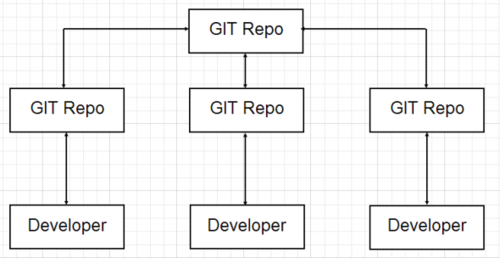
**Git is a distributed version control system** – which just means that when you do a **git clone (+url of your repository)** what you are actually getting is a complete copy of your entire history of that project. This means all your commits! Woot!

**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**



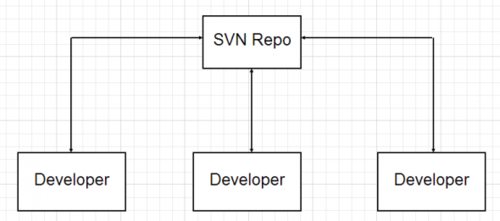
**What is SVN?**

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**

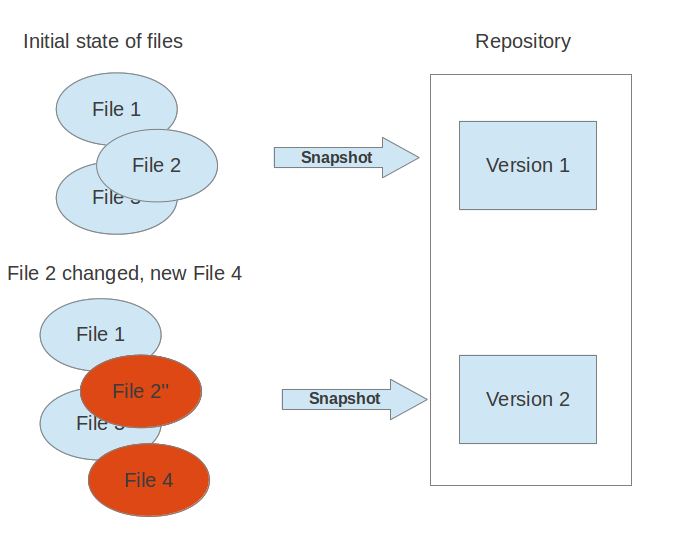
**SVN allows you to check out sub-trees (or branches) only whereas Git requires you to check out the entire repository as a unit.** This is because there is a .svn in each one of your folders while git only has one .git at the top level parent directory.

## [1. Introduction into version control systems](https://www.vogella.com/tutorials/Git/article.html#introduction-into-version-control-systems)

### [1.1. What is a version control system?](https://www.vogella.com/tutorials/Git/article.html#versioncontrolssystems)

A version control system (VCS) allows you to track the history of a collection of files. It supports creating different versions of this collection. Each version captures a snapshot of the files at a certain point in time and the VCS allows you to switch between these versions. These versions are stored in a specific place, typically called a repository.

You may, for example, revert the collection of files to a state from 2 days ago. Or you may switch between versions of your files for experimental features. The process of creating different versions (snapshots) in the repository is depicted in the following graphic. Please note that this picture fits primarily to Git. Other version control systems like Concurrent Versions System (CVS) don’t create snapshots of the files but store file deltas.



VCS are typically used to track changes in text files. These text files can for example be source code for a programming language, HTML or configuration files. Of course, version control systems are not limited to text files, they can also handle other types of files. For example, you may use a VCS to track the different versions of a png file.

### [1.2. Localized and centralized version control systems](https://www.vogella.com/tutorials/Git/article.html#cvcs_definition)

A localized version control system keeps local copies of the files. This approach can be as simple as creating a manual copy of the relevant files.

A centralized version control system provides a server software component which stores and manages the different versions of the files. A developer can copy (checkout) a certain version from the central sever onto their individual computer.

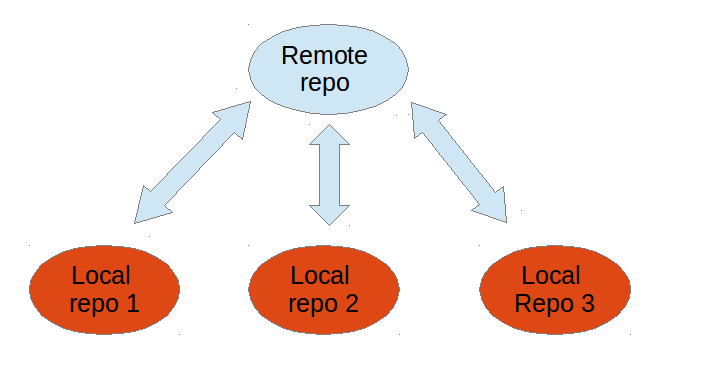
Both approaches have the drawback that they have one single point of failure. In a localized version control systems it is the individual computer and in a centralized version control systems it is the server machine. Both system makes it also harder to work in parallel on different features.

### [1.3. Distributed version control systems](https://www.vogella.com/tutorials/Git/article.html#dvcs_definition)

In a distributed version control system each user has a complete local copy of a repository on his individual computer. The user can copy an existing repository. This copying process is typically called cloning and the resulting repository can be referred to as a clone.

Every clone contains the full history of the collection of files and a cloned repository has the same functionality as the original repository.

Every repository can exchange versions of the files with other repositories by transporting these changes. This is typically done via a repository running on a server which is, unlike the local machine of a developer, always online. Typically, there is a central server for keeping a repository but each cloned repository is a full copy of this repository. The decision which of the copies is considered to be the central server repository is pure convention.



## [2. Introduction into Git](https://www.vogella.com/tutorials/Git/article.html#introduction-into-git)

### [2.1. What is Git?](https://www.vogella.com/tutorials/Git/article.html#gitterminlogy)

Git is currently the most popular implementation of a distributed version control system.

Git originates from the Linux kernel development and was founded in 2005 by Linus Torvalds. Nowadays it is used by many popular open source projects, e.g., the Android or the Eclipse developer teams, as well as many commercial organizations.

The core of Git was originally written in the programming language C, but Git has also been re-implemented in other languages, e.g., Java, Ruby and Python.

### [2.2. Git repositories](https://www.vogella.com/tutorials/Git/article.html#gitdefintion_localrepositories)

A Git repository contains the history of a collection of files starting from a certain directory. The process of copying an existing Git repository via the Git tooling is called *cloning*. After cloning a repository the user has the complete repository with its history on his local machine. Of course, Git also supports the creation of new repositories.

If you want to delete a Git repository, you can simply delete the folder which contains the repository.

If you clone a Git repository, by default, Git assumes that you want to work in this repository as a user. Git also supports the creation of repositories targeting the usage on a server.

* *bare repositories* are supposed to be used on a server for sharing changes coming from different developers. Such repositories do not allow the user to modify locally files and to create new versions for the repository based on these modifications.
* *non-bare repositories* target the user. They allow you to create new changes through modification of files and to create new versions in the repository. This is the default type which is created if you do not specify any parameter during the clone operation.

A *local non-bare Git repository* is typically called *local repository*.

[**2.3. Working tree**](https://www.vogella.com/tutorials/Git/article.html#workingtree)

A local repository provides at least one collection of files which originate from a certain version of the repository. This collection of files is called the *working tree*. It corresponds to a checkout of one version of the repository with potential changes done by the user.

The user can change the files in the *working tree* by modifying existing files and by creating and removing files.

A file in the working tree of a Git repository can have different states. These states are the following:

* untracked: the file is not tracked by the Git repository. This means that the file never staged nor committed.
* tracked: committed and not staged
* staged: staged to be included in the next commit
* dirty / modified: the file has changed but the change is not staged

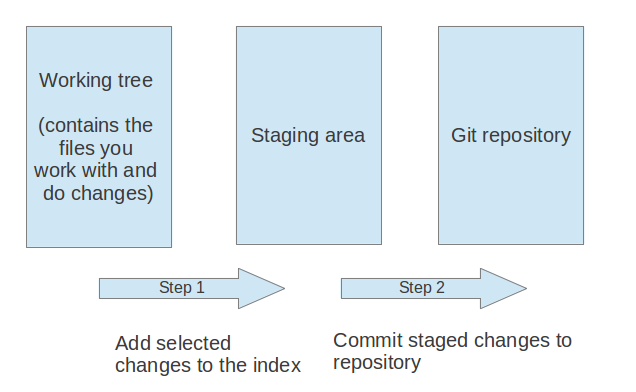
After doing changes in the working tree, the user can add these changes to the Git repository or revert these changes.

[**2.4. Adding to a Git repository via staging and committing**](https://www.vogella.com/tutorials/Git/article.html#gitaddingprocess)

After modifying your *working tree* you need to perform the following two steps to persist these changes in your local repository:

* add the selected changes to the *staging area* (also known as index) via the git add command
* commit the staged changes into the Git repository via the git commit command

This process is depicted in the following graphic.



The git add command stores a snapshot of the specified files in the staging area. It allows you to incrementally modify files, stage them, modify and stage them again until you are satisfied with your changes.

Some tools and Git user prefer the usage of the *index* instead of staging area. Both terms mean the same thing.

After adding the selected files to the staging area, you can *commit* these files to add them permanently to the Git repository. *Committing* creates a new persistent snapshot (called *commit*or *commit object*) of the staging area in the Git repository. A commit object, like all objects in Git, is immutable.

The *staging area* keeps track of the snapshots of the files until the staged changes are committed.

For committing the staged changes you use the git commit command.

If you commit changes to your Git repository, you create a new *commit object* in the Git repository. See [Commit object (commit)](https://www.vogella.com/tutorials/Git/article.html#commit_object) for information about the commit object.

[**2.5. Synchronizing with other Git repositories (remote repositories)**](https://www.vogella.com/tutorials/Git/article.html#gitdefintion_remoterepositories)

Git allows the user to synchronize the local repository with other (remote) repositories.

Users with sufficient authorization can send new version in their local repository to to remote repositories via the *push* operation. They can also integrate changes from other repositories into their local repository via the *fetch* and *pull* operation.

[**2.6. The concept of branches**](https://www.vogella.com/tutorials/Git/article.html#gitdefinition_branching)

Git supports *branching* which means that you can work on different versions of your collection of files. A branch allows the user to switch between these versions so that he can work on different changes independently from each other.

For example, if you want to develop a new feature, you can create a branch and make the changes in this branch. This does not affect the state of your files in other branches. For example, you can work independently on a branch called *production* for bugfixes and on another branch called feature\_123 for implementing a new feature.

Branches in Git are local to the repository. A branch created in a local repository does not need to have a counterpart in a remote repository. Local branches can be compared with other local branches and with *remote-tracking* branches. A remote-tracking branch proxies the state of a branch in another remote repository.

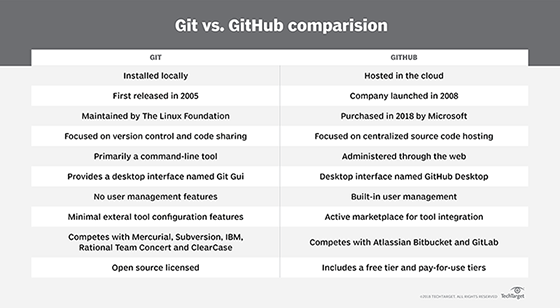
Git supports the combination of changes from different branches. The developer can use Git commands to combine the changes at a later point in time.

[**2.7. Summary of the core Git terminology**](https://www.vogella.com/tutorials/Git/article.html#gitterminology)

The following table provides a summary of important *Git* terminology discussed in this section.

| *Table 1. Git terminology* | |
| --- | --- |
| **Term** | **Definition** |
| Branch | A *branch* is a named pointer to a commit. Selecting a branch in Git terminology is called *to checkout* a branch. If you are working in a certain branch, the creation of a new commit advances this pointer to the newly created commit.  Each commit knows their parents (predecessors). Successors are retrieved by traversing the commit graph starting from branches or other refs, symbolic references (for example: HEAD) or explicit commit objects. This way a branch defines its own line of descendants in the overall version graph formed by all commits in the repository.  You can create a new branch from an existing one and change the code independently from other branches. One of the branches is the default (typically named *master* ). The default branch is the one for which a local branch is automatically created when cloning the repository. |
| Commit | When you commit your changes into a repository this creates a new *commit object* in the Git repository. This *commit object* uniquely identifies a new revision of the content of the repository.  This revision can be retrieved later, for example, if you want to see the source code of an older version. Each commit object contains the author and the committer. This makes it possible to identify who did the change. The author and committer might be different people. The author did the change and the committer applied the change to the Git repository. This is common for contributions to open source projects. |
| HEAD | *HEAD* is a symbolic reference most often pointing to the currently checked out branch.  Sometimes the *HEAD* points directly to a commit object, this is called *detached HEAD mode*. In that state creation of a commit will not move any branch.  If you switch branches, the *HEAD* pointer points to the branch pointer which in turn points to a commit. If you checkout a specific commit, the*HEAD* points to this commit directly. |
| Index | *Index* is an alternative term for the *staging area*. |
| Repository | A *repository* contains the history, the different versions over time and all different branches and tags. In Git each copy of the repository is a complete repository. If the repository is not a bare repository, it allows you to checkout revisions into your working tree and to capture changes by creating new commits. Bare repositories are only changed by transporting changes from other repositories.  This description uses the term *repository* to talk about a non-bare repository. If it talks about a bare repository, this is explicitly mentioned. |
| Revision | Represents a version of the source code. Git implements revisions as *commit objects* (or short *commits* ). These are identified by an SHA-1 hash. |
| Staging area | The *staging area* is the place to store changes in the working tree before the commit. The *staging area* contains a snapshot of the changes in the working tree (changed or new files) relevant to create the next commit and stores their mode (file type, executable bit). |
| Tag | A *tag* points to a commit which uniquely identifies a version of the Git repository. With a tag, you can have a named point to which you can always revert to. You can revert to any point in a Git repository, but tags make it easier. The benefit of tags is to mark the repository for a specific reason, e.g., with a release.  Branches and tags are named pointers, the difference is that branches move when a new commit is created while tags always point to the same commit. Tags can have a timestamp and a message associated with them. |
| URL | A URL in Git determines the location of the repository. Git distinguishes between *fetchurl* for getting new data from other repositories and *pushurl* for pushing data to another repository. |
| Working tree | The *working tree* contains the set of working files for the repository. You can modify the content and commit the changes as new commits to the repository. |

Git is a distributed version control tool that can manage a development project's source code history, while GitHub is a cloud based platform built around the Git tool. Git is a tool a developer installs locally on their computer, while GitHub is an online service that stores code pushed to it from computers running the Git tool. The key difference between Git and GitHub is that Git is an open-source tool developers install locally to manage source code, while GitHub is an online service to which developers who use Git can connect and upload or download resources.



Here are the Git commands which are being covered:

* **git config**
* **git init**
* **git clone**
* **git add**
* **git commit**
* **git diff**
* **git reset**
* **git status**
* **git rm**
* **git log**
* **git show**
* **git tag**
* **git branch**
* **git checkout**
* **git merge**
* **git remote**
* **git push**
* **git pull**
* **git stash**
* **Getting & Creating Projects**

| **Command** | **Description** |
| --- | --- |
| git init | Initialize a local Git repository |
| git clone ssh://git@github.com/[username]/[repository-name].git | Create a local copy of a remote repository |

* **Basic Snapshotting**

| **Command** | **Description** |
| --- | --- |
| git status | Check status |
| git add [file-name.txt] | Add a file to the staging area |
| git add –A | Add all new and changed files to the staging area |
| git commit -m "[commit message]" | Commit changes |
| git rm -r [file-name.txt] | Remove a file (or folder) |



* **Branching & Merging**

| **Command** | **Description** |
| --- | --- |
| git branch | List branches (the asterisk denotes the current branch) |
| git branch –a | List all branches (local and remote) |
| git branch [branch name] | Create a new branch |
| git branch -d [branch name] | Delete a branch |
| git push origin --delete [branch name] | Delete a remote branch |
| git checkout -b [branch name] | Create a new branch and switch to it |
| git checkout -b [branch name] origin/[branch name] | Clone a remote branch and switch to it |
| git checkout [branch name] | Switch to a branch |
| git checkout - | Switch to the branch last checked out |
| git checkout -- [file-name.txt] | Discard changes to a file |
| git merge [branch name] | Merge a branch into the active branch |
| git merge [source branch] [target branch] | Merge a branch into a target branch |
| git stash | Stash changes in a dirty working directory |
| git stash clear | Remove all stashed entries |

| **Command** | **Description** |
| --- | --- |
| git push origin [branch name] | Push a branch to your remote repository |
| git push -u origin [branch name] | Push changes to remote repository (and remember the branch) |
| git push | Push changes to remote repository (remembered branch) |
| git push origin --delete [branch name] | Delete a remote branch |
| git pull | Update local repository to the newest commit |
| git pull origin [branch name] | Pull changes from remote repository |
| git remote add origin ssh://git@github.com/[username]/[repository-name].git | Add a remote repository |
| git remote set-url origin ssh://git@github.com/[username]/[repository-name].git | Set a repository's origin branch to SSH |

* **Sharing & Updating Projects**
* **Inspection & Comparison**

| **Command** | **Description** |
| --- | --- |
| git log | View changes |
| git log –summary | View changes (detailed) |
| git diff [source branch] [target branch] | Preview changes before merging |

Merge conflicts happen when you merge branches that have competing commits, and Git needs your help to decide which changes to incorporate in the final merge.

Git can often resolve differences between branches and merge them automatically. Usually, the changes are on different lines, or even in different files, which makes the merge simple for computers to understand. However, sometimes there are competing changes that Git can't resolve without your help. Often, merge conflicts happen when people make different changes to the same line of the same file, or when one person edits a file and another person deletes the same file.

You must resolve all merge conflicts before you can merge a pull request on GitHub. If you have a merge conflict between the compare branch and base branch in your pull request, you can view a list of the files with conflicting changes above the **Merge pull request** button. The **Merge pull request** button is deactivated until you've resolved all conflicts between the compare branch and base branch.

You can resolve merge conflicts using the command line and a text editor.

Merge conflicts occur when competing changes are made to the same line of a file, or when one person edits a file and another person deletes the same file. For more information, see "[About merge conflicts](https://help.github.com/en/articles/about-merge-conflicts/)."

**Tip:** You can use the conflict editor on GitHub to resolve competing line change merge conflicts between branches that are part of a pull request. For more information, see "[Resolving a merge conflict on GitHub](https://help.github.com/en/articles/resolving-a-merge-conflict-on-github)."

### [Competing line change merge conflicts](https://help.github.com/en/articles/resolving-a-merge-conflict-using-the-command-line#competing-line-change-merge-conflicts)

To resolve a merge conflict caused by competing line changes, you must choose which changes to incorporate from the different branches in a new commit.

For example, if you and another person both edited the file styleguide.md on the same lines in different branches of the same Git repository, you'll get a merge conflict error when you try to merge these branches. You must resolve this merge conflict with a new commit before you can merge these branches.

1. Open Git Bash.
2. Navigate into the local Git repository that has the merge conflict.

cd REPOSITORY-NAME

1. Generate a list of the files affected by the merge conflict. In this example, the file styleguide.md has a merge conflict.
2. $ git status
3. > # On branch branch-b
4. > # You have unmerged paths.
5. > # (fix conflicts and run "git commit")
6. > #
7. > # Unmerged paths:
8. > # (use "git add ..." to mark resolution)
9. > #
10. > # both modified: styleguide.md
11. > #

> no changes added to commit (use "git add" and/or "git commit -a")

1. Open your favorite text editor, such as [Atom](https://atom.io/), and navigate to the file that has merge conflicts.
2. To see the beginning of the merge conflict in your file, search the file for the conflict marker <<<<<<<. When you open the file in your text editor, you'll see the changes from the HEAD or base branch after the line <<<<<<< HEAD. Next, you'll see =======, which divides your changes from the changes in the other branch, followed by >>>>>>> BRANCH-NAME. In this example, one person wrote "open an issue" in the base or HEAD branch and another person wrote "ask your question in IRC" in the compare branch or branch-a.
3. If you have questions, please
4. <<<<<<< HEAD
5. open an issue
6. =======
7. ask your question in IRC.
8. >>>>>>> branch-a
9. Decide if you want to keep only your branch's changes, keep only the other branch's changes, or make a brand new change, which may incorporate changes from both branches. Delete the conflict markers <<<<<<<, =======, >>>>>>> and make the changes you want in the final merge. In this example, both changes are incorporated into the final merge:
10. If you have questions, please open an issue or ask in our IRC channel if it's more urgent.
11. Add or stage your changes.

$ git add .

1. Commit your changes with a comment.

$ git commit -m "Resolved merge conflict by incorporating both suggestions."

You can now merge the branches on the command line or [push your changes to your remote repository](https://help.github.com/en/articles/pushing-to-a-remote/) on GitHub and [merge your changes](https://help.github.com/en/articles/merging-a-pull-request/) in a pull request.

### [Removed file merge conflicts](https://help.github.com/en/articles/resolving-a-merge-conflict-using-the-command-line#removed-file-merge-conflicts)

To resolve a merge conflict caused by competing changes to a file, where a person deletes a file in one branch and another person edits the same file, you must choose whether to delete or keep the removed file in a new commit.

For example, if you edited a file, such as README.md, and another person removed the same file in another branch in the same Git repository, you'll get a merge conflict error when you try to merge these branches. You must resolve this merge conflict with a new commit before you can merge these branches.

1. Open Git Bash.
2. Navigate into the local Git repository that has the merge conflict.

cd REPOSITORY-NAME

1. Generate a list of the files affected by the merge conflict. In this example, the file README.md has a merge conflict.
2. $ git status
3. > # On branch master
4. > # Your branch and 'origin/master' have diverged,
5. > # and have 1 and 2 different commits each, respectively.
6. > # (use "git pull" to merge the remote branch into yours)
7. > # You have unmerged paths.
8. > # (fix conflicts and run "git commit")
9. > #
10. > # Unmerged paths:
11. > # (use "git add/rm ..." as appropriate to mark resolution)
12. > #
13. > # deleted by us: README.md
14. > #

> # no changes added to commit (use "git add" and/or "git commit -a")

1. Open your favorite text editor, such as [Atom](https://atom.io/), and navigate to the file that has merge conflicts.
2. Decide if you want keep the removed file. You may want to view the latest changes made to the removed file in your text editor.

To add the removed file back to your repository:

$ git add README.md

To remove this file from your repository:

$ git rm README.md

> README.md: needs merge

> rm 'README.md'

1. Commit your changes with a comment.
2. $ git commit -m "Resolved merge conflict by keeping README.md file."

> [branch-d 6f89e49] Merge branch 'branch-c' into branch-d