**Devops tool : GIT**

Git is an Open Source Distributed Version Control System. Now that’s a lot of words to define Git.

Let me break it down and explain the wording :

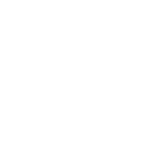
* Control System: This basically means that Git is a content tracker. So Git can be used to store content — it is mostly used to store code due to the other features it provides.
* Version Control System: The code which is stored in Git keeps changing as more code is added. Also, many developers can add code in parallel. So Version Control System helps in handling this by maintaining a history of what changes have happened. Also, Git provides features like branches and merges, which I will be covering later.
* Distributed Version Control System: Git has a remote repository which is stored in a server and a local repository which is stored in the computer of each developer. This means that the code is not just stored in a central server, but the full copy of the code is present in all the developers’ computers. Git is a Distributed Version Control System since the code is present in every developer’s computer. I will explain the concept of remote and local repositories later in this article.

Case :

Suppose you are working on a project.

And you wrote the source code. it was working fine , but you further changed the code as it was needed. After few minutes , you again changed the code , and ….again and….. again.

Now , you are messed up and frustrated as your current code is not fine and all you want is your initial code back. But , you can't as you have not saved the previous versions. Now, all you can do is \*cry in corner\*.



Wait … i have a solution.

First time , i saved my code as source\_code.cpp

then , i saved new version as final\_source\_code.cpp

and then again as final\_source\_code1.cpp

and then final\_source\_code2.cpp

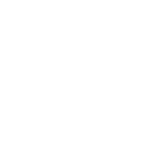
and then ………….

…………..final\_source\_code50.cpp

I have saved all versions of my code.but , you can see this solution is a new problem itself.

Now , suppose a different scenario. you are working a team project. and your team members are working from remote locations. How will you all work on same source code in real time ?

Big problem ? isn’t ?



To solve all these problems , we use version control system (VCS). it basically manages the changes to code or files. it stores all the changes and the revisions can be compared , restored or merged.

Git is a popular, free and open source version control system.

Repository is a data structure used by VCS to store metadata for set of files and/or directories. it stores the set of file as well as history of changes made to those file.

Commands :

| Git task | Notes | Git commands |
| --- | --- | --- |
| [Tell Git who you are](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) | Configure the author name and email address to be used with your commits.  Note that Git [strips some characters](http://stackoverflow.com/questions/26159274/is-it-possible-to-have-a-trailing-period-in-user-name-in-git/26219423#26219423) (for example trailing periods) from user.name. | git config --global user.name "testuser"  git config --global user.email “test@example.com” |
| [Create a new local repository](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) |  | git init |
| [Check out a repository](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) | Create a working copy of a local repository: | git clone /path/to/repository |
| For a remote server, use: | git clone username@host:/path/to/repository |
| [Add files](https://www.atlassian.com/git/tutorials/saving-changes#git-add) | Add one or more files to staging (index): | git add <filename>  git add \* |
| [Commit](https://www.atlassian.com/git/tutorials/saving-changes#git-commit) | Commit changes to head (but not yet to the remote repository): | git commit -m "Commit message" |
| Commit any files you've added with git add, and also commit any files you've changed since then: | git commit -a |
| [Push](https://www.atlassian.com/git/tutorials/syncing#git-push) | Send changes to the master branch of your remote repository: | git push origin master |
| [Status](https://www.atlassian.com/git/tutorials/inspecting-a-repository#git-status) | List the files you've changed and those you still need to add or commit: | git status |
| [Connect to a remote repository](https://www.atlassian.com/git/tutorials/syncing#git-remote) | If you haven't connected your local repository to a remote server, add the server to be able to push to it: | git remote add origin <server> |
| List all currently configured remote repositories: | git remote -v |
| [Branches](https://www.atlassian.com/git/tutorials/using-branches) | Create a new branch and switch to it: | git checkout -b <branchname> |
| Switch from one branch to another: | git checkout <branchname> |
| List all the branches in your repo, and also tell you what branch you're currently in: | git branch |
| Delete the feature branch: | git branch -d <branchname> |
| Push the branch to your remote repository, so others can use it: | git push origin <branchname> |
| Push all branches to your remote repository: | git push --all origin |
| Delete a branch on your remote repository: | git push origin :<branchname> |
| [Update from the remote repository](https://www.atlassian.com/git/tutorials/syncing) | Fetch and merge changes on the remote server to your working directory: | git pull |
| To merge a different branch into your active branch: | git merge <branchname> |
| Tags | You can use tagging to mark a significant changeset, such as a release: | git tag 1.0.0 <commitID> |
| CommitId is the leading characters of the changeset ID, up to 10, but must be unique. Get the ID using: | git log |
| Push all tags to remote repository: | git push --tags origin |
| [Undo local changes](https://www.atlassian.com/git/tutorials/undoing-changes) | If you mess up, you can replace the changes in your working tree with the last content in head:  Changes already added to the index, as well as new files, will be kept. | git checkout -- <filename> |

git stash - How to Save Your Changes Temporarily

There are lots of situations where a *clean working copy* is recommended or even required: when merging branches, when pulling from a remote, or simply when checking out a different branch.

The "git stash" command can help you to (temporarily but safely) store your uncommitted local changes - and leave you with a clean working copy.

git stash: a Clipboard for Your Changes

Let's say you currently have a couple of local modifications:

$ git status

modified: index.php

modified: css/styles.css

If you have to switch context - e.g. because you need to work on an urgent bug - you need to get these changes out of the way. You shouldn't just commit them, of course, because it's unfinished work.

This is where "git stash" comes in handy:

$ git stash

Saved working directory and index state WIP on master:

2dfe283 Implement the new login box

HEAD is now at 2dfe283 Implement the new login box

Your working copy is now clean: all uncommitted local changes have been saved on this kind of "clipboard" that Git's Stash represents. You're ready to start your new task (for example by pulling changes from remote or simply switching branches).

Continuing Where You Left Off

As already mentioned, Git's Stash is meant as a temporary storage. When you're ready to continue where you left off, you can restore the saved state easily:

$ git stash pop

The "pop" flag will reapply the *last saved* state and, at the same time, delete its representation on the Stash (in other words: it does the clean-up for you).

### **First, why would I Need to Rebase Something?**

Let's say you're a junior dev starting at a cupcake store called Cupid's Cupcakes. It does lots of online selling, and has many experienced devs constantly improving it. You're brought in to mostly work on the front-end.

Your first assignment is updating a card component. When people look for cupcakes to buy, each is in one of these cards. So you go to the repo, pull the most recent version of the master branch, create a new branch from that one, and get to work!

A few commits later, you're all set. The card looks nicer, all the tests pass, and you've even improved the mobile layout. All that's left is to merge your feature branch back into master branch so it goes live!

**But wait a moment!**

Unsurprisingly, other people were working on the site while you were making this card component.

* One developer changed the navigation
* One adjusted the database fields to remove unneeded info
* Another added extra info about each cupcake
* Someone else secretly embezzled money through the store's bank records

All these changes make you worry. What if someone merged a change that affects or overlaps with the ones you made? It could lead to bugs in the cupcake website! If you look at the different changes made, one does! (Another change should be reported to the police, but that's actually less important). Is there a safe way to merge your changes without risking any conflicts, and missing out on all the other changes made?

**Situations like these are a big example of when you'd want to rebase.**

### **What are the details of Rebasing?**

Let's say when you created your branch off of the master branch, the master branch was on commit #1. Every commit in your branch was layered on top of commit #1. When you're ready to merge your branch, you discover other developers made changes and the most recent commit is commit #5.

Rebasing is **taking all your branch's commits and adding them on top of commit #5 instead of commit #1**. If you consider commit #1 as the "base" of your branch, you're changing that base to the most recent one, commit #5. Hence why it's called rebasing!



### **Okay, so HOW do I Rebase something?**

So you've got this great card component for Cupid's Cupcakes. Now that you know what a rebase *is*, let's look at the *how* in more detail.

First, **make sure you have the most up-to-date version of the branch you're rebasing on.** Let's keep assuming it's the master branch in this example. Run git checkout master to, y'know, check it out, and then run git pull to get the most recent version. Then checkout your branch again - here's it'd be with git checkout updated-card or something similar.

A straightforward rebase has a pretty simple command structure: git rebase <branch>. branch is the one you're rebasing *off of.* So here you'd run git rebase master. Assuming there's no conflicts, that's all the rebase needs!

The rebase itself technically removes your old commits and makes new commits identical to them, rewriting the repo's commit history. That means pushing the rebase to the remote repo will need some extra juice. Using git push --force will do the trick fine, but a safer option is git push --force-with-lease. The latter will alert you of any upstream changes you hadn't noticed and prevent the push. This way you avoid overwriting anyone else's work, so it's the safer option.

With all that, your rebase is now complete! However, rebases won't always go so smoothly...

### **How do I Handle Rebase Conflicts?**

Remember how we worried our new card would conflict with someone else's changes? Turns out, one does! One developer added extra info onto the new cupcake card, such as calorie count or how many elves it takes to make it at night. The updated markups from both sets of change are in the same lines - this means the rebase can't happen automatically. Git won't know which parts of the changes to keep and which to remove. It must be resolved!

Thankfully, git makes this very easy. During the rebase, git adds each commit onto the new base one by one. If it reaches a commit with a conflict, **it'll pause the rebase and resume once it's fixed.**

If you've dealt with [merge conflicts before](https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/), rebase conflicts are handled essentially the same way. Running git status will tell you where the conflicts are, and the two conflicting sections of code will be next to each other so you can decide how to fix them.

Once everything is fixed, add and commit the changes like you would a normal merge conflict. Then run git rebase --continue so git can rebase the rest of your commits. It'll pause for any more conflicts, and once they're set you just need to push --force-with-lease.

There's two lesser-used options you could also use. One is git rebase --abort, which would bring you back to before you started the rebase. It's useful for unexpected conflicts that you can't rush a decision for. Another is git rebase --skip, which skips over the commit causing the conflict altogether. Unless it's an unneeded commit and you're feeling lazy, you likely won't use it much.

Git has taken the programming community by storm. Hundreds of thousands of organizations and developers is starting  to use Git as their Version Control System (VCS).  But you might wonder what makes Git so special?

In this post, I'll going to dig into one of the mysterious aspect of Git --- The 3-Tree Architecture.

To get started with, lets  first take a look at how the typical VCS works. Usually, a VCS works by having two places to store things:

1. Working Copy
2. Repository

**Working copy** is the place where you make your changes. Whenever you edit something, it is saved in working copy and it is a physically stored in a disk.

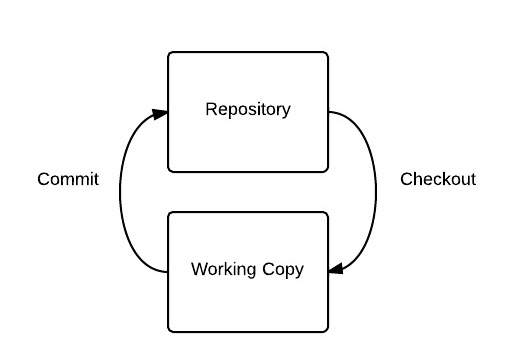
**Repository** is the place where all the version of the files or  commits,  logs etc is stored. It is also saved in a disk and has its own set of files.

You cannot however change or get the files  in a repository directly, in able to retrieve a specific file from there, you have to *checkout*

**Checking-out** is the process of getting files from repository to your working copy.  This is because you can only edit files when it is on your working copy. When you are done editing the file, you will save it back to the repository by *commiting* it.

**Committing** is the process of putting back the files from working copy to repository.

3 tree archi vs 2 tree arch

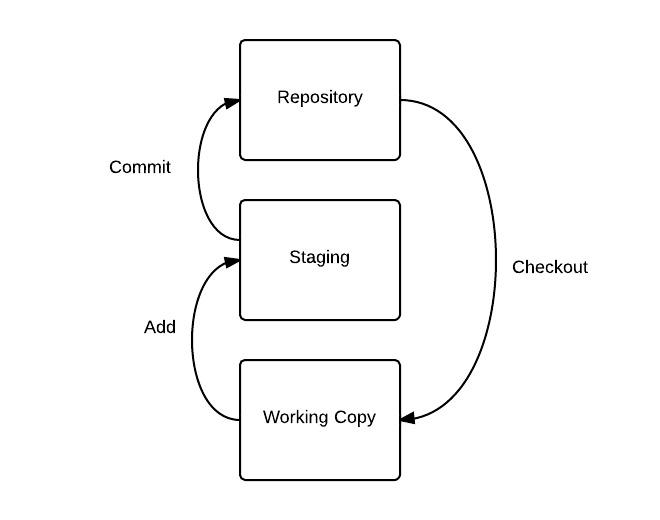


In this process, Working Copy and  Repository is saved in the disk as series of folders and files like a *Tree,*since files and directories resembles a tree wherein folder represents a branch of a tree and files represents the leaf. Hence, this architecture is called **2 Tree Architecture**. Because you have two tree in there -- Working Copy and Repository. The famous VCS with this kind of architecture is Subversion or SVN.

Now, that you know what a 2 Tree Architecture looks like, interesting to say Git has different one, it is instead powered by 3 Trees!

Why three you might ask?

Well, interestingly, Git has also the Working Copy and Repository as well, but it had added an extra tree in between:



As you can see above, there is a new tree called Staging. What this is for?

This is one of the fundamental difference of Git that sets it apart from other VCS, this Staging tree (usually termed as Staging area) is a place where you prepare all the things that you are going to commit.

In Git, you don't move things directly from your working copy to the repository, you have to stage them first, one of the main benefits of this is, lets say:

*You did changes on your 10 files, 2 of the files is something related to fixing an alignment issue in a webpage, while the other 8 changed files is related to database connection..*

HEAD