## **SVN vs Git: What is the Difference Between Git and SVN?**

While Git and SVN are both enterprise version control systems (VCS) that help with workflow and project management in coding, they do have their differences. Thedifference between Git and SVN version control systems is that Git is a distributed version control system, whereas SVN is a centralized version control system. Git uses multiple repositories including a centralized repository and server, as well as some local repositories. SVN does not have a centralized repository or server.

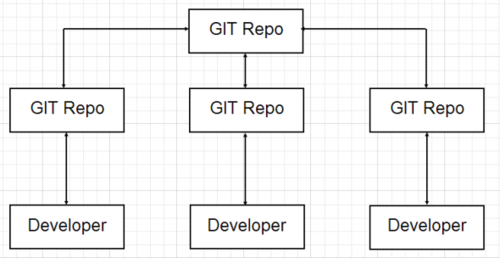
### **What is GIT?**

Git is a distributed version control system – which just means that when you do a git clone (+url of your repository) what you are actually getting is a complete copy of your entire history of that project. This means all your commits! Woot!

### **What are the advantages of Git?**

Git has a staging area. This just means that if you made 100 new changes to your code, you can break these 100 changes into 10 or 20 or more commits each with their own comments and their own detailed explanation of what just happened! Not only can you stage your commits out to logically display what changes were made, but you can also do patch staging that ask you if you want. You would use patch staging if you and a co-worker are both working on the same file and you only want to commit a particular function that you’ve worked on. You do a Git patch using “git add -p”

Git Work Flow



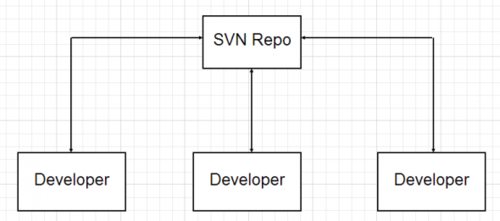
### **What is SVN?**

Subversion (SVN) may be one of the most well known centralized version control systems. In Subversion or SVN, you are checking out a single version of the repository.With SVN, your data is stored on a central server. Having the entire history on yourlocal repository just means that even when you are not connected to the Internet, you can still do commits, diffs, logs, branches, merges, file annotations, etc.

### **What are the advantages of SVN?**

SVN has one central repository – which makes it easier for managers to have more of a top down approach to control, security, permissions, mirrors and dumps. Additionally, many say SVN is easier to use than Git. For example, it is easier to create a new feature. With Git, it takes an extra step to create a new feature. Others say that the way SVN is set up results in greater trunk stability, and having everything on a central server feels more controlled and secure for some.

SVN Work Flow



### **SVN vs Git**

SVN allows you to check out sub-trees (or branches) only whereas Git requires you to check out the entire repository as a unit. This is because there is a .svn in each one of your folders while git only has one .git at the top level parent directory.

### **Is Git better than SVN?**

Git and SVN are each viable workflow and version control systems, but for different reasons. Git may have more difficulty compressing and storing binary files, while SVN doesn’t as much. That said, many claim Git is better than SVN because it works well even for developers who aren’t always connected to the master repository, as it is available offline. Branching and merging support are also thought to be superior with Git. When it comes to disk space storage, it’s pretty close to equal between both SVN and Git repositories. Git is also a bit newer than SVN.

### **SVN vs Git: It’s up to you**

I’m sure there are a lot more ways that GIT and SVN differ. If you take time to look for an example or two of how one or the other is best suited to your enterprise or workflow, you may have an easier time talking to management about getting the right VCS for your needs.

Version Control System (VCS) is a software that helps software developers to work together and maintain a complete history of their work.

Listed below are the functions of a VCS −

Allows developers to work simultaneously.

Does not allow overwriting each other’s changes.

Maintains a history of every version.

Following are the types of VCS −

Centralized version control system (CVCS).

Distributed/Decentralized version control system (DVCS).

In this chapter, we will concentrate only on distributed version control system and especially on Git. Git falls under distributed version control system.

### Working Directory and Staging Area or Index

The working directory is the place where files are checked out. In other CVCS, developers generally make modifications and commit their changes directly to the repository. But Git uses a different strategy. Git doesn’t track each and every modified file. Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.

Let us see the basic workflow of Git.

**Step 1** − You modify a file from the working directory.

**Step 2** − You add these files to the staging area.

**Step 3** − You perform commit operation that moves the files from the staging area. After push operation, it stores the changes permanently to the Git repository.



Suppose you modified two files, namely “sort.c” and “search.c” and you want two different commits for each operation. You can add one file in the staging area and do commit. After the first commit, repeat the same procedure for another file.

git config --global user.name "your name"

git config --global user.email "MY\_NAME@example.com"

# First commit

[bash]$ git add sort.c

# adds file to the staging area

[bash]$ git commit –m “Added sort operation”

# Second commit

[bash]$ git add search.c

# adds file to the staging area

[bash]$ git commit –m “Added search operation”

**To create a branch locally:-**

You can create a branch locally as long as you have a cloned version of the repo.

From your terminal window, list the branches on your repository.

$ git branch

\* master

This output indicates there is a single branch, the master and the asterisk indicates it is currently active.

Create a new feature branch in the repository

$ git branch <feature\_branch>

Switch to the feature branch to work on it.

$ git checkout <feature\_branch>

You can list the branches again with the git branch command.

Commit the change to the feature branch:

$ git add .

$ git commit -m "adding a change from the feature branch"

Switch back to the master branch.

**Merge branch in local**

# Start a new feature

git checkout -b new-feature master

# Edit some files

git add <file>

git commit -m "Start a feature"

# Edit some files

git add <file>

git commit -m "Finish a feature"

# Merge in the new-feature branch

git checkout master

git merge new-feature

git branch -d new-feature

**Rebase Branch in Local**

If <branch> is specified, git rebase will perform an automatic git checkout <branch> before doing anything else. Otherwise it remains on the current branch.

All changes made by commits in the current branch but that are not in <upstream> are saved to a temporary area. This is the same set of commits that would be shown by git log <upstream>..HEAD; or by git log 'fork\_point'..HEAD, if --fork-point is active (see the description on --fork-point below); or by git log HEAD, if the --root option is specified.

The current branch is reset to <upstream>, or <newbase> if the --onto option was supplied. This has the exact same effect as git reset --hard <upstream> (or <newbase>). ORIG\_HEAD is set to point at the tip of the branch before the reset.

The commits that were previously saved into the temporary area are then reapplied to the current branch, one by one, in order. Note that any commits in HEAD which introduce the same textual changes as a commit in HEAD..<upstream> are omitted (i.e., a patch already accepted upstream with a different commit message or timestamp will be skipped).

It is possible that a merge failure will prevent this process from being completely automatic. You will have to resolve any such merge failure and run git rebase --continue. Another option is to bypass the commit that caused the merge failure with git rebase --skip. To check out the original <branch> and remove the .git/rebase-apply working files, use the command git rebase --abort instead.

Assume the following history exists and the current branch is "topic":

A---B---C topic

/

D---E---F---G master

From this point, the result of either of the following commands:

**git rebase master**

**git rebase master topic**

would be:

A'--B'--C' topic

/

D---E---F---G master

NOTE: The latter form is just a short-hand of git checkout topic followed by git rebase master. When rebase exits topic will remain the checked-out branch.

**Diff in Local**

The main objective of version controlling is to enable you to work with different versions of files. Git provides a command diff to let you to compare different versions of your files.

The most common scenario to use diff is to see what changes you made after your last commit. Let’s see how to do it.

I opened the helloworld project from my last example with a clean working directory. i.e. I have already committed all my code changes. So, a git status will give an output like this:

C:\vraa\projects\helloworld> git status

# On branch master

nothing to commit (working directory clean)

Let’s make a change in the helloworld.txt file now and compare this file with previously committed version.

C:\vraa\projects\helloworld> edit .\helloworld.txt

C:\vraa\projects\helloworld> git diff HEAD .\helloworld.txt

diff --git a/helloworld.txt b/helloworld.txt

index e4f37c4..557db03 100644

--- a/helloworld.txt

+++ b/helloworld.txt

@@ -1 +1 @@

-Hello India

+Hello World

There it is. Git shows the exact change I made in the file. But, if you look at the diff command, you might wonder what HEAD is doing there! Well, it is there for a purpose.

If you can recall, Git has an index between local repository and your working directory. So most of Git commands can either refer to index or the local repo. When you say HEAD in your Git command, it refers the local repo.

git diff HEAD [filename] // compare the working directory with local repository.

git diff [filename] // compare the working directory with index.

git diff --cached [filename] // compare the index with local repository.

You can also compare files between two different commits. Every commit in Git has a commit id which you can get when you give git log. Then you can use the commit id if diff command like this.

git diff 7eb2..e03 812...a3f35

You can compare not just a single file, but all your changes at once. If you made changes in many files, just don’t mention any file name in the diff command which will diff all the changed files.

git diff // compares working directory with index, i.e. shows the changes that are not staged yet.

git diff HEAD // compares working directory with local repository. shows the list of changes after your last commit.

git diff --cached // compares index with local repository. shows the diff between your last commit and changes to be committed next.

Git stash

The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy. For example:

$ git status

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

$ git stash

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

$ git status

On branch master

nothing to commit, working tree clean

**Re-applying your stashed changes**

You can reapply previously stashed changes with git stash pop:

$ git status

On branch master

nothing to commit, working tree clean

$ git stash pop

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)

**Stashing untracked or ignored files**

By default, running git stash will stash:

changes that have been added to your index (staged changes)

changes made to files that are currently tracked by Git (unstaged changes)

But it will not stash:

new files in your working copy that have not yet been staged

files that have been ignored

So if we add a third file to our example above, but don't stage it (i.e. we don't run git add), git stash won't stash it.

$ script.js

$ git status

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Untracked files:

script.js

$ git stash

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

$ git status

On branch master

Untracked files:

script.js

Adding the -u option (or --include-untracked) tells git stash to also stash your untracked files:

$ git status

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Untracked files:

script.js

$ git stash -u

Saved working directory and index state WIP on master: 5002d47 our new homepage

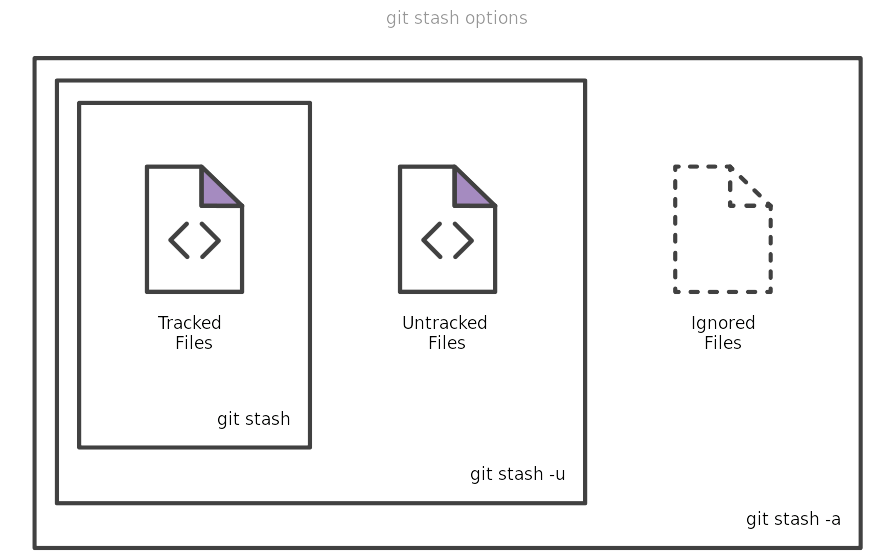
HEAD is now at 5002d47 our new homepage

$ git status

On branch master

nothing to commit, working tree clean

You can include changes to ignored files as well by passing the -a option (or --all) when running git stash.



Git revert

The git revert command can be considered an 'undo' type command, however, it is not a traditional undo operation. Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.

**How it works**

The git revert command is used for undoing changes to a repository's commit history. Other 'undo' commands like, git checkout and git reset, move the HEAD and branch ref pointers to a specified commit. Git revert also takes a specified commit, however, git revert does not move ref pointers to this commit. A revert operation will take the specified commit, inverse the changes from that commit, and create a new "revert commit". The ref pointers are then updated to point at the new revert commit making it the tip of the branch.

To demonstrate let’s create an example repo using the command line examples below:

$ mkdir git\_revert\_test

$ cd git\_revert\_test/

$ git init .

Initialized empty Git repository in /git\_revert\_test/.git/

$ touch demo\_file

$ git add demo\_file

$ git commit -am"initial commit"

[master (root-commit) 299b15f] initial commit

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 demo\_file

$ echo "initial content" >> demo\_file

$ git commit -am"add new content to demo file"

[master 3602d88] add new content to demo file

n 1 file changed, 1 insertion(+)

$ echo "prepended line content" >> demo\_file

$ git commit -am"prepend content to demo file"

[master 86bb32e] prepend content to demo file

1 file changed, 1 insertion(+)

$ git log --oneline

86bb32e prepend content to demo file

3602d88 add new content to demo file

299b15f initial commit

Here we have initialized a repo in a newly created directory named git\_revert\_test. We have made 3 commits to the repo in which we have added a file demo\_file and modified its content twice. At the end of the repo setup procedure, we invoke git log to display the commit history, showing a total of 3 commits. With the repo in this state, we are ready to initiate a git revert.

$ git revert HEAD

[master b9cd081] Revert "prepend content to demo file"

1 file changed, 1 deletion(-)

Git revert expects a commit ref was passed in and will not execute without one. Here we have passed in the HEAD ref. This will revert the latest commit. This is the same behavior as if we reverted to commit 3602d8815dbfa78cd37cd4d189552764b5e96c58. Similar to a merge, a revert will create a new commit which will open up the configured system editor prompting for a new commit message. Once a commit message has been entered and saved Git will resume operation. We can now examine the state of the repo using git log and see that there is a new commit added to the previous log:

$ git log --oneline

1061e79 Revert "prepend content to demo file"

86bb32e prepend content to demo file

3602d88 add new content to demo file

299b15f initial commit

Note that the 3rd commit is still in the project history after the revert. Instead of deleting it, git revert added a new commit to undo its changes. As a result, the 2nd and 4th commits represent the exact same code base and the 3rd commit is still in our history just in case we want to go back to it down the road.

**Git clone**

git clone is primarily used to point to an existing repo and make a clone or copy of that repo at in a new directory, at another location. The original repository can be located on the local filesystem or on remote machine accessible supported protocols. The git clone command copies an existing Git repository. This is sort of like SVN checkout, except the “working copy” is a full-fledged Git repository—it has its own history, manages its own files, and is a completely isolated environment from the original repository.

As a convenience, cloning automatically creates a remote connection called "origin" pointing back to the original repository. This makes it very easy to interact with a central repository. This automatic connection is established by creating Git refs to the remote branch heads under refs/remotes/origin and by initializing remote.origin.url and remote.origin.fetch configuration variables.

An example demonstrating using git clone can be found on the setting up a repository guide. The example below demonstrates how to obtain a local copy of a central repository stored on a server accessible at example.com using the SSH username john:

git clone ssh://john@example.com/path/to/my-project.git

cd my-project

# Start working on the project

The first command initializes a new Git repository in the my-project folder on your local machine and populates it with the contents of the central repository. Then, you can cd into the project and start editing files, committing snapshots, and interacting with other repositories. Also note that the .git extension is omitted from the cloned repository. This reflects the non-bare status of the local copy.

Cloning to a specific folder

git clone <repo> <directory>

Clone the repository located at <repo> into the folder called ~<directory>! on the local machine.

Cloning a specific tag

git clone --branch <tag> <repo>

Clone the repository located at <repo> and only clone the ref for <tag>.

Shallow clone

git clone -depth=1 <repo>

Clone the repository located at <repo> and only clone the

history of commits specified by the option depth=1. In this example a clone of <repo> is made and only the most recent commit is included in the new cloned Repo. Shallow cloning is most useful when working with repos that have an extensive commit history. An extensive commit history may cause scaling problems such as disk space usage limits and long wait times when cloning. A Shallow clone can help alleviate these scaling issues.

Configuration options

git clone -branch

The -branch argument lets you specify a specific a branch to clone instead of the branch the remote HEAD is pointing to, usually the master branch. In addition you can pass a tag instead of branch for the same effect.

git clone -branch new\_feature git://remoterepository.git

This above example would clone only the new\_feature branch from the remote Git repository. This is purely a convince utility to save you time from downloading the HEAD ref of the repository and then having to additionally fetch the ref you need.

**git pull**

The "pull" command is used to download and integrate remote changes.

The target (which branch the data should be integrated into) is always the currently checked out HEAD branch. By default, pull uses a merge operation, but it can also be configured to use rebase instead.

# git pull remote branch

eg- git pull origin master

**Git push**

The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to git fetch, but whereas fetching imports commits to local branches, pushing exports commits to remote branches. Remote branches are configured using the git remote command. Pushing has the potential to overwrite changes, caution should be taken when pushing. These issues are discussed below.

**Git push usage**

git push <remote> <branch>

Push the specified branch to <remote>, along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won’t let you push when it results in a non-fast-forward merge in the destination repository.

Git fetch

The git fetch command downloads commits, files, and refs from a remote repository into your local repo. Fetching is what you do when you want to see what everybody else has been working on. It’s similar to svn update in that it lets you see how the central history has progressed, but it doesn’t force you to actually merge the changes into your repository. Git isolates fetched content as a from existing local content, it has absolutely no effect on your local development work. Fetched content has to be explicitly checked out using the git checkout command. This makes fetching a safe way to review commits before integrating them with your local repository.

**Gitignore:**

GitIgnore file is a file that specifying the files or folders that we want to ignore. There are several ways to specifying those

The first one is specifying by the specific filename. Here is an example, let’s say we want to ignore a file called readme.txt, then we just need to write readme.txt in the .gitignore file.

The second one we can also write the name of the extension. For example, we are going to ignore all .txt files, then write \*.txt.

There is also a method to ignore a whole folder. Let’s say we want to ignore folder named test. Then we can just write test/ in the file.

There are other ways to define the ignore files. But, as a starter, you will probably only use those 3 ways.

Hands On

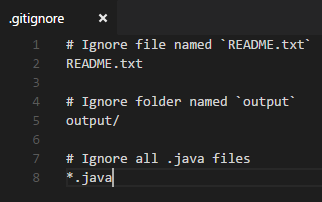
Let’s try to create the .gitignore file. I’m going to continue my repository. Let’s define our rules of the files that need to be keep. It is just an example.

1.For readme in Github, we will only need a README.md file. Let’s ignore if there is an README.txt file.

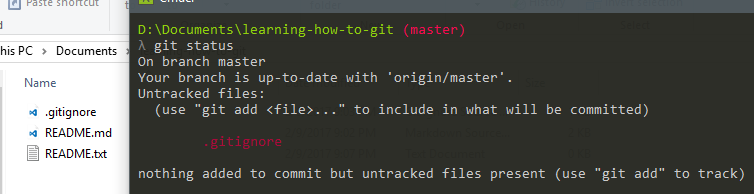
2.We also don’t need a folder called output.

3.Because there will be no Java files added to the repository, let’s ignore all of Java files (.java).

You can check the rules based on the previous section to convert those rule into a .gitignore file. You might get a similar file like this.



Save those file as .gitignore file and put it into the root folder of the Git project. Now, if you try to add those files to the Git repository, it will be ignored. Because we already define the rule for files that going to be ignored by the Git.



Notice that I try to add README.txt, but it is not listed when trying to check with `git status`.

One thing to note, if the files or folders already added to the Git project before we add it to the .gitignore file, then those files or folders will still available in the Git project. Because we try to ignore those files or folders after we already add those to the Git tracking system.

**What is the difference between ‘git pull’ and ‘git fetch’?**

In the simplest terms, git pull does a git fetch followed by a git merge.

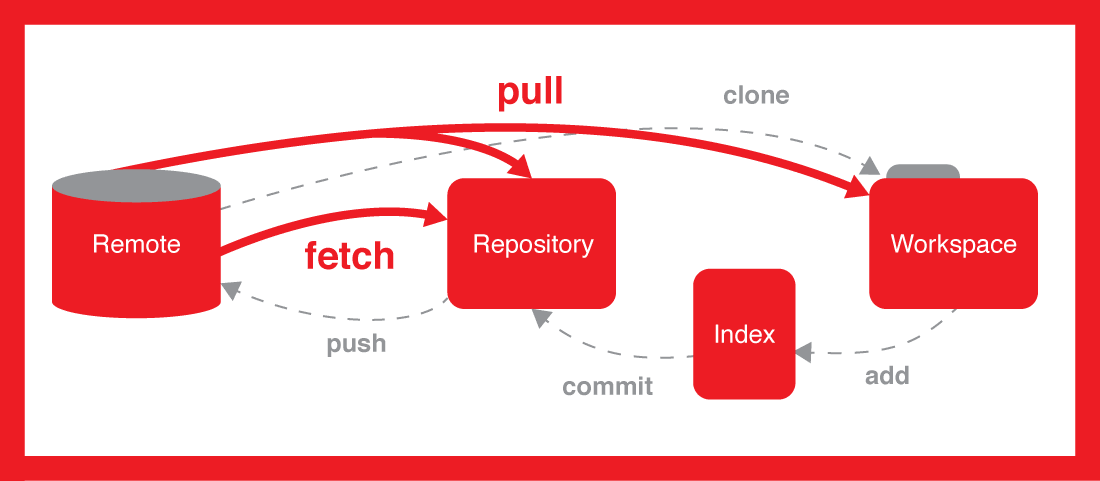
You can do a git fetch at any time to update your remote-tracking branches under refs/remotes/<remote>/.

This operation never changes any of your own local branches under refs/heads, and is safe to do without changing your working copy.

I have even heard of people running git fetch periodically in a cron job in the background (although I wouldn’t recommend doing this).

A git pull is what you would do to bring a local branch up-to-date with its remote version, while also updating your other remote-tracking branches.

git pull = git fetch + git merge.



**Cherry-pick**

Cherry picking in Git means to choose a commit from one branch and apply it onto another.

This is in contrast with other ways such as merge and rebase which normally apply many commits onto another branch.

Make sure you are on the branch you want to apply the commit to.

git checkout master

Execute the following:

git cherry-pick <commit-hash>

If you cherry-pick from a public branch, you should consider using

git cherry-pick -x <commit-hash>

# [**Push a new local branch to a remote Git repository**](https://www.freecodecamp.org/forum/t/push-a-new-local-branch-to-a-remote-git-repository-and-track-it-too/13222)

1.Create a new branch:

git checkout -b feature\_branch\_name

2.Edit, add and commit your files.

3.Push your branch to the remote repository:

git push -u origin feature\_branch\_name