do not want to keep them. So you can use that command.

```
$ git checkout -- CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

    renamed:    README.md -> README
```

All those editing job is gone for CONTRUBUTING.md.

k) ~ remote

When you have more than one repository, and using them together, you can manage to know which repository you are working on, when you got, and where to push etc.

```
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
```

l) ~ fetch, ~ merge and ~ pull

For example, you have got a new repository,

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
pb     https://github.com/paulboone/ticgit (fetch)
pb     https://github.com/paulboone/ticgit (push)
```

But after a while it is updated, and you want to get the updated version and work on that version.

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Now you have the latest version in the name of pb/master. It does not merge to your work automatically. You can/have to merge into your branches or just inspect it.

Instead of ~fetch and ~merge commands, there is ~pull.

```
$ git pull <remote>
```

This automatically fetch and merge the latest version to your work.

m) ~ push

When you finish your work and want to share you should use this command.

```
$ git push origin master
```

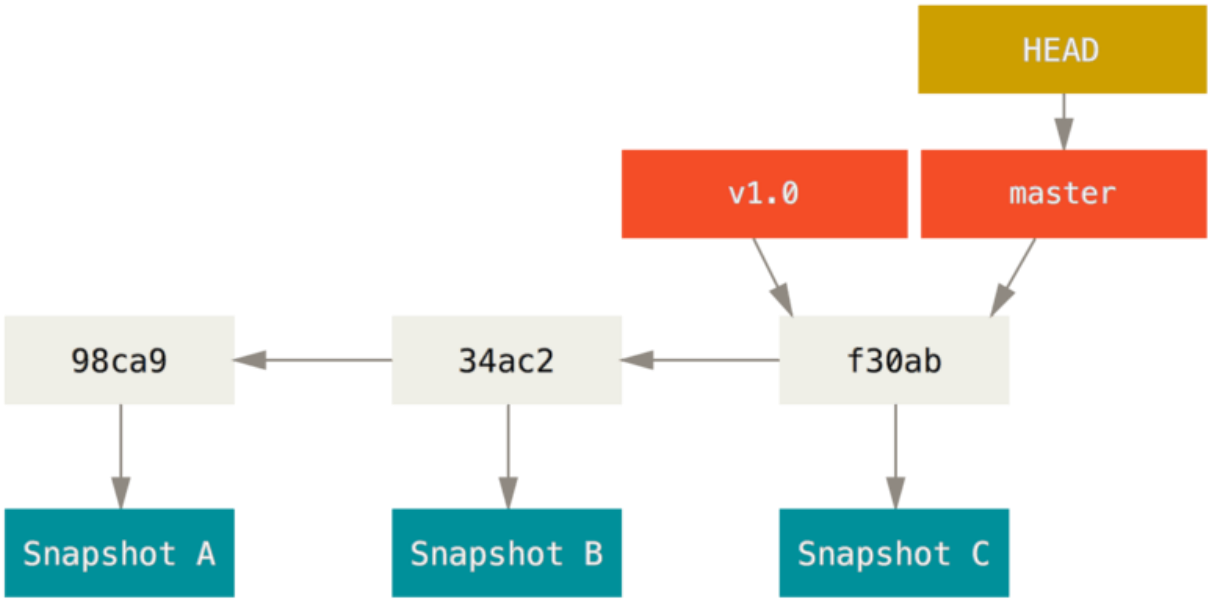
But if someone else was working on the project and push his/her work before you, you are going to be rejected. You first of all need to pull the new one, second update it with your code and finally try to push again.

4) **General Mechanism Of Git And The General İdea Of How The Project Process İn Git Mechanism**

a) A Brief Of Branch, Master, Head And Checkout Concept And Processes

Branch is an object that you use to work on. Every time you commit an object, there is a new branch. The first branch that you work on or clone called as 'Master'.

Master is basically the base branch of the one that you are working on. Every time you open a new branch you did commit the new one. When you edit and commit the new version the master become the one you committed.

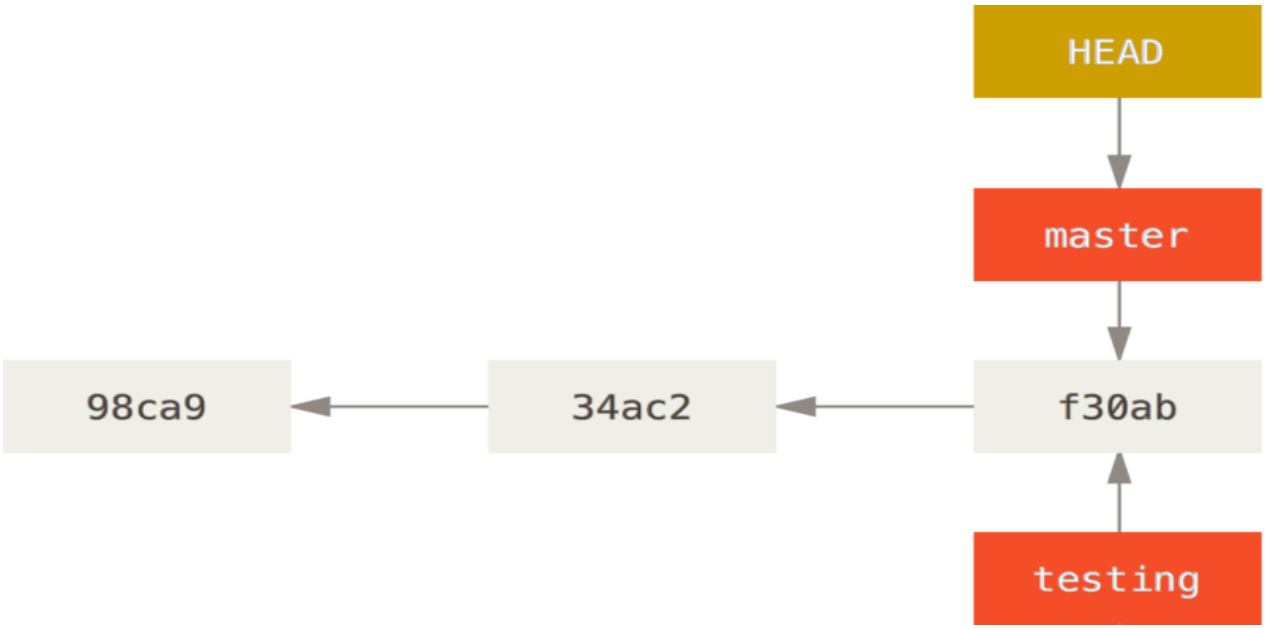


While you were editing the (B), (B) was the master and the head, there were no (C). And when you commit, it would be named as (C), the master and the head. If you would start working on the (C), master and the head would be still (C). when you commit (C), master and the head is (D).

If you want to start a new branch callad 'testing'

```
$ git branch testing
```

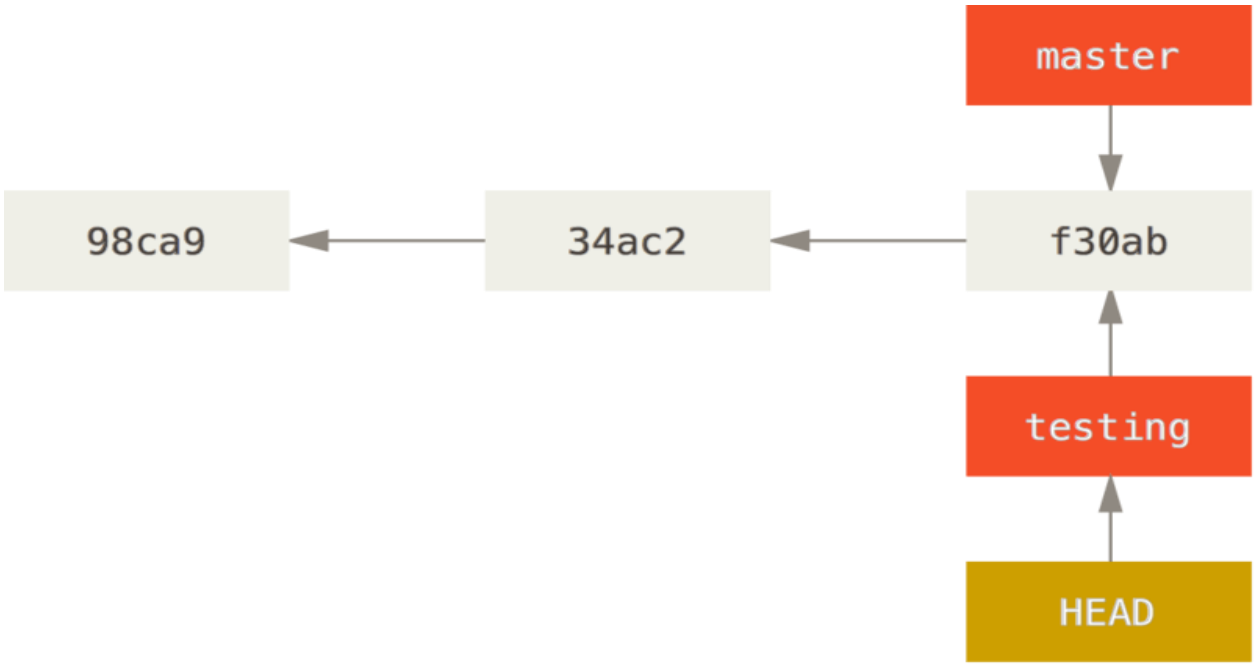
Now the tree becomes like that:



As you can see, you commit the edited (B), it becomes (C) and (C) is master. Because you open testing branch while (C) is master, (C) is also called as 'testing' branch. And the master branch is the head.

You created a new branch 'testing'. But you are still working on master branch means that master is still head. To switch to the 'testing':

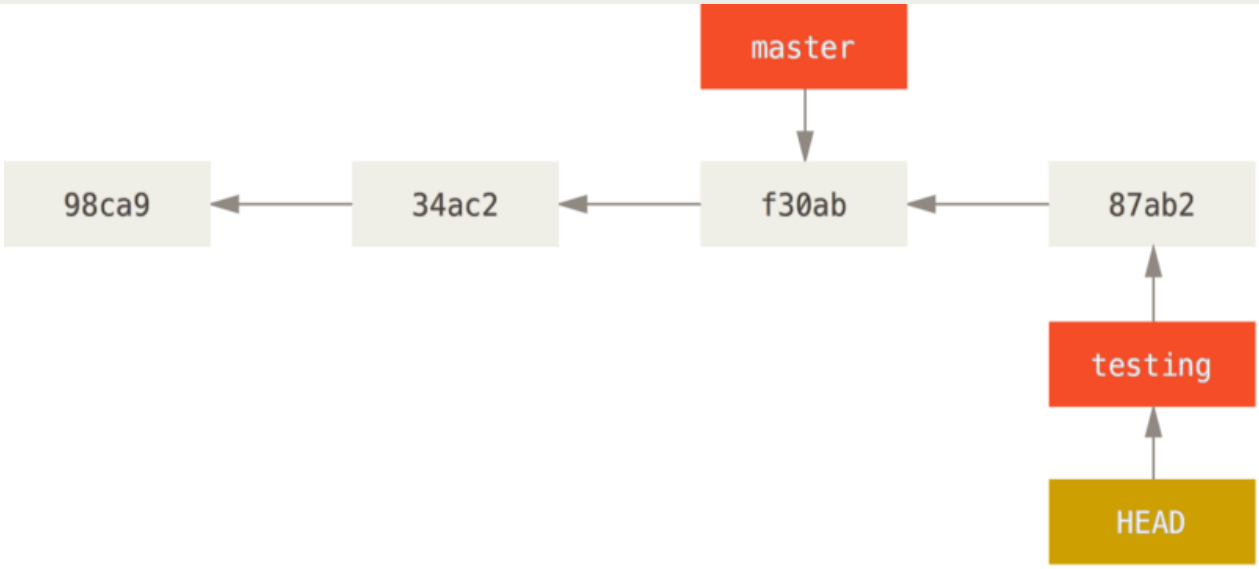
```
$ git checkout testing
```



Now you are working on 'testing' and it is the head.

After that, if you do another commit:

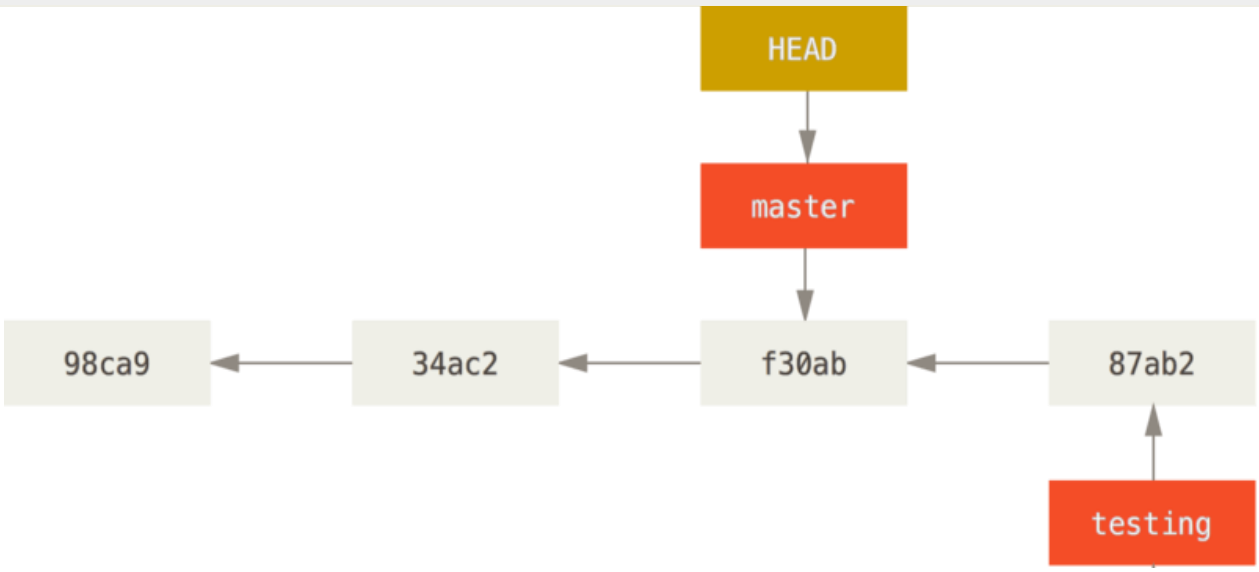
```
$ git commit -a -m 'made a change'
```



When you commit, you were working on the testing, so when you commit, master stood same, but the testing has moved forward. And it is head.

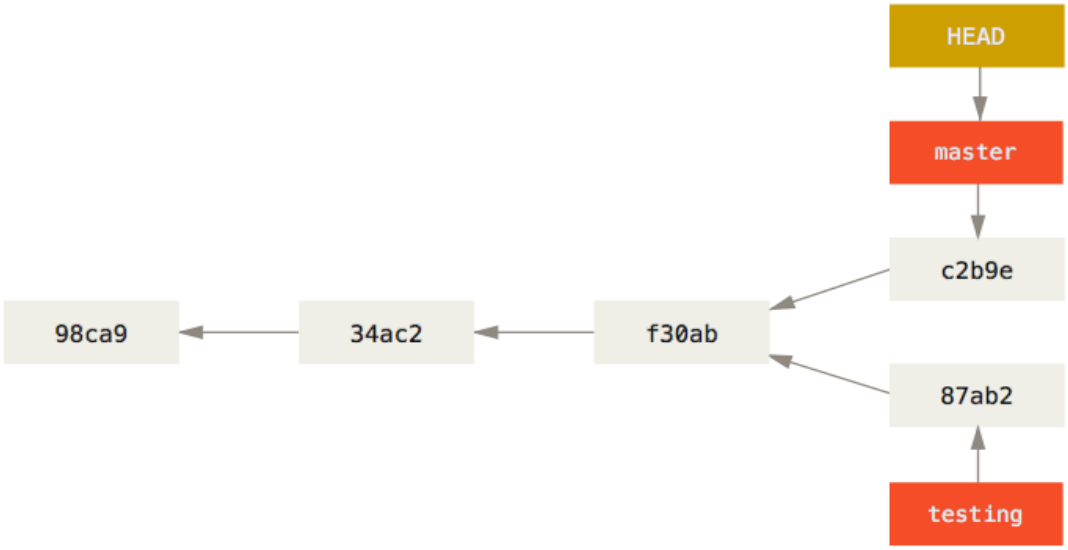
If tou switch back to master branch:

```
$ git checkout master
```



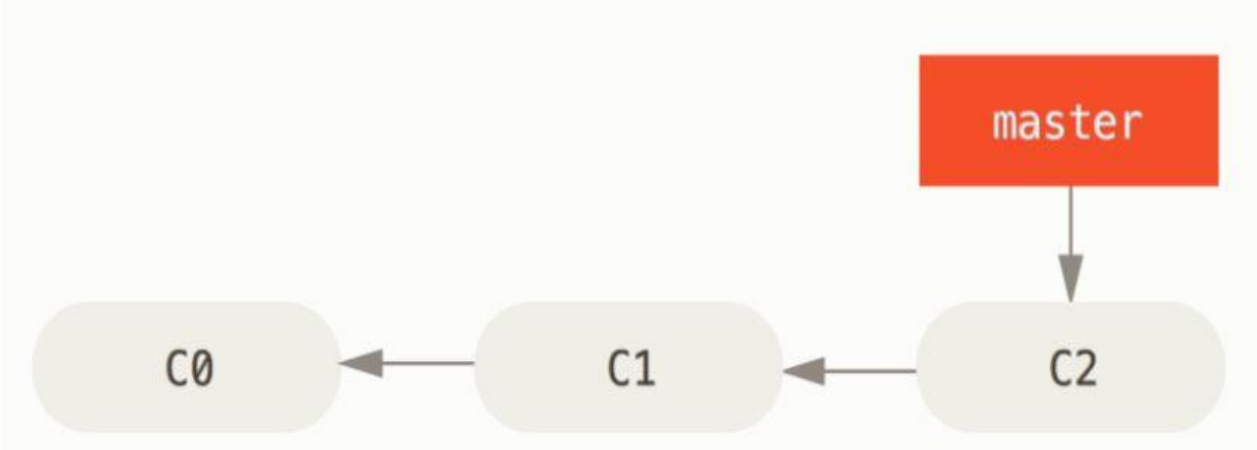
Not there is very important thing in this situation. You switched your branch to master branch, and it is head. But the important thing is that, you just reverted all your files to master branch. The latest workings are still on the testing branch. But if you commit again while in this position (while your head is master) there will be another arm that is apart from testing branch.

```
$ git commit -a -m 'made other changes'
```



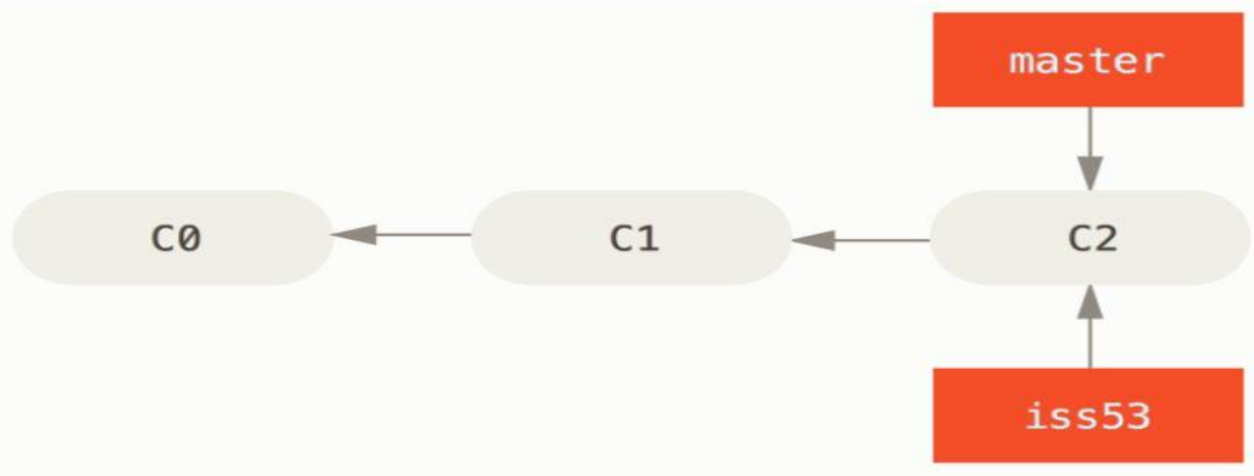
b) An Example Of Mechanism

Let's assume you have some commits already.



You are going to work on 'issue #53'. Create the new branch and checkout to it:

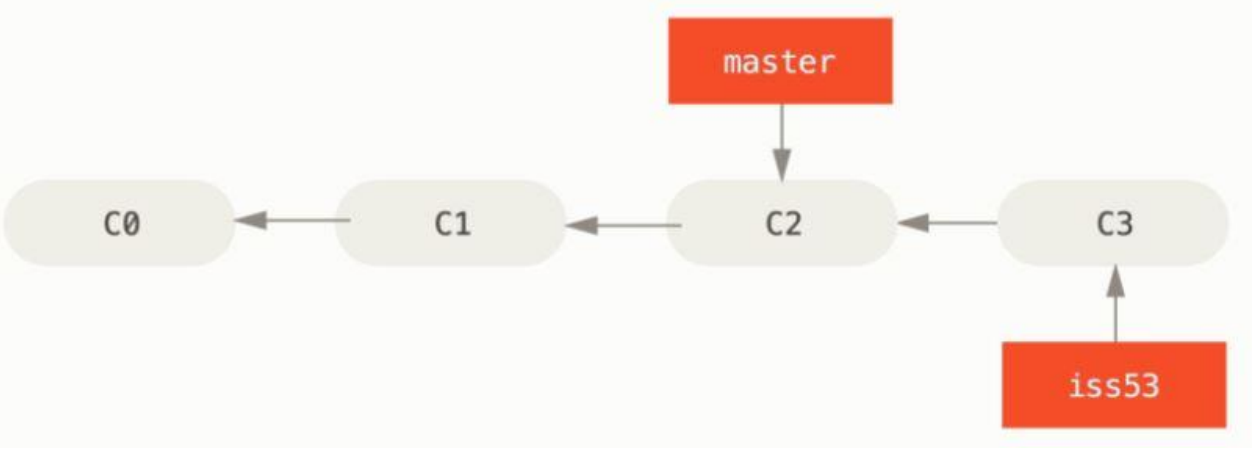
```
$ git branch iss53
$ git checkout iss53
```



Now you are working on issue #53. And the head is issue #53.

You finished and commit it:

```
$ git commit -a -m 'added a new footer [issue 53]'
```

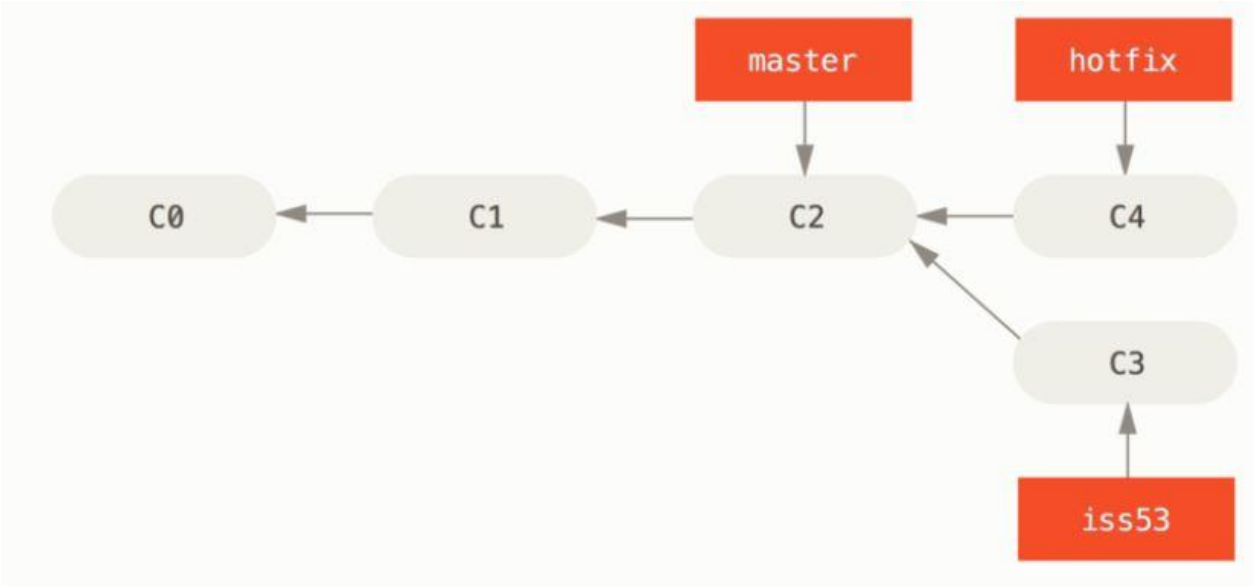


Then, you need to do some work on master branch too. Need to switch to it:

```
$ git checkout master
Switched to branch 'master'
```

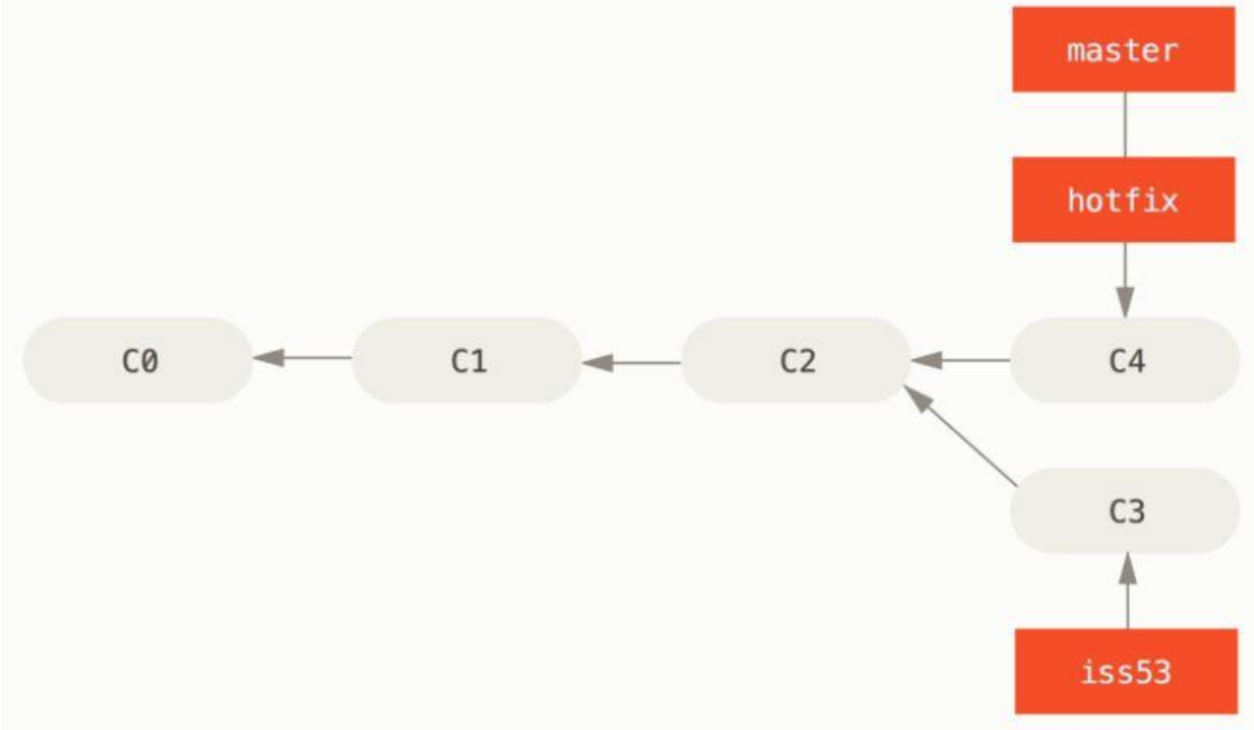
Now the head is master. The work is called ‘hotfix’:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```



When you are OK with hotfix branch, you can merge master branch with it.

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

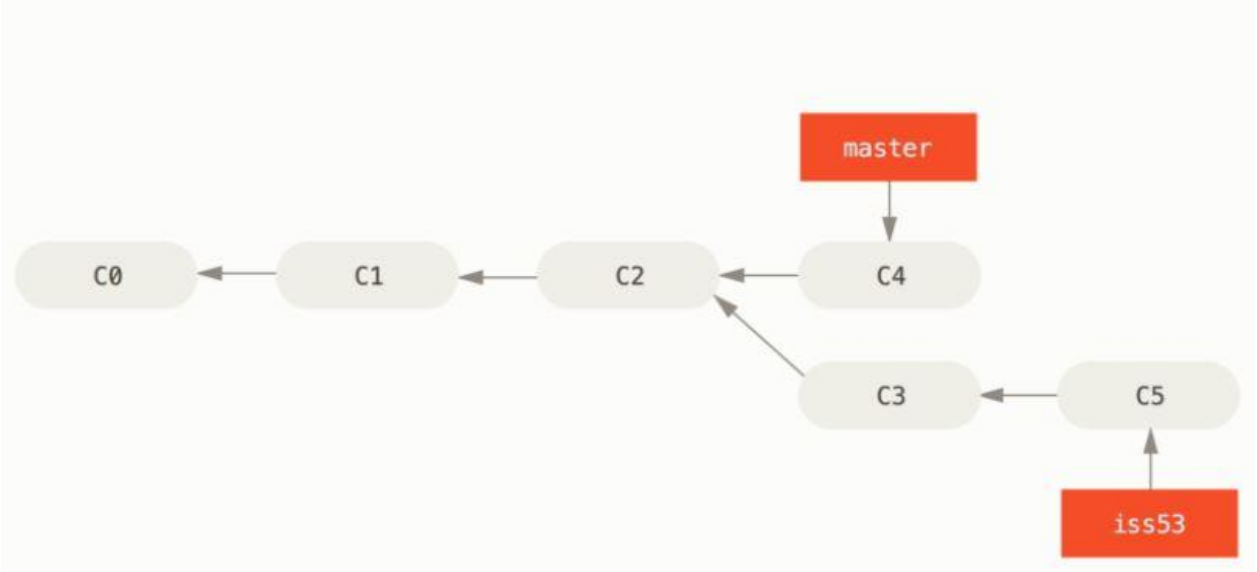


As you can see, because the hotfix is directly the newer version of the master, master just ‘fast-forward’ to the hotfix branch. So that you can delete the hotfix, just to prevent the possible complications in the future.

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Now you can go back to issue #53 and working on it.

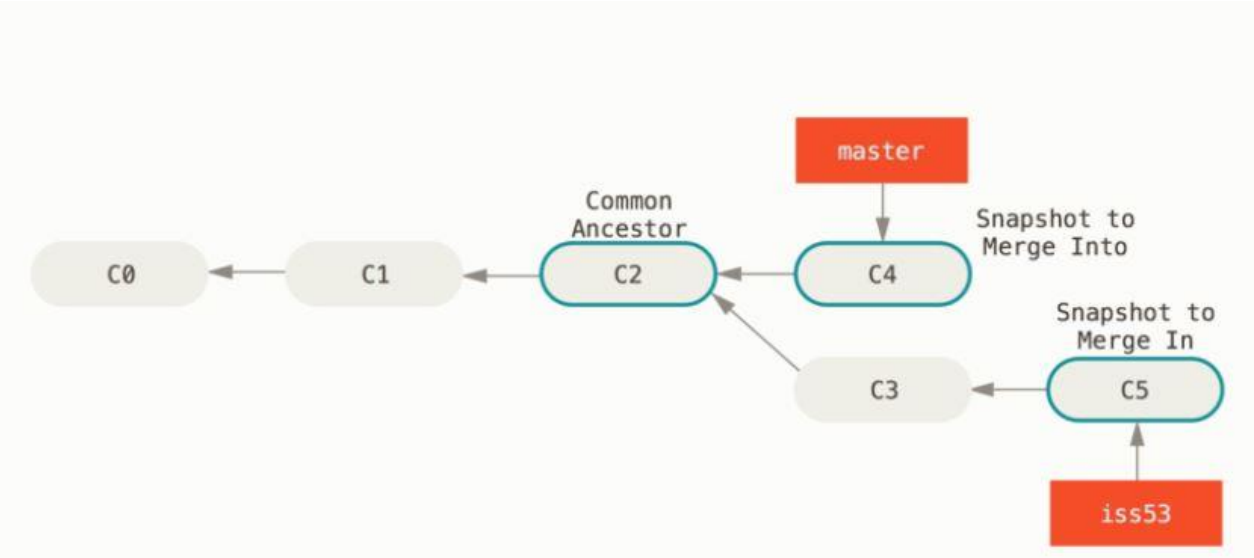
```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

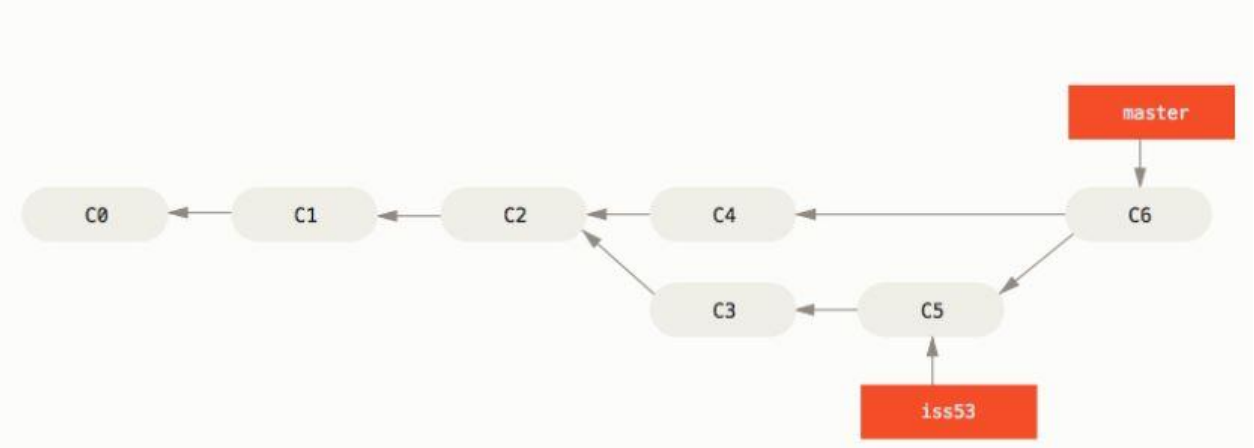


You see that the new branch comes after the 2. arm. Because you were on the iss53 branch. After that you can merge the two branches.

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

There are more than one parent, so it has to point out to all parents.





So, you have finished your work. And the final project is the master branch. Every update is in the master branch. Then, you can delete iss53 if you want.

```
$ git branch -d iss53
```

c) Merge conflicts

As we speak before, while merging the branches, if the older one is updated, means someone else also worked on that branch, you cannot just merge them. Git is going to say that there are conflicts, so you need to find and fix them.

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

If you do a status check:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:      index.html
no changes added to commit (use "git add" and/or "git commit -a")
```

At this step, you can find the differences at the code in this shape:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

When you delete the HEAD, iss53 and the === lines and fix the code, you can move forward.

d) Branch management

While you are doing the project, there are a lot of branches that you created, changed, deleted etc. you can check the existing branches and the last commit on each branch by:

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

Notice that the * symbol shows you which branch you are in, means it is the head.

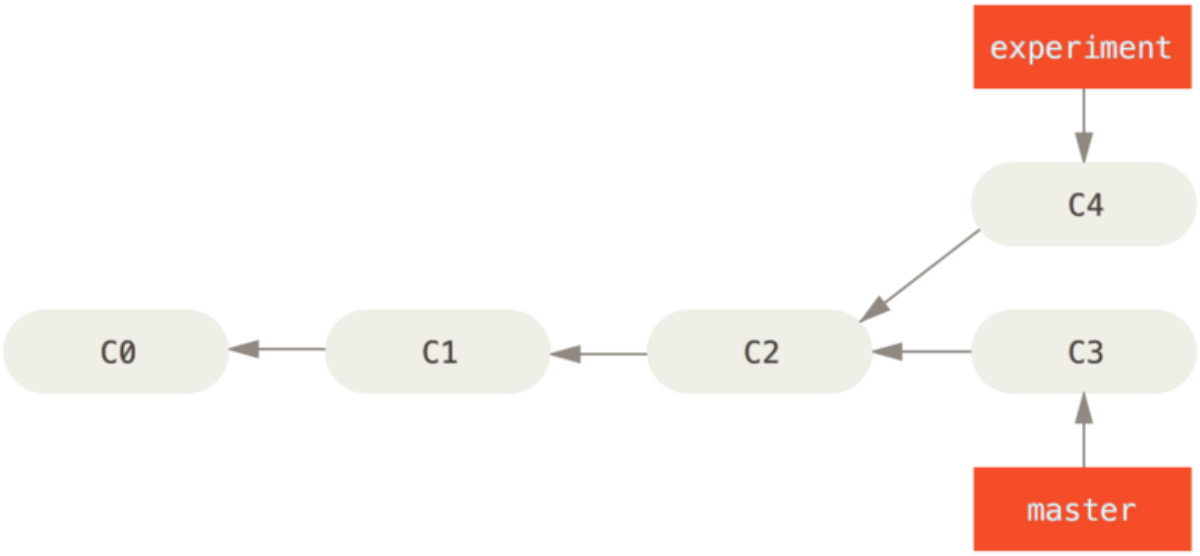
You can also check whether the branches are merged or not by:

```
$ git branch --merged
  iss53
* master
$ git branch --no-merged
  testing
```

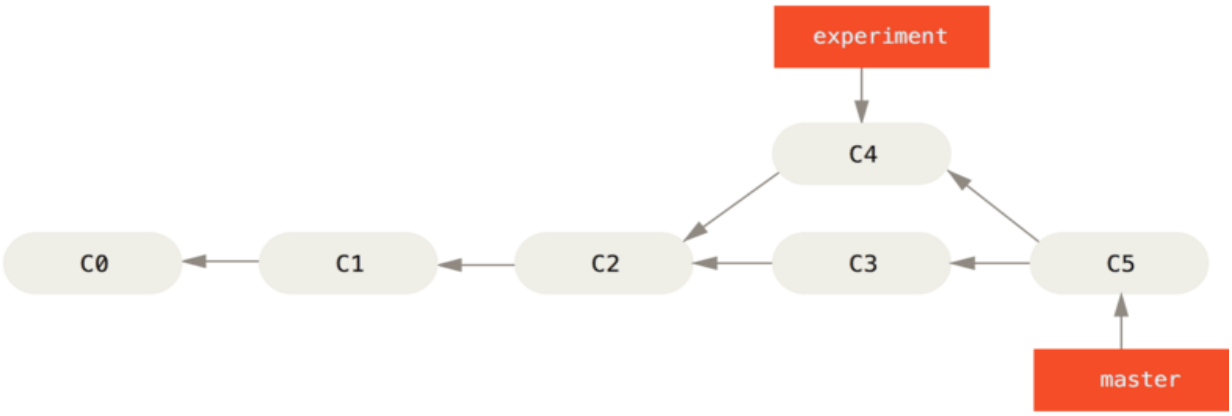
e) Rebasing

Rebasing is a type of merging. Its advantage is if you use rebasing, your path will be clear. But the thing is that it is hard to follow back when you need.

For example let's take the previous example and call the hotfix branch as experiment.

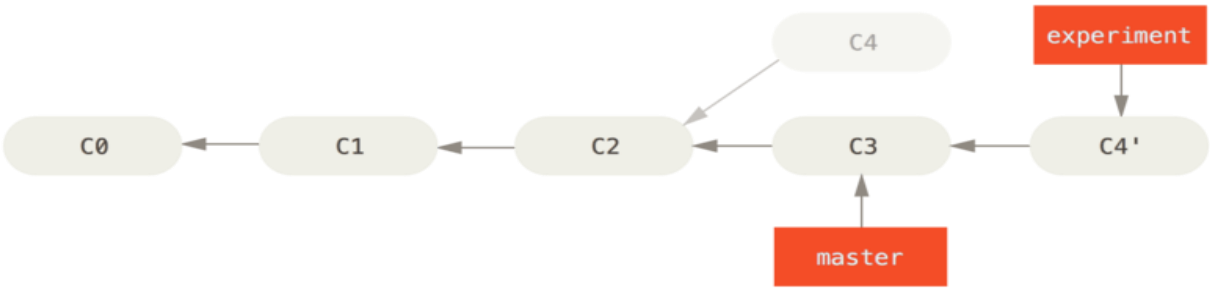


Now, if we use merge command:



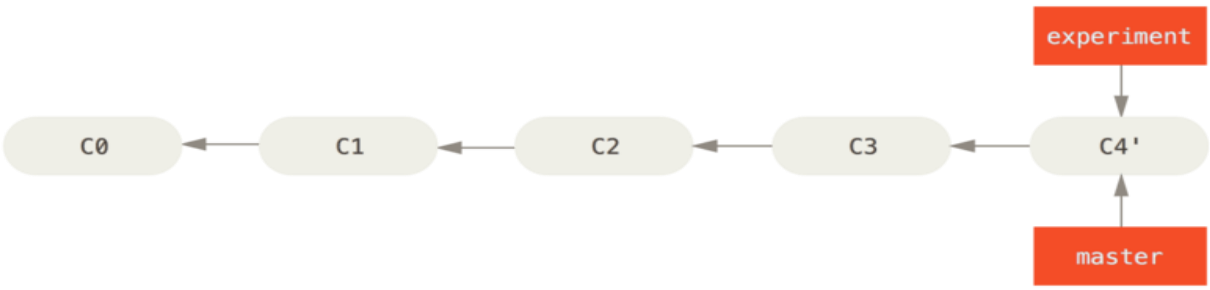
As can be seen, there is another arm, however if we use rebase:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```



The master is C3 and there is experiment comes after the master. Like a newly edited branch. Now we can fast-forward the master:

```
$ git checkout master
$ git merge experiment
```

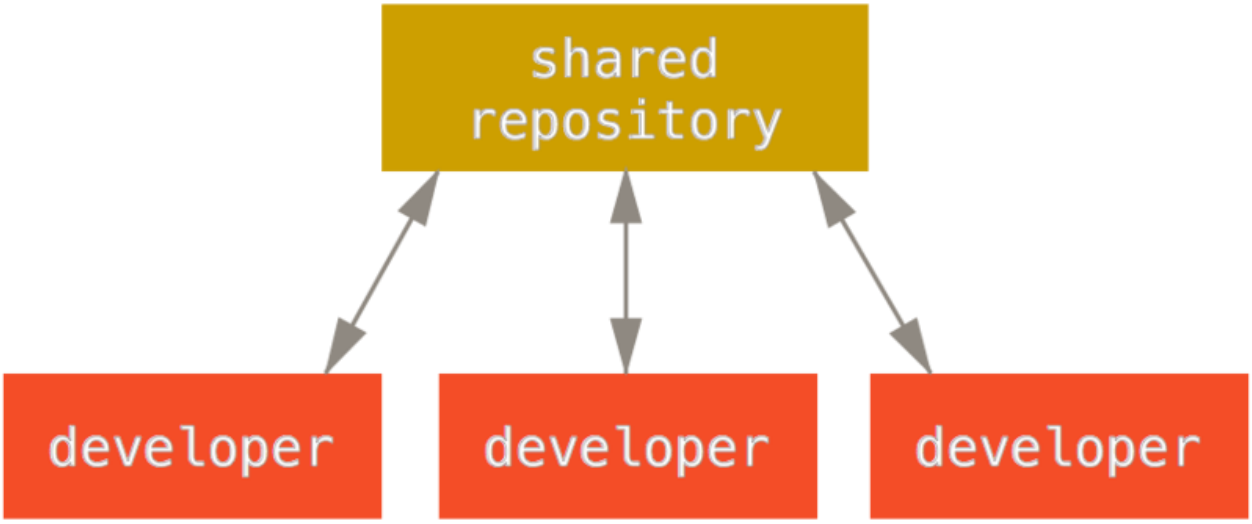


Finally, we have that smooth line.

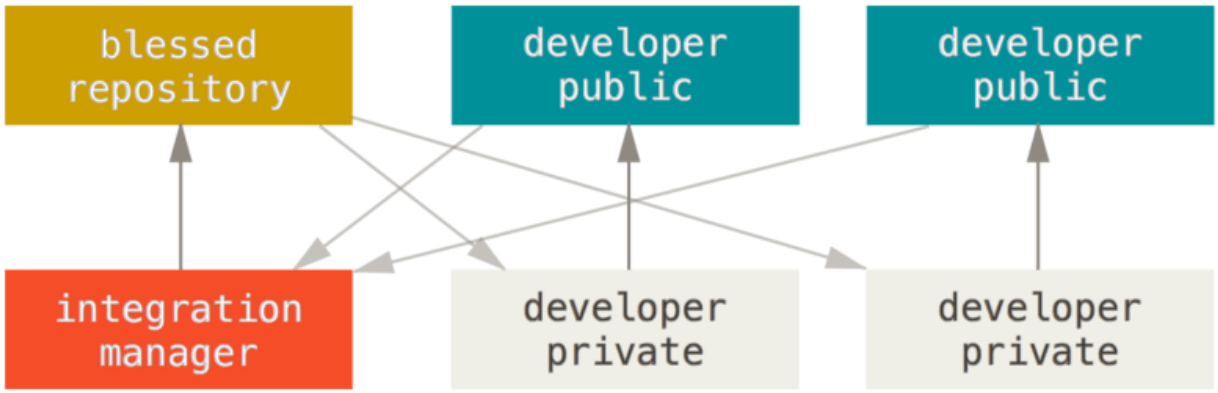
5) Workflows of Git

a) Centralized workflow

There is no manager or control mechanism in this type of workflow. And we are not going to use this type.



b) Integration-Manager Workflow



This is the appropriate one that we are going to use. There is a controller (manager), main repository that can only be changed by manager, private developers that can use pull request and public developers who are simply have the same rights with privates but they are not in the project directly.

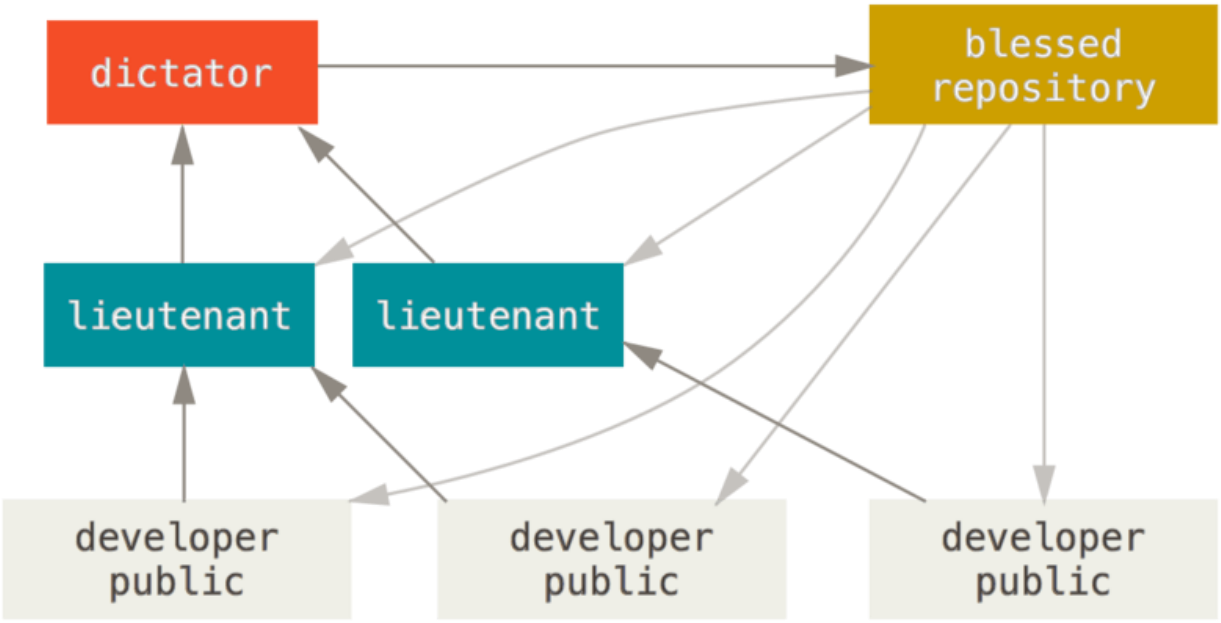
Main thing in this workflow, every developer is using their own cloned repository. The privates have the write access to the repository, but have only read access to the other developers' repository. They pushes their own repo and pull request to the manager. If manager change the main repo, the other developers have to fetch and merge the new one to their repo so that they can have the pull request.

The advantage of this, developers do not have to wait for the other developers to proceed. They can work on their issue and when it finished they can start the new issue. That means the work is always going on.

Process works as follows:

1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an email asking them to pull changes.
5. The maintainer adds the contributor's repository as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.

c) Dictator and Lieutenants Workflows



This workflow type is also not suitable for us. Because it is generally used in very large scale of projects with hundreds of contributors.