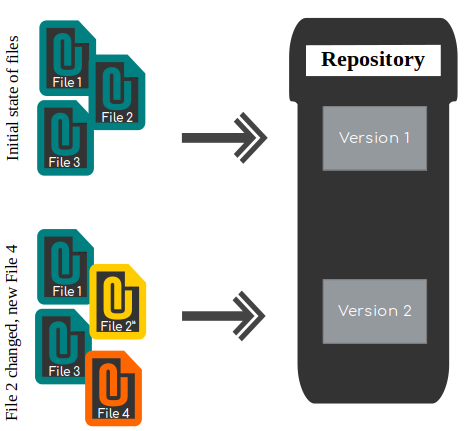
**[1. Introduction into version control systems](https://www.vogella.com/tutorials/Git/article.html" \l "introduction-into-version-control-systems)**

[**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 manage a collection of files and gives access to different versions of these files.

The VCS allows you to capture the content and structure of your files at a certain point in time. You can use the VCS to switch between these versions and you can work on different versions of these files in parallel. The different versions are stored in a storage system which is typically called a *repository*. The process of creating different versions (snapshots) in the repository is depicted in the following graphic.



In this example, your repository contains two versions, one with three files and another version with four files, two one them in the same state as in the first version, one modified one and another new one.

VCS are very good at tracking changes in text files. For example, you may track changes in HTML code or Java source code. It is also possible to use VCS for other file types but VCS are not that efficient to trace changes in binary files.

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 server onto their individual computer.

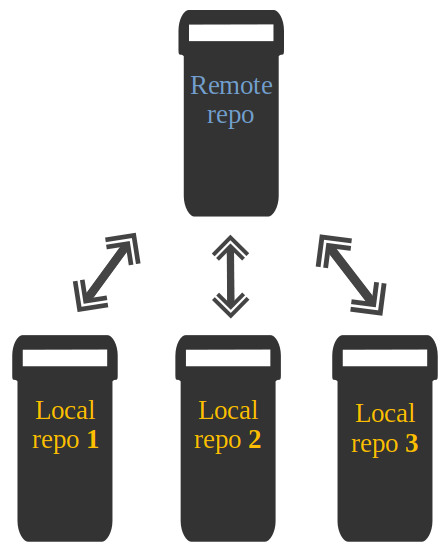
Both approaches have the drawback that they have one single point of failure. In a localized version control system it is the individual computer and in a centralized version control system it is the server machine. Both systems also make it harder to work in parallel on different features. To remove the limitations of local and centralized version control systems, distributed version control systems have been created.

[**1.2. 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 regarding 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)

The following description gives you a very high-level overview of the Git version control system.

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

*Git* is the leading 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., Visual Studio Code from Microsoft, Android from Google 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 and working trees**](https://www.vogella.com/tutorials/Git/article.html#gitdefintion_localrepositories)

A Git repository manages a collection of files in a certain directory. A Git repository is file based, i.e., all versions of the managed files are stored on the file system.

A Git repository can be designed to be used on a server or for a user:

* *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 local 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*.

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

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

Every local repository has a *working tree*. The files in the working tree may be new or based on a certain version from the repository. The user can change and create files or delete them.

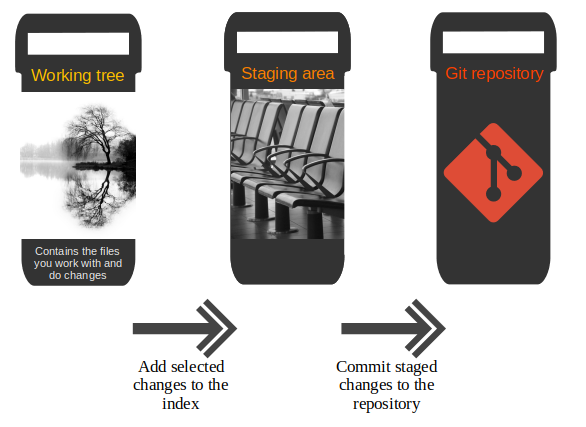
After doing changes in the working tree, the user can capture new versions of the files in the Git repository. Or the user can restore files to a state already captured by Git.

[**2.3. Adding a new version of the files to a Git repository**](https://www.vogella.com/tutorials/Git/article.html#gitaddingprocess)

After modifying files in your *working tree* you need to perform two steps to add them to your local repository.

* mark the desired file changes as relevant for the next commit; this operation is called staging
* instruct Git to create a new version of the managed files via the commit operation, the new created version is called *commit*.

This process is depicted in the following graphic.



During the stage operation, copies of the specified files are added to a persisted storage called the *staging area* (sometimes it is also called index). This allows you to do further modifications to the same file without including these modifications in the next commit. You can repeat the staging operation until you are satisfied and continue with the commit operation.

The *commit* operation creates a new persistent snapshot called *commit object* (short form: *commit*) of the managed files in the Git repository. A commit object, like all objects in Git, is immutable.

[**2.4. Alternative versions of files with branches**](https://www.vogella.com/tutorials/Git/article.html#gitdefinition_branching)

Git allows you to work on different versions of your files in parallel. For this, Git uses *branches*. 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.

[**2.5. Summary of the core Git terminology**](https://www.vogella.com/tutorials/Git/article.html#summary-of-the-core-git-terminology)

The following table summarizes important *Git* terminology. It is intended to be used as a reference, so you can skip this now and return to it if you need clarification.

| **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 its 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.  The term *repository* typically refers to a non-bare repository. If a bare repository is referred, 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. |

[**3. Git configuration**](https://www.vogella.com/tutorials/Git/article.html#setup)

Git requires at least the user name and a valid email to work. [Git settings](https://www.kernel.org/pub/software/scm/git/docs/git-config.html) for all possible settings. This description describes the most important ones.

[**3.1. Git configuration levels**](https://www.vogella.com/tutorials/Git/article.html#setup_configurationlevels)

You configure git via the git config command. These settings can be system wide, user or repository specific. A setting for the repository overrides the user setting and a user setting overrides a system wide setting.

[3.1.1. Git system-wide configuration](https://www.vogella.com/tutorials/Git/article.html#setup_systemwideconfiguration)

You can provide a system wide configuration for your Git settings. A system wide configuration is not very common. Most settings are user specific or repository specific as described in the next chapters.

On a Unix based system, Git uses the /etc/gitconfig file for this system-wide configuration. To set this up, ensure you have sufficient rights, i.e. root rights, in your OS and use the --system option for the git config command.

[3.1.2. Git user configuration](https://www.vogella.com/tutorials/Git/article.html#setup_userconfiguration)

Git allows you to store user settings in the .gitconfig file located in the user home directory. This is also called the *global* Git configuration.

For example Git stores the committer and author of a change in each commit. This and additional information can be stored in the Git user settings.

In each Git repository you can also configure the settings for this repository. User configuration is done if you include the --global option in the git config command.

[3.1.3. Repository specific configuration](https://www.vogella.com/tutorials/Git/article.html#setup_configuration)

You can also store repository specific settings in the .git/config file of a repository. Use the --local or use no flag at all. If neither the --system not the --global parameter is used, the setting is specific for the current Git repository.

[**6. Configure files and directories to ignore**](https://www.vogella.com/tutorials/Git/article.html#configure-files-and-directories-to-ignore)

[**6.1. Ignoring files and directories with a .gitignore file**](https://www.vogella.com/tutorials/Git/article.html#ignoring-files-and-directories-with-a-.gitignore-file)

Git can be configured to ignore certain files and directories for repository operations. This is configured via one or several .gitignore files. Typically, this file is located at the root of your Git repository but it can also be located in sub-directories. In the second case the defined rules are only valid for the sub-directory and below.

You can use certain wildcards in this file. \* matches several characters. More patterns are possible and described under the following URL: [gitignore manpage](https://www.kernel.org/pub/software/scm/git/docs/gitignore.html)

For example, the following .gitignore file tells Git to ignore the bin and target directories and all files ending with a ~.

You can create the .gitignore file in the root directory of the working tree to make it specific for the Git repository.

[**7. Working with a (local) remote repository**](https://www.vogella.com/tutorials/Git/article.html#exercise_workingwithremotes)

You now create a local bare repository based on your existing Git repository. In order to simplify the examples, the Git repository is hosted locally in the filesystem and not on a server in the Internet.

Afterwards you pull from and push to your bare repository to synchronize changes between your repositories.

[**7.1. Create a Git repository via the clone operation**](https://www.vogella.com/tutorials/Git/article.html#remotes_setupexercise)

Execute the following commands to create a bare repository based on your existing Git repository.

**#** switch to the folder where you keep your projects

cd ~/projects

**#** create a new repository by cloning

git clone *[url]*

[**8. Using Branches**](https://www.vogella.com/tutorials/Git/article.html#using-branches)

[**8.1. What are branches?**](https://www.vogella.com/tutorials/Git/article.html#gitbranch_def)

Git allows you to create *branches*. Branches are named pointers to commits. You can work on different branches independently from each other. The default branch is most often called *master*.

A branch pointer in Git is 41 bytes large, 40 bytes of characters and an additional new line character. Therefore, the creating of branches in Git is very fast and cheap in terms of resource consumption. Git encourages the usage of branches on a regular basis.

If you decide to work on a branch, you *checkout* (or *switch* to) this branch. This means that Git populates the *working tree* with the version of the files from the commit to which the branch points and moves the *HEAD* pointer to the new branch.

*HEAD* is a symbolic reference usually pointing to the branch which is currently checked out.

[**8.2. List available branches**](https://www.vogella.com/tutorials/Git/article.html#gitbranch_listbranches)

The git branch command lists all local branches. The currently active branch is marked with \*.

**#** lists available branches

git branch

If you want to see all branches (including remote-tracking branches), use the -a for the git branch command.

**#** lists all branches including the remote branches

git branch -a

The -v option lists more information about the branches.

In order to list branches in a remote repository use the git branch -r command as demonstrated in the following example.

**#** lists branches **in** the remote repositories

git branch -r

[**8.3. Create new branch**](https://www.vogella.com/tutorials/Git/article.html#gitbranch_createnewbranch)

You can create a new branch via the git branch [newname] command. This command allows you to specify the commit (commit id, tag, remote or local branch) to which the branch pointer original points. If not specified, the commit to which the HEAD reference points is used to create the new branch.

**#** syntax: git branch <name> <**hash>**

**#** <**hash>** **in** the above is optional

git branch testing

[**8.4. Checkout branch**](https://www.vogella.com/tutorials/Git/article.html#gitbranch_checkout)

To start working in a branch you have to *checkout* the branch. If you *checkout* a branch, the HEAD pointer moves to the last commit in this branch and the files in the working tree are set to the state of this commit.

The following commands demonstrate how you switch to the branch called *testing*, perform some changes in this branch and switch back to the branch called *master*.

**#** switch to your new branch

git checkout testing

**#** **do** some changes

**echo "Cool new feature in this branch" >** test01

git commit -a -m "new feature"

**#** switch to the master branch

git checkout master

**#** check that the content of

**#** the test01 file is the old one

cat test01

To create a branch and to switch to it at the same time you can use the git checkout command with the -b parameter.

**#** create branch and switch to it

git checkout -b bugreport12

**#** creates a new branch based on the master branch

**#** without the last commit

git checkout -b mybranch master~1

[**9. Merging**](https://www.vogella.com/tutorials/Git/article.html#gitmerge_definition)

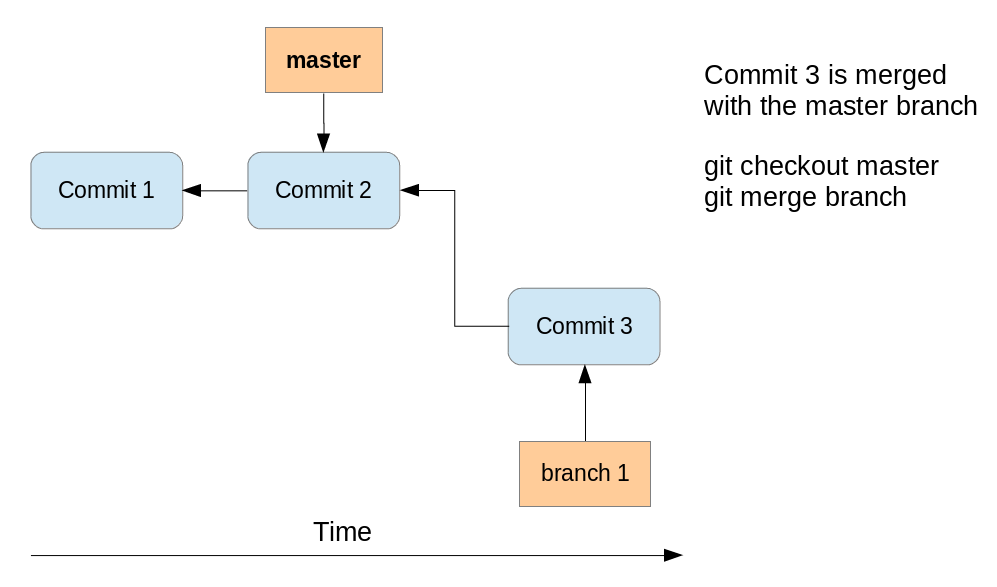
Git allows you to combine the changes which were created on two different branches. One way to achieve this is *merging*, which is described in this chapter. You can merge based on branches, tags or commits. Other ways are using rebase or cherry-pick.

This part explains how to merge changes between two different branches under the assumption that no merging conflicts happen. Solving conflicts is covered in [What is a conflict during a merge operation?](https://www.vogella.com/tutorials/Git/article.html#mergeconflict_definition).

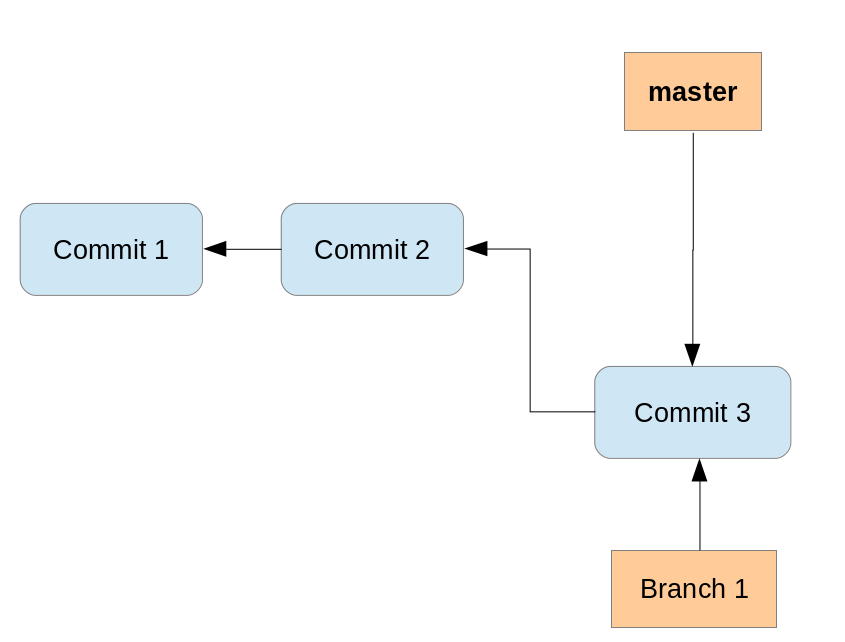
[**9.1. Fast-forward merge**](https://www.vogella.com/tutorials/Git/article.html#fast-forward-merge)

If the commits which are merged are direct successors of the *HEAD* pointer of the current branch, Git performs a so-called *fast forward merge*. This *fast forward merge* only moves the *HEAD* pointer of the current branch to the tip of the branch which is being merged.

This process is depicted in the following diagram. The first picture assumes that master is checked out and that you want to merge the changes of the branch labeled "branch 1" into your "master" branch. Each commit points to its predecessor (parent).

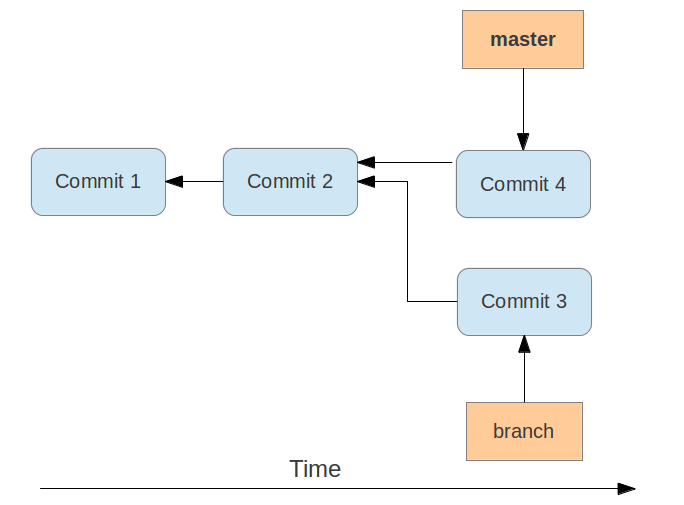


After the fast-forward merge the *HEAD* points to the master branch pointing to "Commit 3". The "branch 1" branch points to the same commit.

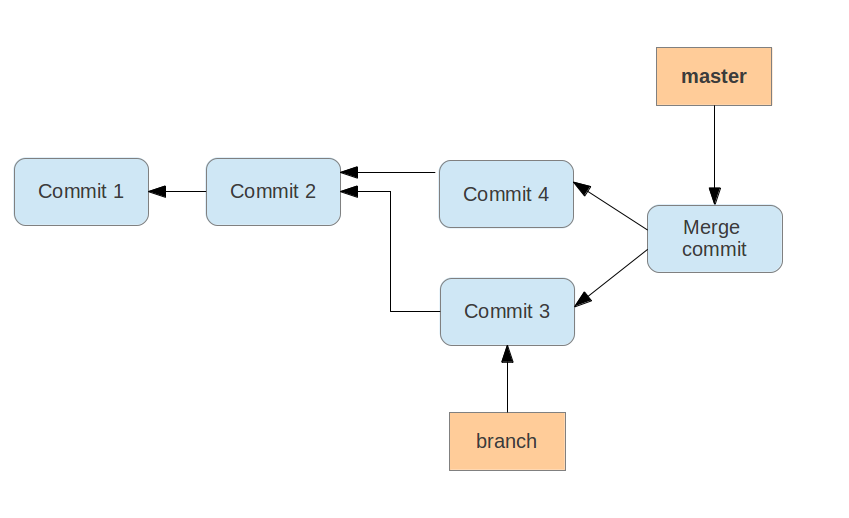


[**9.2. Merge commit**](https://www.vogella.com/tutorials/Git/article.html#gitmerge_mergecommit)

If commits are merged which are not direct predecessors of the current branch, Git performs a so-called *three-way-merge* between the latest commits of the two branches, based on the most recent common predecessor of both.



As a result a so-called *merge commit* is created on the current branch. It combines the respective changes from the two branches being merged. This commit points to both of its predecessors.



If multiple common predecessors exist, Git uses recursion to create a virtual common predecessor. For this Git creates a merged tree of the common ancestors and uses that as the reference for the 3-way merge. This is called the *recursive merge* strategy and is the default merge strategy.

Typically, you would merge by creating a merge request on the remote (such as in gitlab or github).