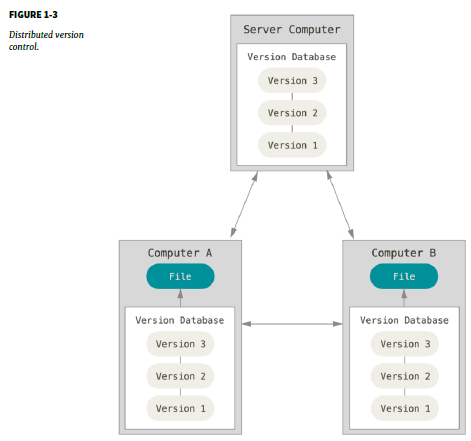
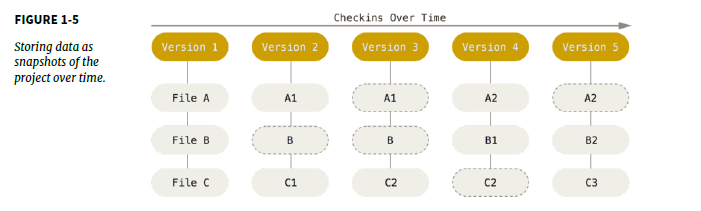
* Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.
* In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don’t just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data. Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project.



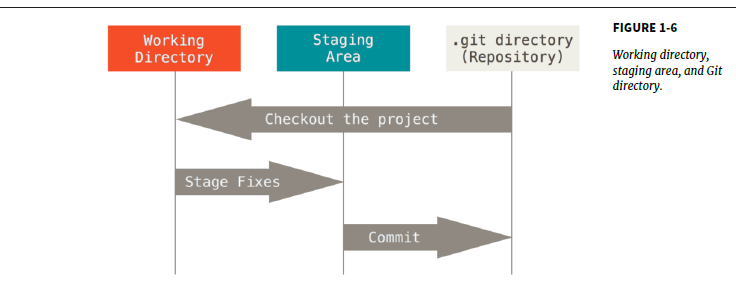
* Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**.

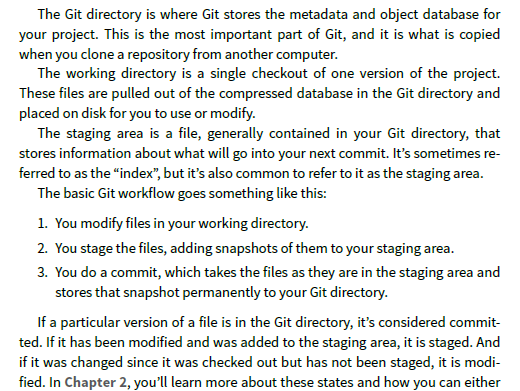


* Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous thus, most operations have that network latency overhead.
* To browse the history of the project, Git doesn’t need to go out to the server to get the history and display it for you – it simply reads it directly from your local database. This means you see the project history almost instantly. If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally. This also means that there is very little you can’t do if you’re offline or off VPN.
* Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it’s impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can’t lose information in transit or get file corruption without Git being able to detect it. The mechanism that Git uses for this check summing is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this: 24b9da6552252987aa493b52f8696cd6d3b00373
* You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything in its database not by file name but by the hash value of its contents.

**The Three States:**

* 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. This leads us to the three main sections of a Git project: **the Git directory, the working directory, and the staging area.**

****

****

****

**Important links for github:** [**https://desktop.github.com/**](https://desktop.github.com/)

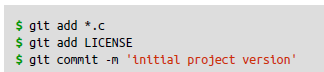
[***https://git-for-windows.github.io/***](https://git-for-windows.github.io/).

**Initializing a repository in an existing directory:**

* If you’re starting to track an existing project in Git, you need to go to the project’s directory and type:

**$ git init**

This creates a new subdirectory named .git that contains all of your necessary repository files – a Git repository skeleton. At this point, nothing in your project is tracked yet. If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few git add commands that specify the files you want to track, followed by a git commit:

****

**At this point, you have a Git repository with tracked files and an initial commit**

If you want to get a copy of an existing Git repository into the project folder you are working in the command you need is **git clone**. Every version of every file for the history of the project is pulled down by default when you run git clone. In fact, if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned.

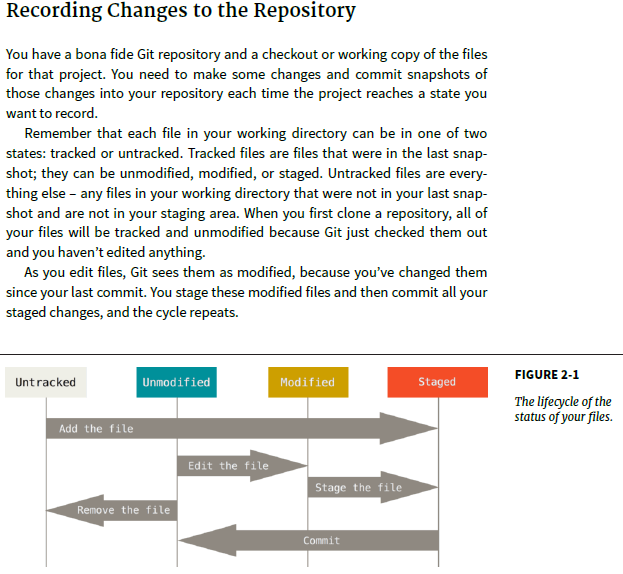
**> git clone** [**https://github.com/Navdeep-Kalia/sampleTest**](https://github.com/Navdeep-Kalia/sampleTest)

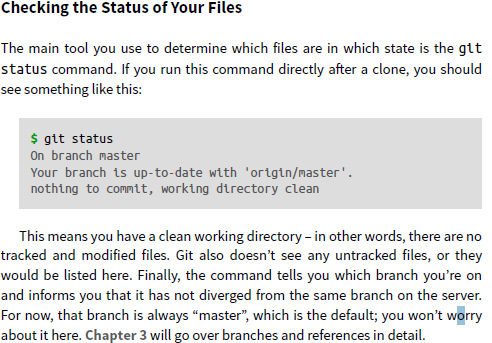
**This command will copy the repository sampleTest in the folder you are working currently from online source**

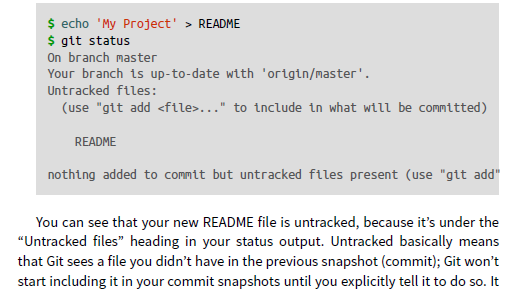
**OR**

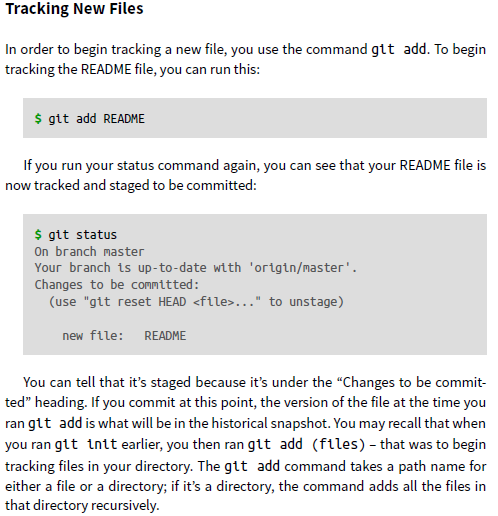
**> git clone https://github.com/Navdeep-Kalia/sampleTest notSample**

**This command will copy the repository sampleTest in the folder you are working currently from online source with given name notSample**

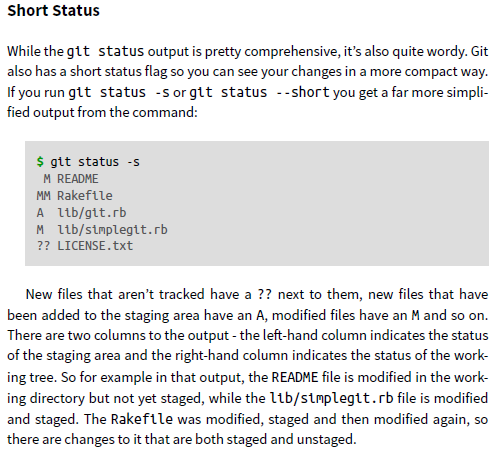
****

****

****

****

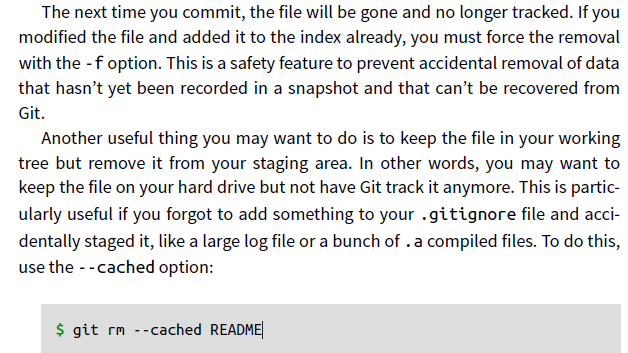
**Git add** is a multipurpose command – you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add this content to the next commit” rather than “add this file to the project”.

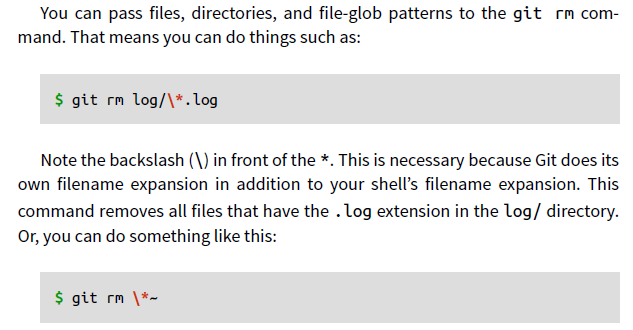
****

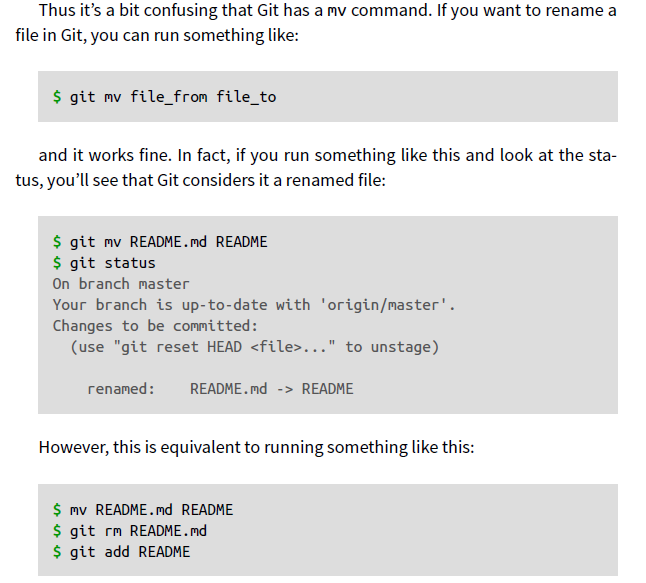
**To create git ignore file in windows using shell is:**

touch .gitignore

* Setting up a .gitignore file before you get going is generally a good idea so you don’t accidentally commit files that you really don’t want in your Git repository. Patterns inside the .gitignore file are matched from the root directory of the git repository. Patterns are comprised of a wildcard character **\***, to match any character, and literal characters to match the exact phrase
* If the git status command is too vague for you – you want to know exactly what you changed, not just which files were changed – you can use the git diff command. We’ll cover git diff in more detail later, but you’ll probably use it most often to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit? Although git status answers those questions very generally by listing the file names, git diff shows you the exact lines added and removed – the patch, as it were. The diff command matches what’s in your working directory and what is in our staging area.
* If you want to see what you’ve staged that will go into your next commit, you can use **git diff --staged**. This command compares your staged changes to your last commit. It’s important to note that git diff by itself doesn’t show all changes made since your last commit – only changes that are still un-staged. This can be confusing, because if you’ve staged all of your changes, git diff will give you no output.
* **git diff --cached** to see what you have staged so far (--staged and --cached are synonyms)

****

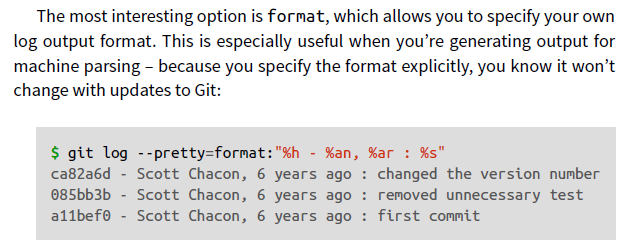
****

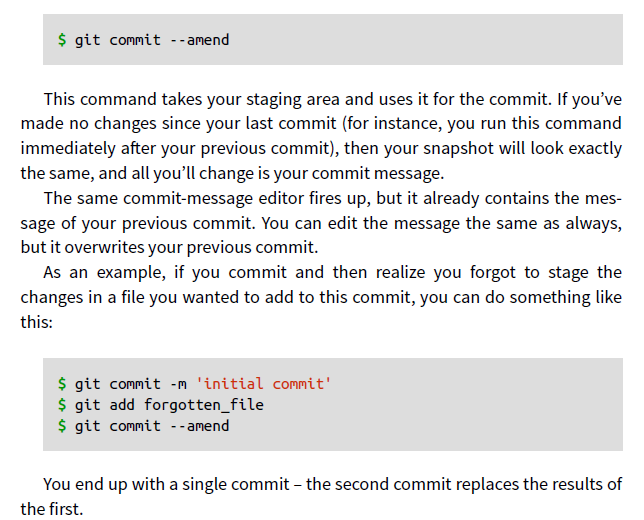
****

**Git log** command is used to check the history of commits. It lists the commits made in the repository in reverse chronological order. One of the more helpful options is -p, which shows the difference introduced in each commit. You can also use -2, which limits the output to only the last two entries:

**Git log –p -2:** this option display history of the commits with diff values as well that means full detail about all the changes made.

**Git log --stat:** if you want to see some abbreviated stats for each commit, you can use the --stat option

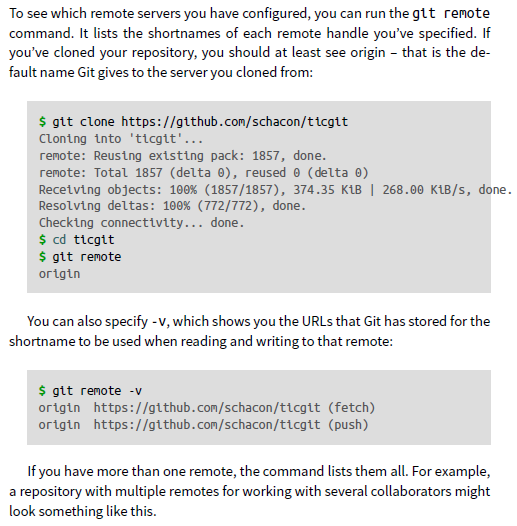


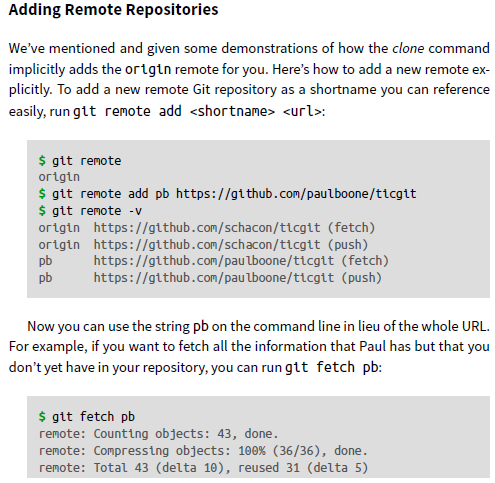
****

USE: **git reset HEAD <file>...** to un-stage.

USE: **git checkout -- pract3.txt**… used to reset the recent modified file back to the one in staging area.

* To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches and define them as being tracked or not, and more. In this section, we’ll cover some of these remote-management skills.

****

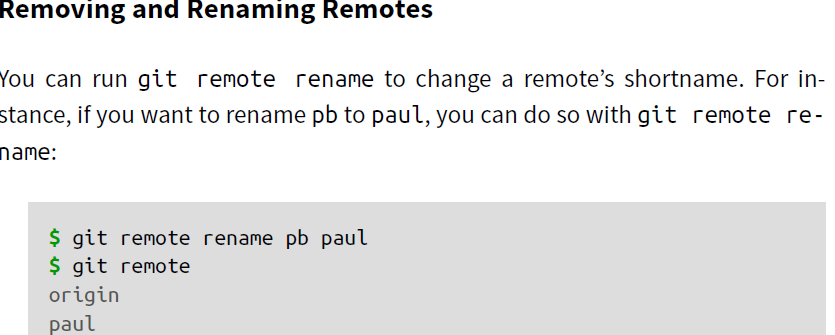
****

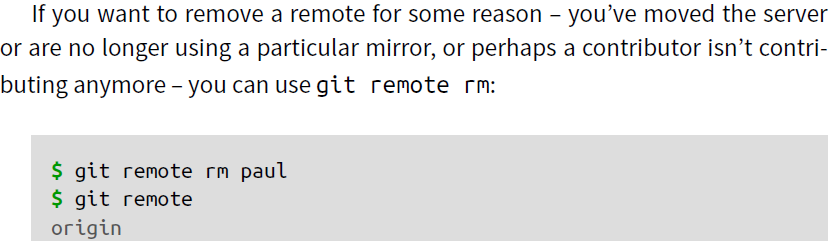
If you clone a repository, the command automatically adds that remote repository under the name “origin”. So, **git fetch origin** fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It’s important to note that the git fetch command only downloads the data to your local repository – it doesn’t automatically merge it with any of your work or modify what you’re currently working on. You have to merge it manually into your work when you’re ready. If your current branch is set up to track a remote branch you can use the git pull command to automatically fetch and then merge that remote branch into your current branch. This may be an easier or more comfortable workflow for you; and by default, the git clone command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is

called) on the server you cloned from. Running git pull generally fetches data

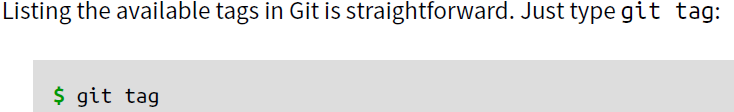
from the server you originally cloned from and automatically tries to merge it

into the code you’re currently working on.

****

****

Like most VCSs, Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points.

****

**To search for tags with particular pattern:**

****

A lightweight tag is very much like a branch that doesn’t change – it’s just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. They’re check summed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).



You can see the tag data along with the commit that was tagged by using the git show command:

**$ git show v1.4**

Another way to tag commits is with a lightweight tag

**$ git tag v1.4**

**To check the log for all**

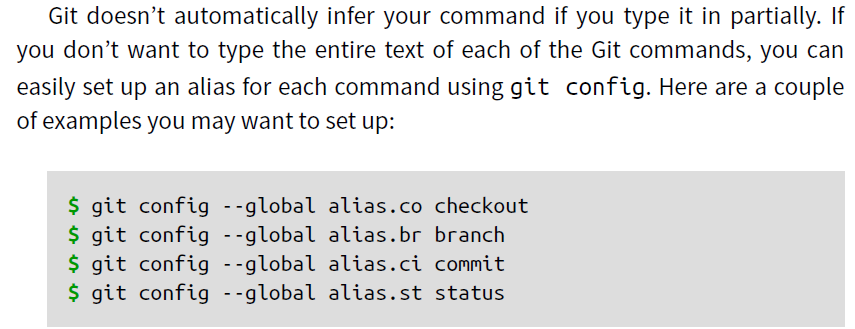
**$ git log - - pretty=oneline**

**To push version on the the git remote**

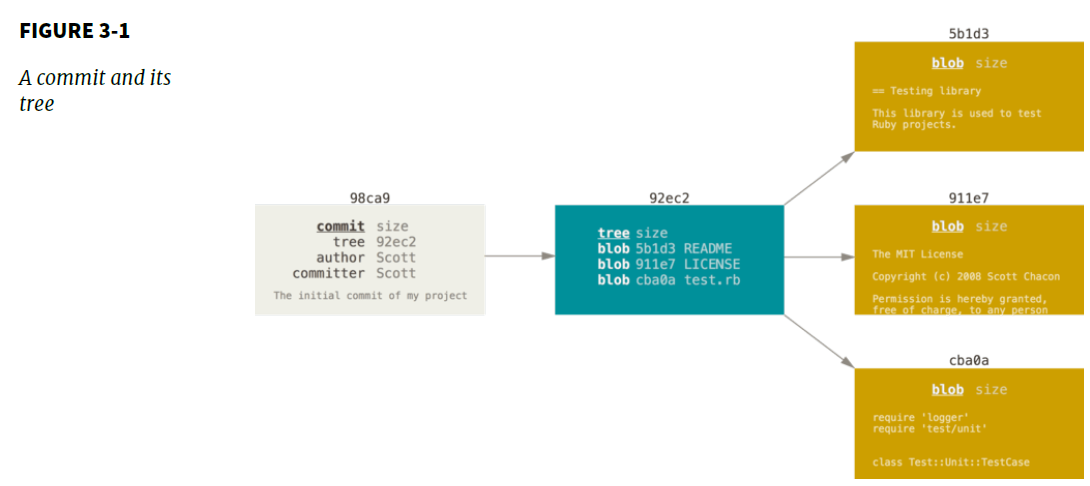
**$ git push origin v1.5**

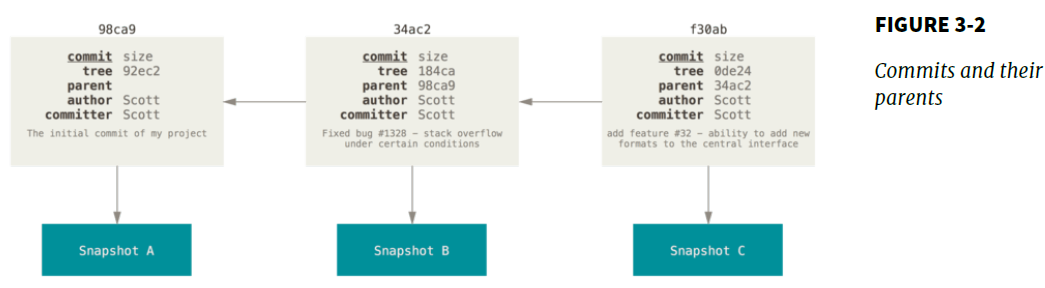
**To push all the tags to remote**

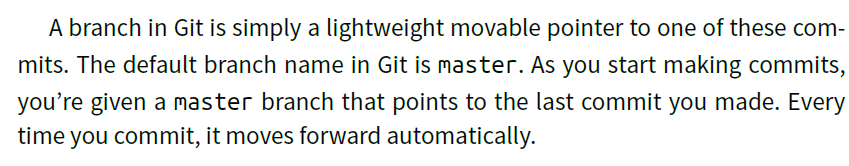
**$ git push origin - -tags**

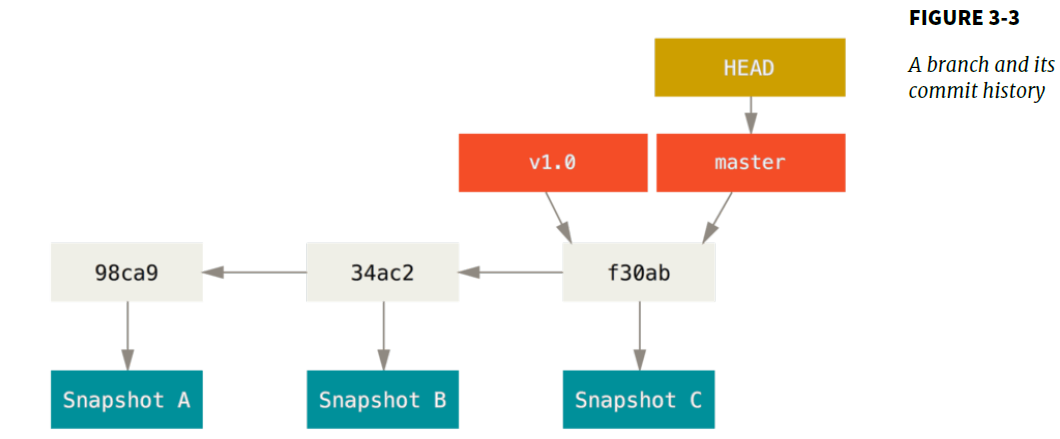
****

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author’s name and email, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

****

****

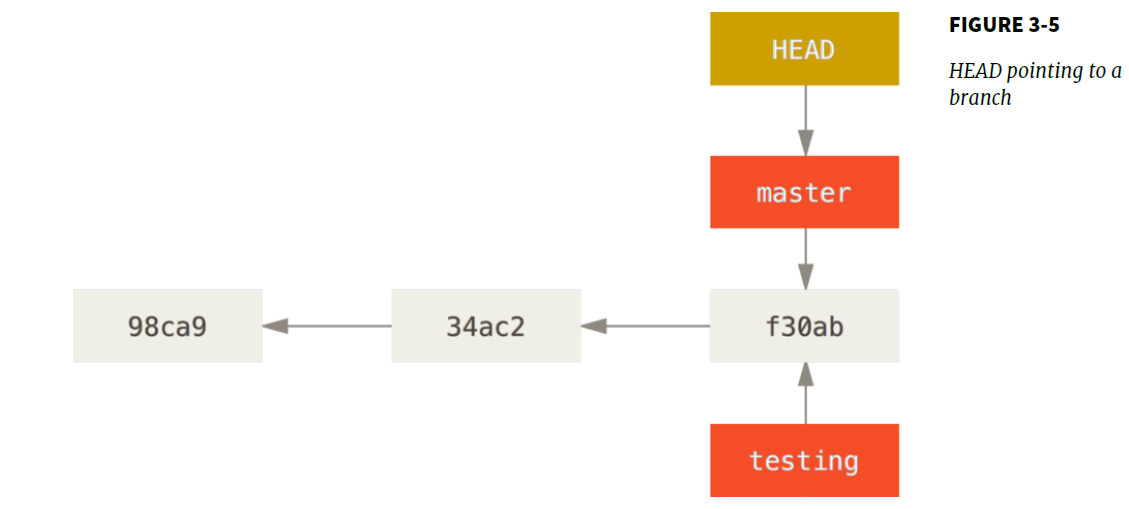
****

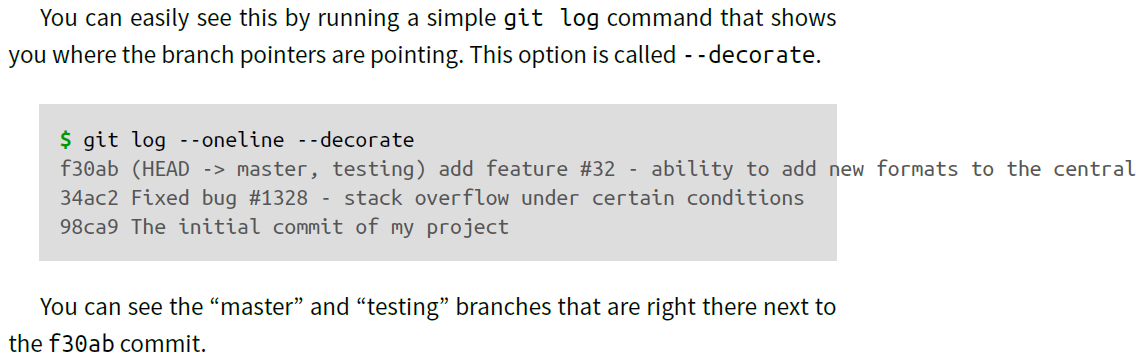
****

**How does Git know what branch you’re currently on?**

It keeps a special pointer called HEAD. Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer

to the local branch you’re currently on. In this case, you’re still on master. The git branch command only *created* a new branch – it didn’t switch to that branch.



****

**To create branch**

$ git branch testBranch

**To switch to an existing branch--** This moves HEAD to point to the testing branch

$ git checkout testBranch

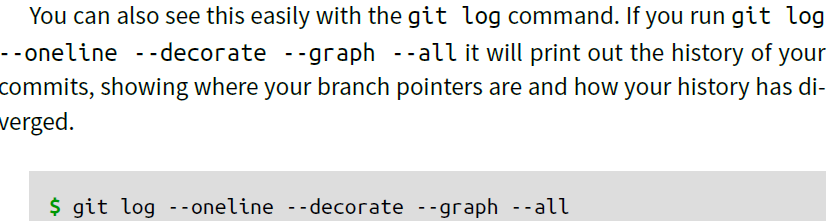
*After doing commit from branch the head moves to the branch but the master commit branch will still point to the same commit we made before checking out to the new branch*

**If u want to commit from branch other than master branch to remote use the following:**

$ git push - -set – upstream origin testBranch **this will create a new branch at remote location.**

**To remove the changes you did in the file u can do**

$ git stash **this will remove the changes from the file and bring it to a stage same as previous commit**

****

**Note:**

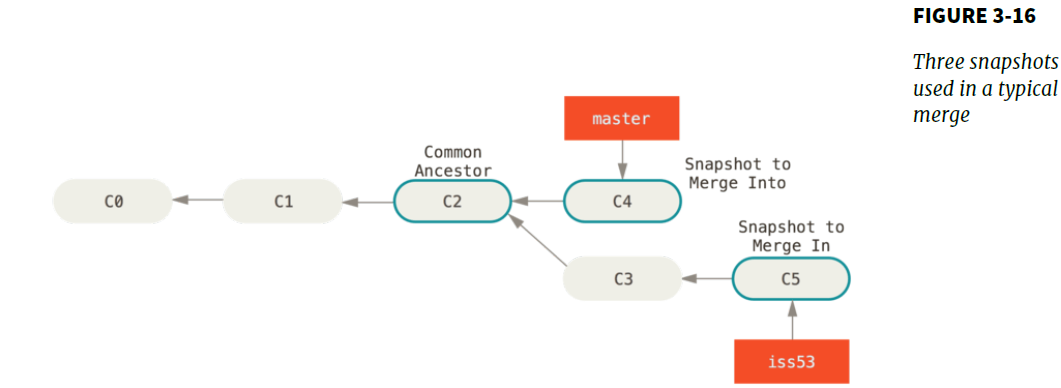
**When you do git merge the branch and master branch will both point to the same point and if u want to delete the other branch its perfectly ok as master branch is there pointing to same snapshot**

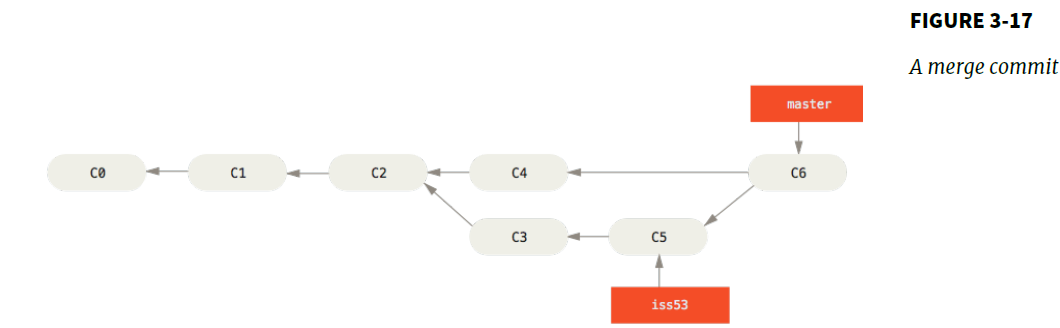
You can delete it with the -d option to git branch:

**$ git branch -d hotfix**

**When merging master with first branch which different than second branch**

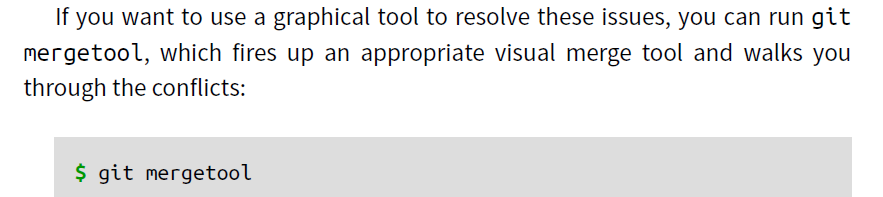
Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.



****

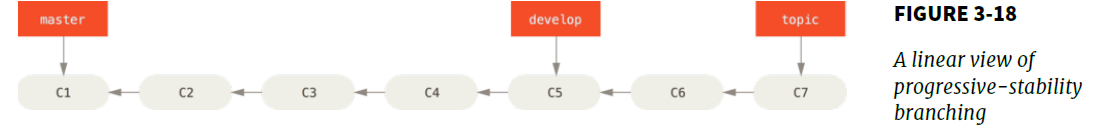
**To delete the branch**

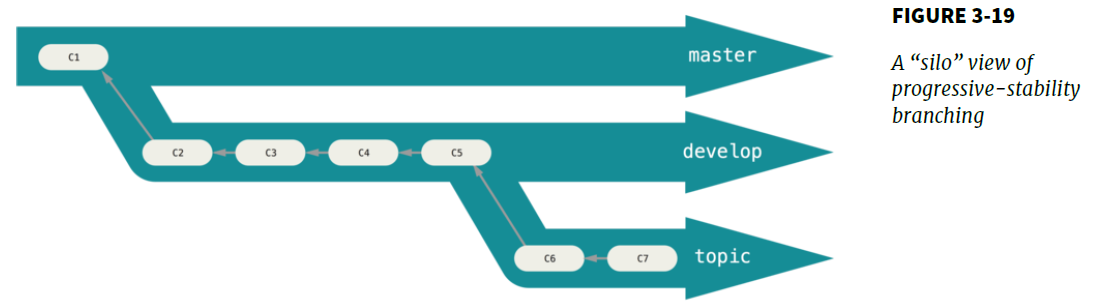
**$** git branch -d iss53



**To see the last commit on each branch, you can run git branch -v:**

**$** git branch –v

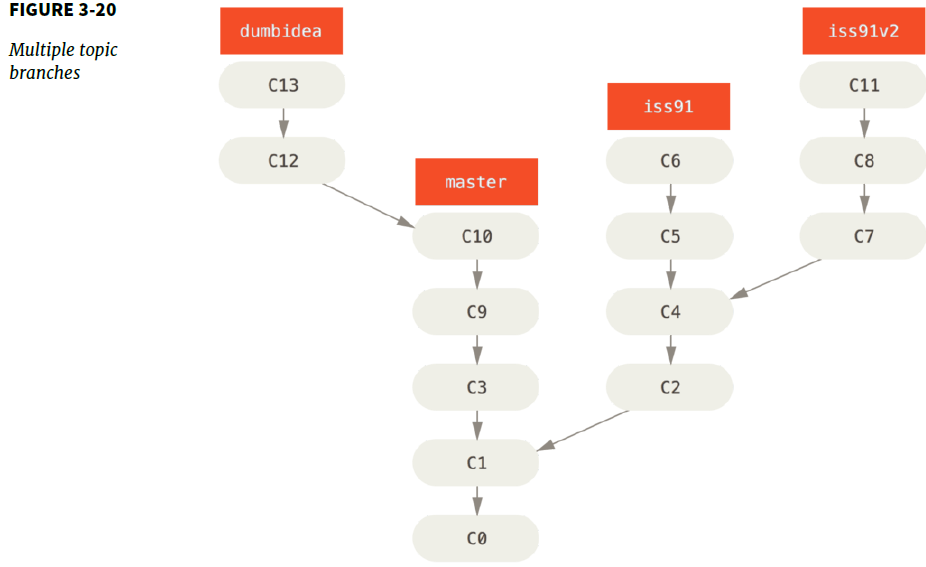
****

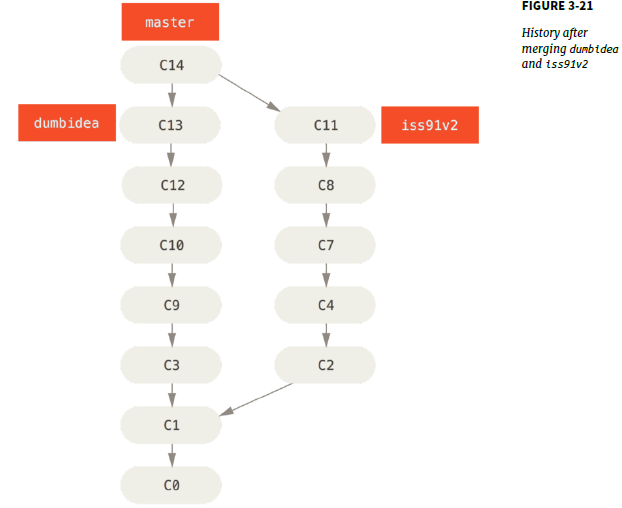


**Long Running Branches:**

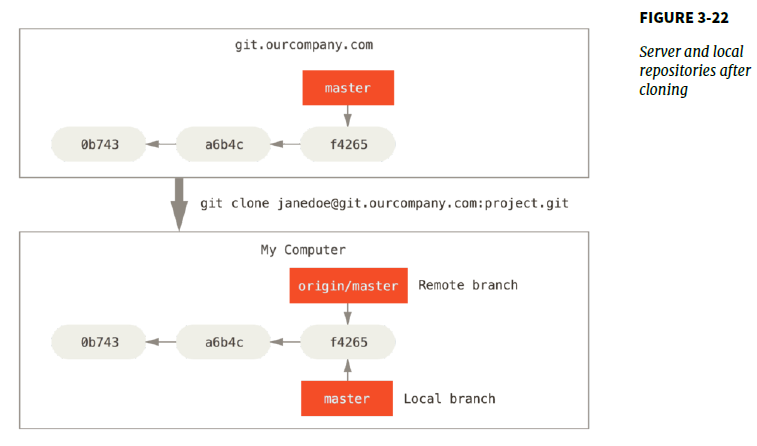
Because Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do. This means you can have several branches that are always open and that you use for different stages of your development cycle; you can merge regularly from some of them into others.

A topic branch is a short-lived branch that you create and use for a single particular feature or related work.

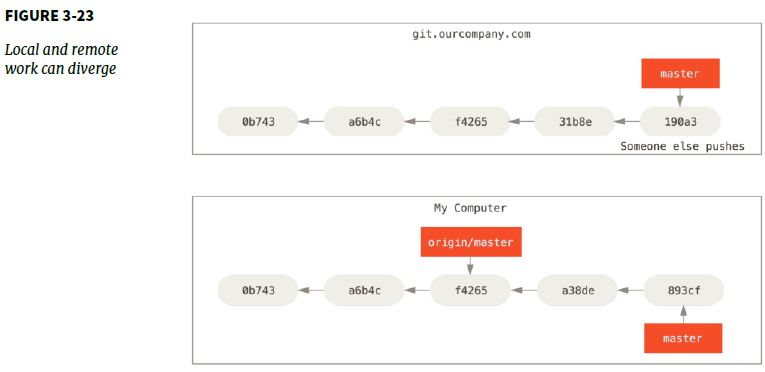
****

****

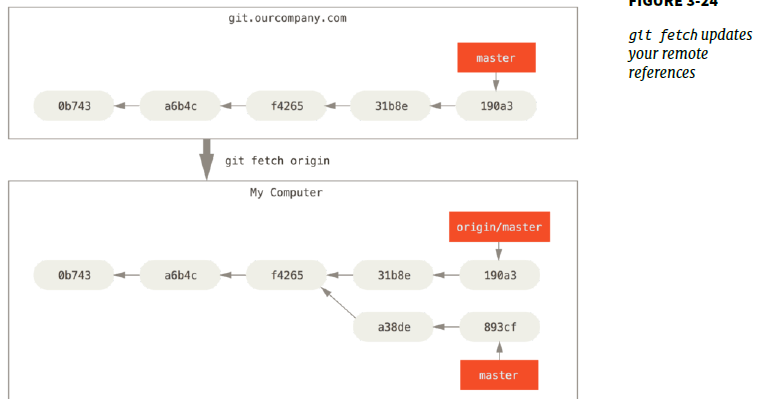
You can get a full list of remote references explicitly with **git ls-remote [remote], or git remote show [remote]**

****

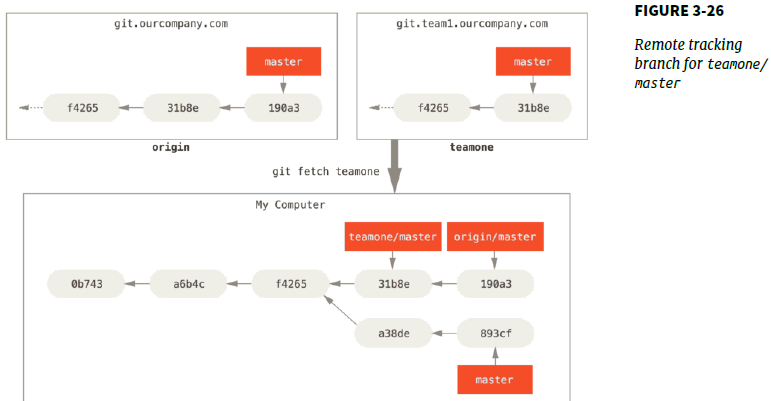
Just like the branch name “master” does not have any special meaning in Git, neither does “origin”. While “master” is the default name for a starting branch when you run git init which is the only reason it’s widely used, “origin” is the default name for a remote when you run git clone. If you run git clone -o booyah instead, then you will have booyah/ master as your default remote branch. If you do some work on your local master branch, and, in the meantime, someone else pushes to git.ourcompany.com and updates its master branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your origin/master pointer doesn’t move.

****

To synchronize your work, you run a git fetch origin command. This command looks up which server “origin” is (in this case, it’s git.ourcompany. com), fetches any data from it that you don’t yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position.

****

**In case you have two remotes than it will look like:**

****

If you have a branch named serverfix that you want to work on with others, you can push it up the same way you pushed your first branch. Run git push <remote> <branch>:

**$ git push origin serverfix**

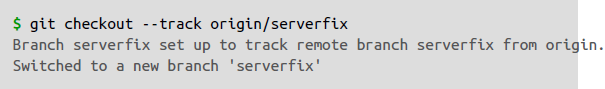
This is a bit of a shortcut. Git automatically expands the serverfix branchname out to refs/heads/serverfix:refs/heads/serverfix, which means, “Take my serverfix local branch and push it to update the remote’s serverfix branch.”

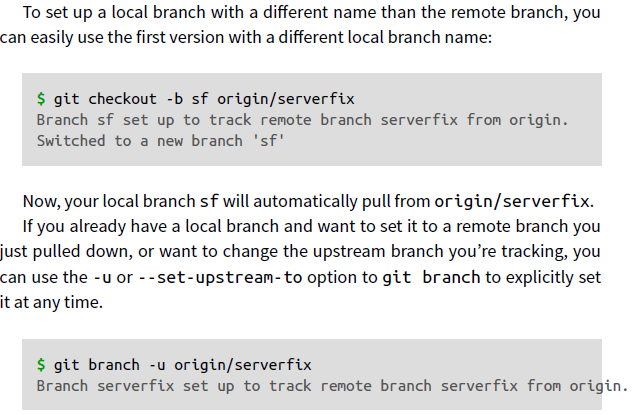
**git push origin serverfix: serverfix -- means same thing as above.**

If you didn’t want it to be called serverfix on the remote, you could instead run git push origin serverfix: awesomebranch to push your local serverfix branch to the awesomebranch branch on the remote project.

If you want your own serverfix branch that you can work on, you can base it off your remote-tracking branch:

**$ git checkout -b serverfix origin/serverfix**

****

****

If you want to see what tracking branches you have set up

**$ git branch –vv**

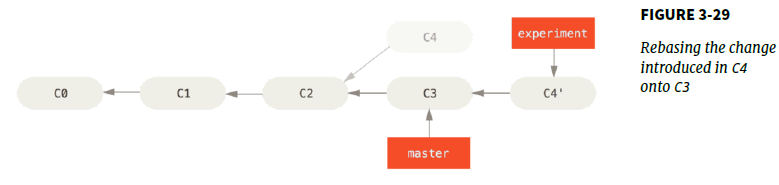
There is a command called git pull which is essentially a git fetch immediately followed by a git merge in most cases. If you have a tracking branch set up as demonstrated in the last section, either by explicitly setting it or by having it created for you by the clone or checkout commands, git pull will look up what server and branch your current branch is tracking, fetch from that server and then try to merge in that remote branch.

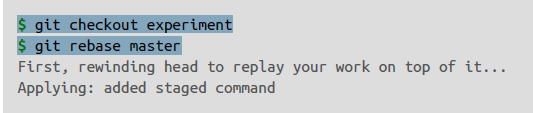
To delete your serverfix branch from the server, you run the following:

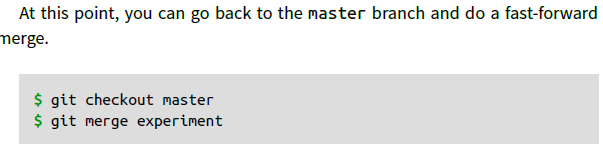
**$ git push origin --delete serverfix**

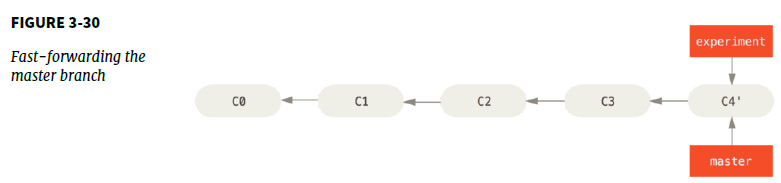
Basically all this does is remove the pointer from the server. The Git server will generally keep the data there for a while until a garbage collection runs, so if it was accidentally deleted, it’s often easy to recover.

You can take the patch of the change that was introduced in C4 and reapply it on top of C3. In Git, this is called *rebasing*. With the rebase command, you can take all the changes that were committed on one branch and replay them on another one. It works by going to the common ancestor of the two branches (the one you’re on and the one you’re rebasing onto), getting the diff introduced by each commit of the branch you’re on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

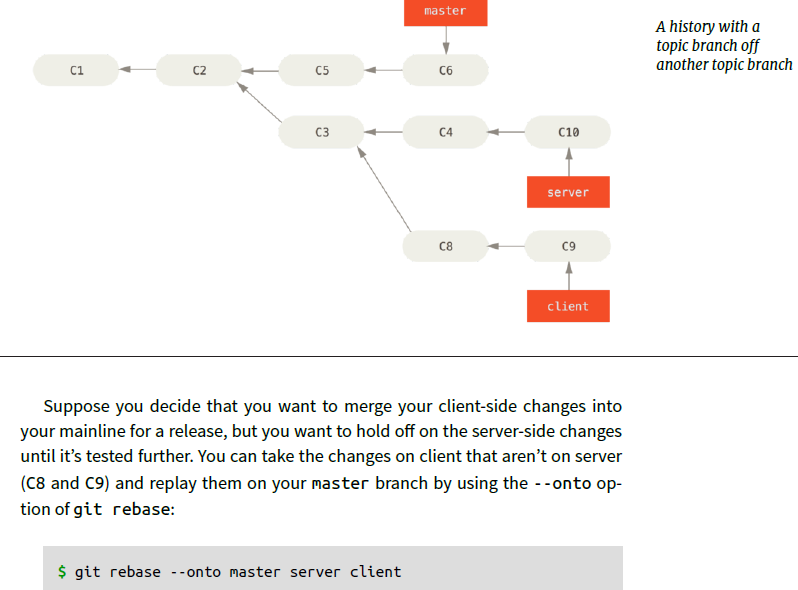
****

****

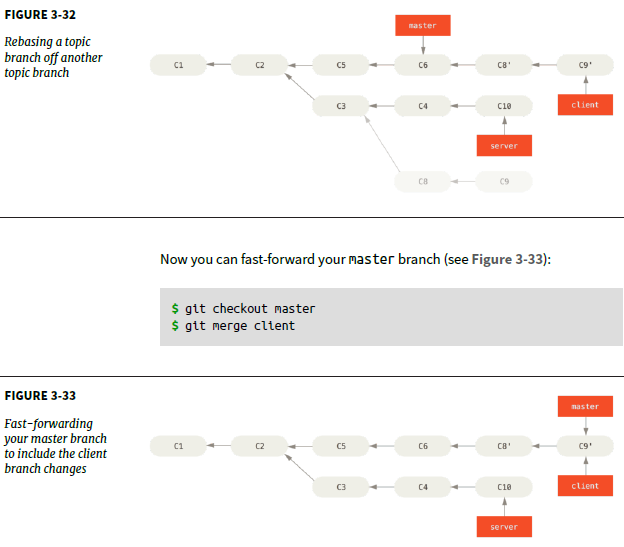
****

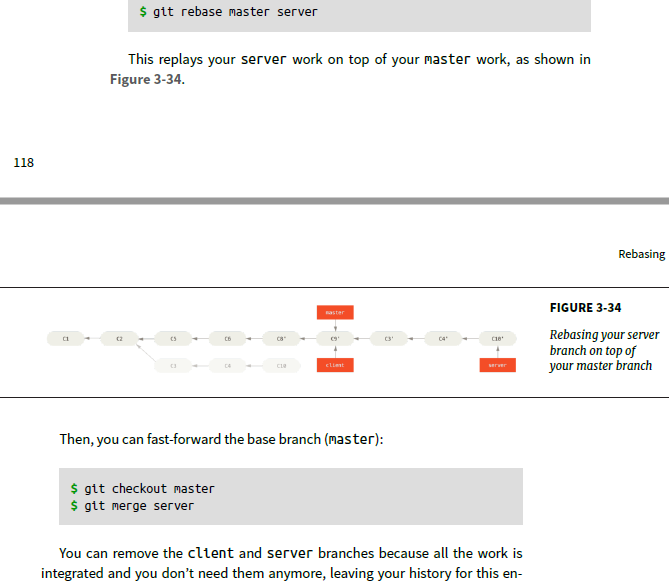
****

Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

****

This basically says, “Check out the client branch, figure out the patches from the common ancestor of the client and server branches, and then replay them onto master.” It’s a bit complex, but the result is pretty cool.

****

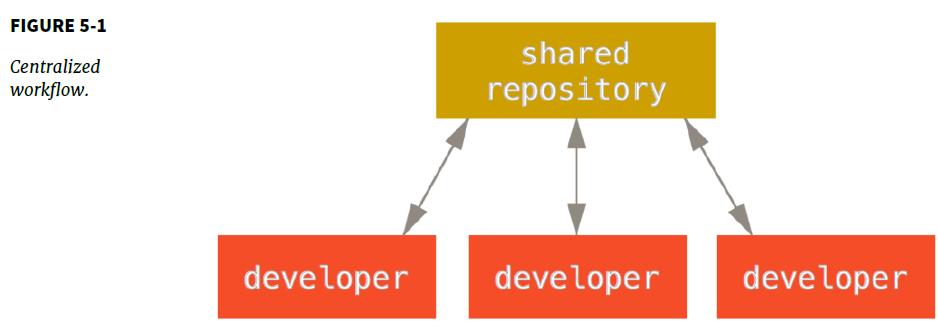
****

In general the way to get the best of both merge and rebase is to rebase local changes

You have made but haven’t shared yet before you push them in order to clean up your story, but never rebase anything you’ve pushed somewhere.

**Centralized Workflow**

In centralized systems, there is generally a single collaboration model–the centralized workflow. One central hub, or repository, can accept code, and everyone synchronizes their work to it. A number of developers are nodes – consumers of that hub – and synchronize to that one place.

****

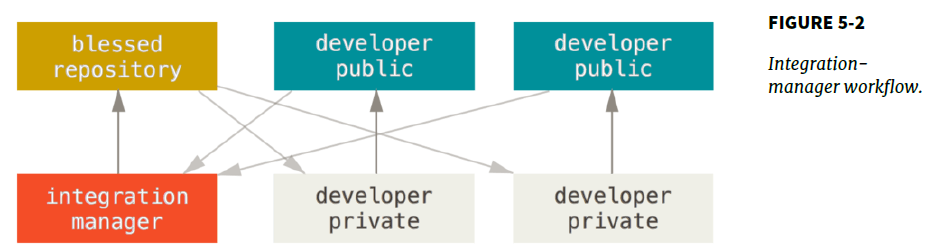
This means that if two developers clone from the hub and both make changes, the first developer to push their changes back up can do so with no problems. The second developer must merge in the first one’s work before pushing changes up, so as not to overwrite the first developer’s changes. This concept is as true in Git as it is in Subversion (or any CVCS), and this model works perfectly well in Git.

*Say John and Jessica both start working at the same time. John finishes his change and pushes it to the server. Then Jessica tries to push her changes, but the server rejects them. She is told that she’s trying to push non-fast-forward changes and that she won’t be able to do so until she fetches and merges. This workflow is attractive to a lot of people because*

*it’s a paradigm that many are familiar and comfortable with.*

**Integration-Manager Workflow**

Because Git allows you to have multiple remote repositories, it’s possible to have a workflow where each developer has write access to their own public repository and read access to everyone else’s. This scenario often includes a canonical repository that represents the “official” project.

****

The process works as follows (see **Figure 5-2**):

1. The project maintainer pushes to their public repository.

2. A contributor clones that repository and makes changes.

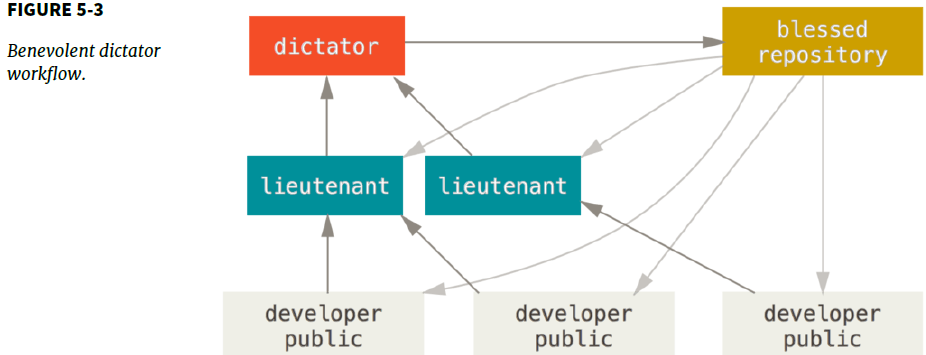
3. The contributor pushes to their own public copy.

4. The contributor sends the maintainer an email asking them to pull changes.

5. The maintainer adds the contributor’s repo as a remote and merges locally.

6. The maintainer pushes merged changes to the main repository.

**Dictator and Lieutenants Workflow**

****

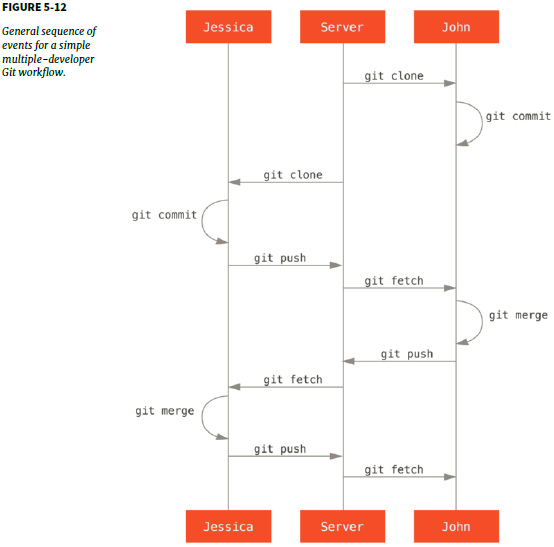
1. Regular developers work on their topic branch and rebase their work on top of master. The master branch is that of the dictator.

2. Lieutenants merge the developers’ topic branches into their master branch.

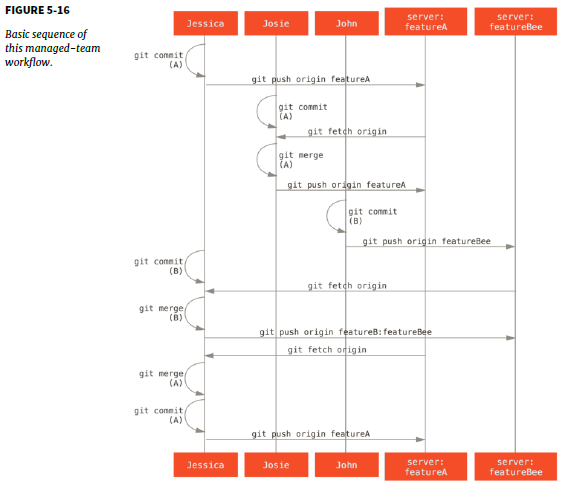
3. The dictator merges the lieutenants’ master branches into the dictator’s master branch.

4. The dictator pushes their master to the reference repository so the other developers can rebase on it.

Before you commit, run git diff --check, which identifies possible whitespace errors and lists them for you. If you run that command before committing, you can tell if you’re about to commit whitespace issues that may annoy other developers.



**$ git push -u origin featureB:featureBee ----** This is called a *refspec ----* the -u flag; this is short for --set-upstream, which configures the branches for easier pushing and pulling later.

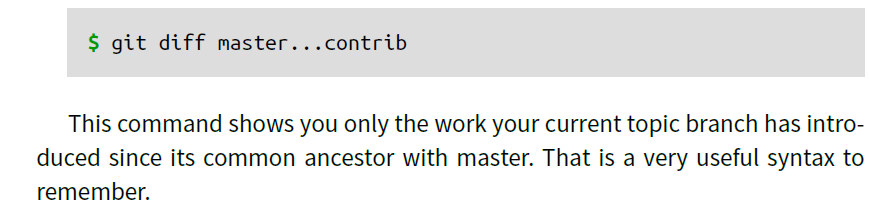
**

The request-pull command takes the base branch into which you want your topic branch pulled and the Git repository URL you want them to pull from, and outputs a summary of all the changes you’re asking to be pulled in.

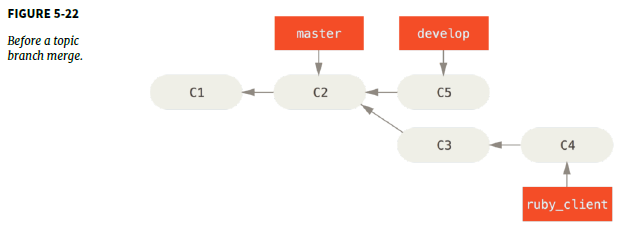
The --squash option takes all the work on the merged branch and squashes it into one change set producing the repository state as if a real merge happened, without actually making a merge commit. This means your future commit will have one parent only and allows you to introduce all the changes from another branch and then make more changes before recording the new commit.

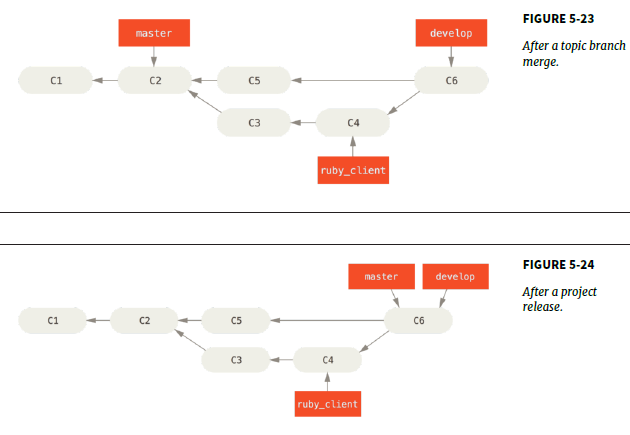
**COMMANDS:**

1. To go to particular directory in bash: cd<space>.\<directory name>
2. To check status: git status
3. To check difference: git diff
4. To put to staging: git add . (for all files) **or** git add pract.txt (for single file)
5. To give message while saving it as well: git commit -m "Story 182: Fix benchmarks for speed"
6. To do git commit with add and message togather: git commit -a -m 'added new benchmarks'
7. To remove file from your working directory and git: rm abc.txt

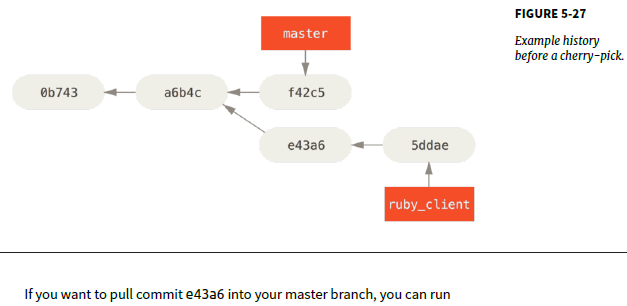


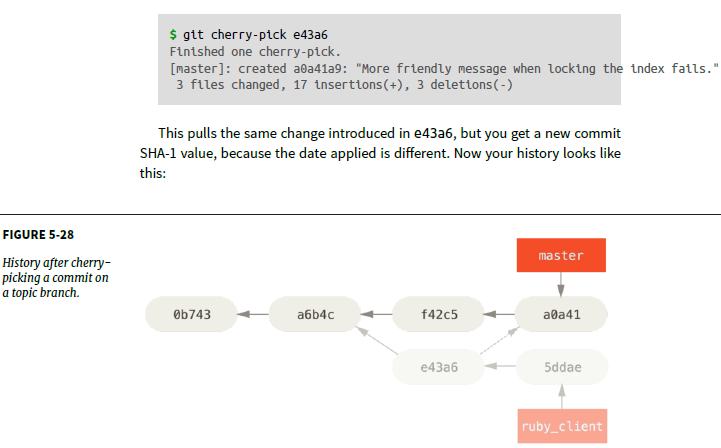
**Integrating Contributed Work**





Way to move introduced work from one branch to another is to cherry-pick it. A cherry-pick in Git is like a rebase for a single commit. It takes the patch that was introduced in a commit and tries to reapply it on the branch you’re currently on. This is useful if you have a number of commits on a topic branch and you want to integrate only one of them, or if you only have one commit on a topic branch and you’d prefer to cherry-pick it rather than run rebase.





**RERERE**

Rerere stands for “reuse recorded resolution” – it’s a way of shortcutting manual conflict resolution. When rerere is enabled, Git will keep a set of preand post-images from successful merges, and if it notices that there’s a conflict that looks exactly like one you’ve already fixed, it’ll just use the fix from last time, without bothering you with it. This feature comes in two parts: a configuration setting and a command. The configuration setting is rerere.enabled, and it’s handy enough to put in your global config:

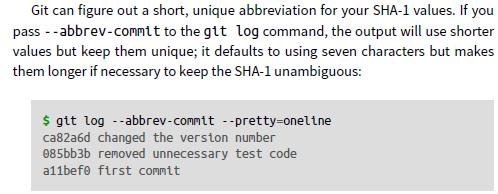
**$ git config --global rerere.enabled true**

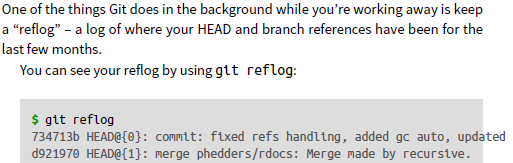
**$ git describe master --** Gitgives you the name of the nearest tag with the number of commits on top of that tag and a partial SHA-1 value of the commit you’re describing



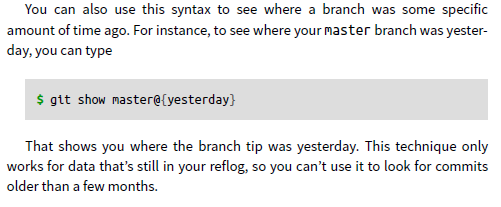
A nice way of quickly getting a sort of changelog of what has been added to your project since your last release or email is to use the git shortlog command. It summarizes all the commits in the range you give it; for example, the following gives you a summary of all the commits since your last release, if your last release was named v1.0.1:

**$ git shortlog --no-merges master --not v1.0.1**

****

****

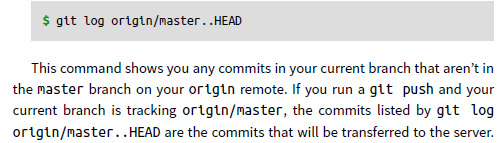
****

****

It’s important to note that the reflog information is strictly local – it’s a log of what you’ve done in your repository. The references won’t be the same on someone else’s copy of the repository; and right after you initially clone a repository, you’ll have an empty reflog, as no activity has occurred yet in your repository. Running git show HEAD@{2.months.ago} will work only if you cloned the project at least two months ago – if you cloned it five minutes ago, you’ll get no results.

**$ git log master..<branch-name> or $ git log <branch-name>..master**

**These commands are basically used to check which commits are not done from branch to master or master to branch**

****

**TRIPLE DOT**

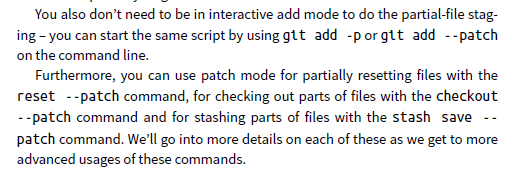
The last major range-selection syntax is the triple-dot syntax, which specifies all the commits that are reachable by either of two references but not by both of them.

**$ git log master...experiment**

If you run git add with the -i or --interactive option, Git goes into an interactive shell mode, displaying something like this: **$ git add –i**

**Staging Patches**

It’s also possible for Git to stage certain parts of files and not the rest. For example, if you make two changes to your simplegit.rb file and want to stage one of them and not the other, doing so is very easy in Git. From the interactive prompt, type 5 or p (for patch). Git will ask you which files you would like to partially stage; then, for each section of the selected files, it will display hunks of the file diff and ask if you would like to stage them, one by one

****

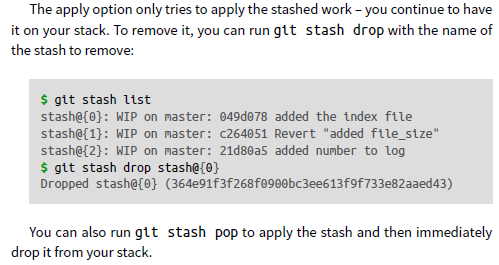
**Git stash command**--Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

**To see which stashes you’ve stored, you can use git stash list**

You can reapply the one you just stashed by using the command shown in the help output of the original stash command: git stash apply. If you want to apply one of the older stashes, you can specify it by naming it, like this: git stash apply stash@{2}. If you don’t specify a stash, Git assumes the most recent stash and tries to apply it.

The changes to your files were reapplied, but the file you staged before wasn’t restaged. To do that, you must run the git stash apply command with a --index option to tell the command to try to reapply the staged changes. If you had run that instead, you’d have gotten back to your original position

**$ git stash apply –index**

****