Contents

[1. Introduction to Version Control Systems 2](#_Toc89442927)

[1.1 Benefits of version control systems 2](#_Toc89442928)

[1.2 Localized and centralized version control systems 2](#_Toc89442929)

[1.3 Distributed version control systems 3](#_Toc89442930)

[1.4 DVCS Terminologies 3](#_Toc89442931)

[1.4.1 Local Repository 3](#_Toc89442932)

[Every VCS tool provides a private workplace as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository. Git takes it one step further by providing them a private copy of the whole repository. Users can perform many operations with this repository such as add file, remove file, rename file, move file, commit changes, and many more. 3](#_Toc89442933)

[1.4.2 Working Directory and Staging Area or Index 3](#_Toc89442934)

[2. GIT 4](#_Toc89442935)

[1.1 Advantages: 4](#_Toc89442936)

[3. GIT Repository and Work space: 4](#_Toc89442937)

# Introduction to Version Control Systems

Version control systems are categories of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Software developers working in teams are continually writing new source code and changing existing source code. The code for a project, app or software component is typically organized in a folder structure or "file tree". One developer on the team may be working on a new feature while another developer fixes an unrelated bug by changing code, each developer may make their changes in several parts of the file tree.

Version control helps teams solve these kinds of problems, tracking every individual change by each contributor and helping prevent concurrent work from conflicting. Changes made in one part of the software can be incompatible with those made by another developer working at the same time. This problem should be discovered and solved in an orderly manner without blocking the work of the rest of the team

VCS are known as **SCM (Source Code Management)** tools **or RCS (Revision Control System).**

## Benefits of version control systems

1. **Change history of every file**: Every change made by many individuals over the years. Changes include the creation and deletion of files as well as edits to their contents. Having the complete history enables going back to previous versions to help in root cause analysis for bugs and it is crucial when needing to fix problems in older versions of software.
2. **Branching and merging:** Creating a "branch" in VCS tools keeps multiple streams of work independent from each other while also providing the facility to merge that work back together, enabling developers to verify that the changes on each branch do not conflict
3. **Traceability:** Being able to trace each change made to the software and connect it to project management and bug tracking software such as Jira, and being able to annotate each change with a message describing the purpose and intent of the change can help not only with root cause analysis and other forensics.

## Localized and centralized version control systems

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

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.

DVCS clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the server goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

Example: GIT

## 1.4 DVCS Terminologies

### 1.4.1 Local Repository

### Every VCS tool provides a private workplace as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository. Git takes it one step further by providing them a private copy of the whole repository. Users can perform many operations with this repository such as add file, remove file, rename file, move file, commit changes, and many more.

### 1.4.2 Working Directory and Staging Area or Index

**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.



**1.4.3 Blobs**:

A Git blob (binary large object) is the object type used to store the contents of each file in a repository

# GIT

## Advantages:

1. Free and Open Source
2. Fast and Small- No need to interact with remote server for every operation. Need to connect to server to push changes. Git is written in C, which avoids runtime overheads associated with other high-level languages. Though Git mirrors entire repository, the size of the data on the client side is small. This illustrates the efficiency of Git at compressing and storing data on the client side.
3. Implicit back up – data present in client can be used for recovery
4. Security -Git uses a common cryptographic hash function called secure hash function (SHA1), to name and identify objects within its database. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. It implies that, it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git.
5. No need of Powerful hardware - developers don’t interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side, so the server hardware can be very simple indeed.
6. Easier branching - branch management with Git is very simple. It takes only a few seconds to create, delete, and merge branches.

# GIT Repository and Workspace:

The **repository** is essentially the .git hidden folder inside the working directory (workspace).

The **working directory** (**workspace**) is essentially your project folder.

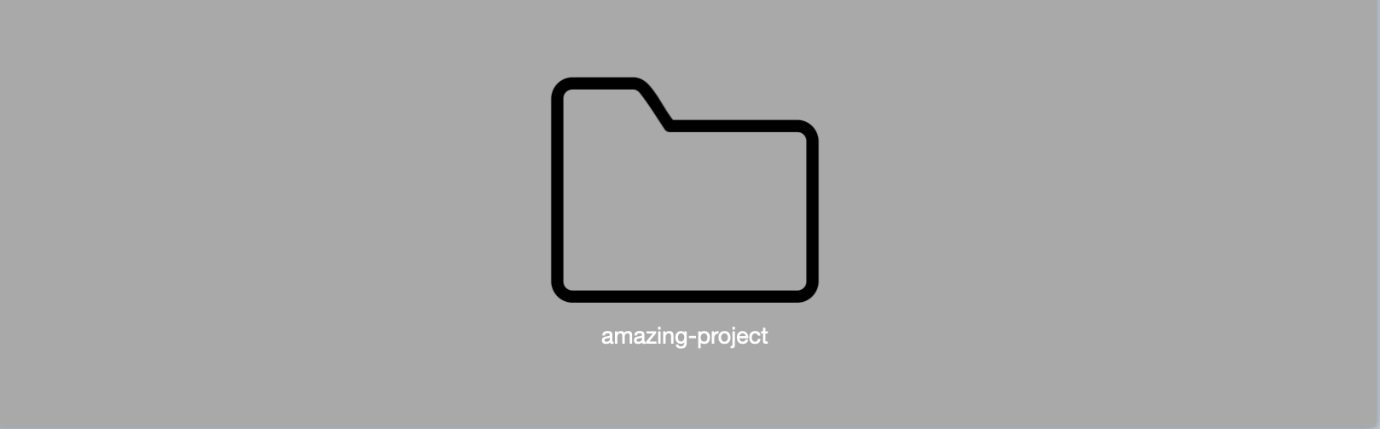
Keep in mind these are basically synonyms: workspace = working directory = project folder

Let me show the difference with an example.

**Step 1: Creating a working directory/workspace**

Assume you create a project folder locally on your computer. I'm going to call mine amazing-project.

Then what you have is this, a project folder called 'amazing-project'.

[](https://i.stack.imgur.com/lVzlw.png)

At this point amazing-project is NOT a repository.  Right now it is just a working directory (workspace). It is just a project folder.

**Step 2: Creating a repository**

Now assume you want to make amazing-project into a repository so that you have a local repository.

In order to do this you will need to use the 'git init' command.   This command will create a hidden folder inside your amazing-project folder called '.git'. This is essentially your repository.

Now what you have is this:

[](https://i.stack.imgur.com/TBFIh.png)

**Step 3: The working directory and repository in depth**

Now let's look at the workspace and repository in more depth.   We can distinguish between the two.

[](https://i.stack.imgur.com/ONBYG.png)

The amazing-project folder like we said represents our working directory:

[](https://i.stack.imgur.com/ZoaIp.png)

And the .git folder represents our repository:

[](https://i.stack.imgur.com/yfRUx.png)

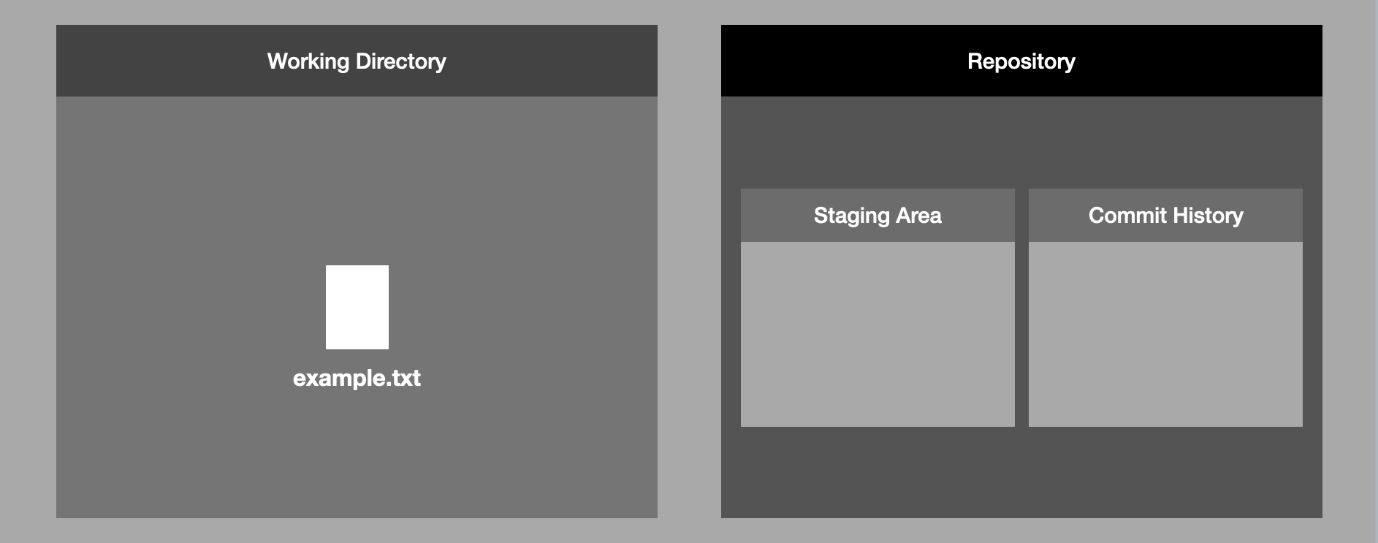
And actually within our repository there are in a way two important place to keep in mind. The staging area and the commit history.

[](https://i.stack.imgur.com/E2pcr.png)

**Step 4: Adding a file**

Now let's add the first file, I'll call it example.txt.

So using our diagram this is:

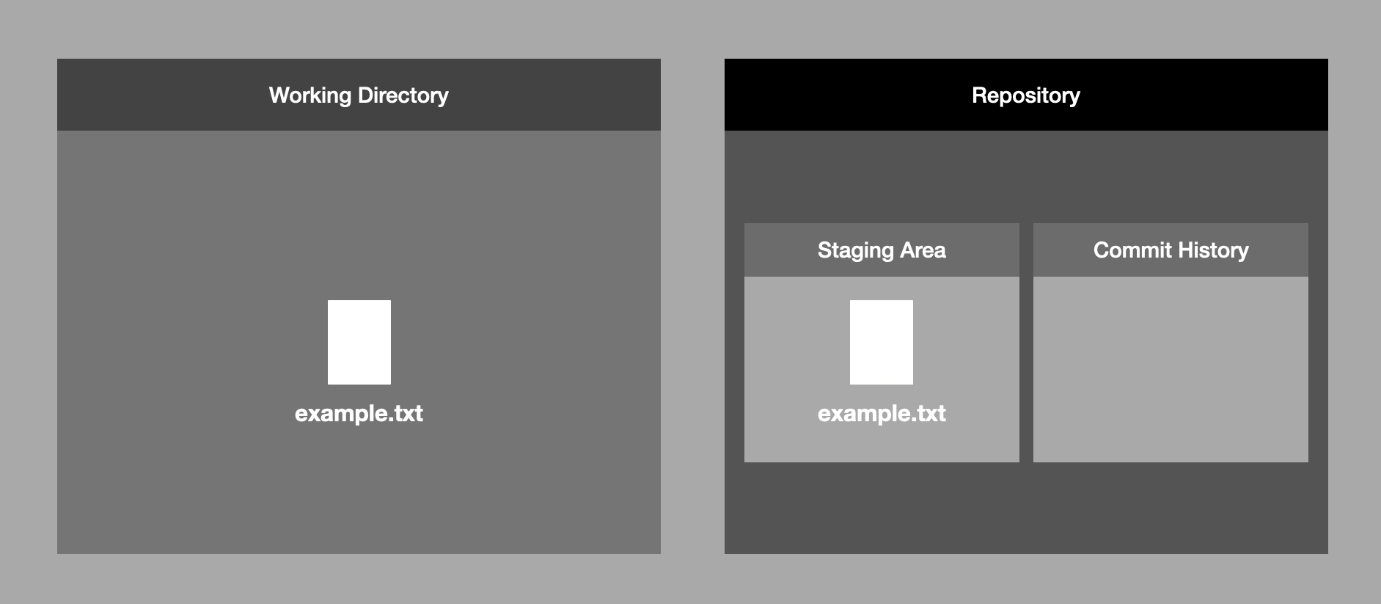
[](https://i.stack.imgur.com/KqsUo.png)

So, the file is in our working directory (workspace).   But this file is NOT part of our repository. This file is NOT yet being version controlled.

**Step 5: Adding the file to our repository**

In order for this file to come part of our repository and to start being version controlled we need to let Git know about it by adding it to the staging area using the 'git add' command.

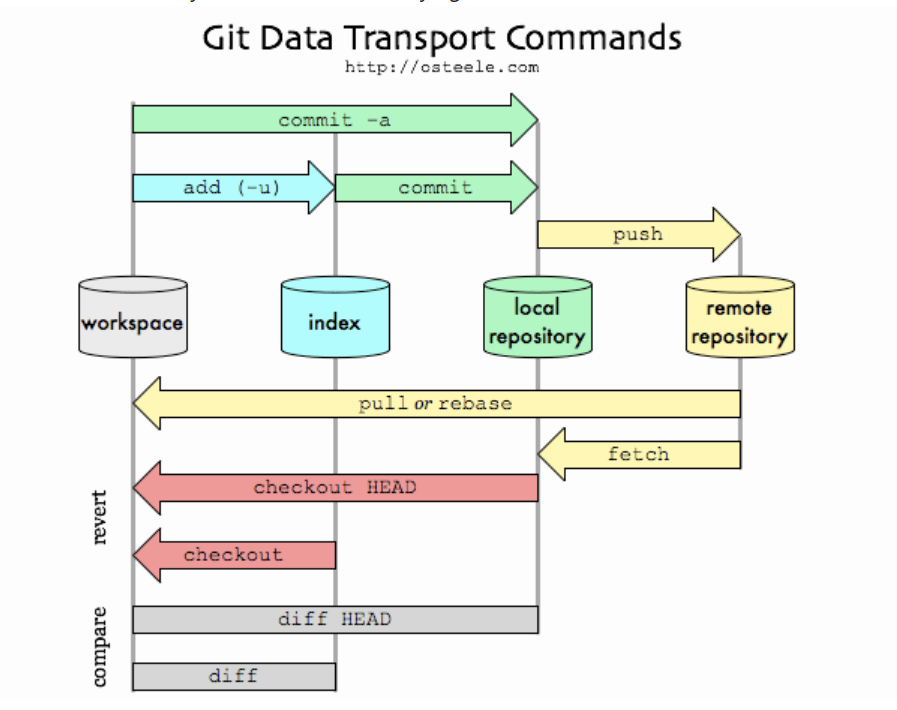
So then what we have is:

[](https://i.stack.imgur.com/EMuDq.png)

The 'git add' command copies the file from the working directory to the staging area.

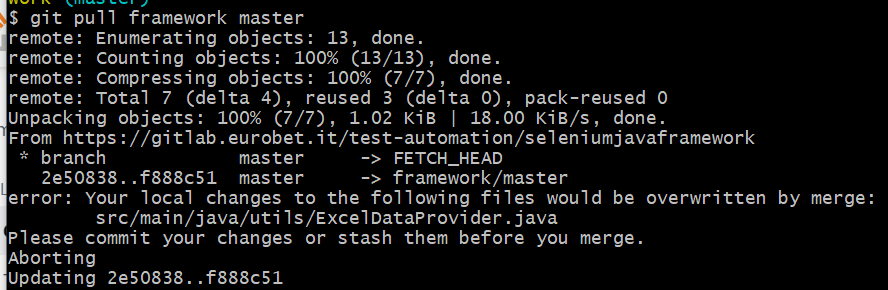
WARNING: A common misconception is that the file is moved, that is not the case. The file is copied.   Now the file is being tracked by Git. And if you were to commit now it would be included in your commit.

# GIT DATA Transport Commands:



## Get latest changes from GIT repo.

Attempting a git pull when you have unstaged changes will fail, saying you can commit or stash then.



### Ignore local changes and Pull

|  |
| --- |
| reset the master:  git reset --hard origin/master |



### Frequently Used Commands

|  |
| --- |
| git config --global http.proxy http://1xx.1xx.1xx.xx:xxxx  git init  git remote add origin https://username:token@gitlab.eurobet.it/test-automation/payments.git  **To Pull Complete code from GIT**  ==================================  git pull origin master  **To Push Fresh code to GIT**  ===============================  git add "/Automation.java"  git add \*  git add "/AutomationFolder"  git commit -m " testcases Push"  git push origin master  **To Check out Single file from GIT**  ==================================  git fetch  git checkout origin/master -- GeneralObjectsRepo.java  **To check git local repo name**  =================  basename `git rev-parse --show-toplevel`  git remote -v  **Git merge**  ==========  git commit -m "coments" filelist  git pull ITPmsHome homePage  \*\*\*\*merge conflicts manually - add files for commit at one go  git commit -m "merged"  git push ITPmsHome homePage  **create new branch**  ==========  git branch homePage  **Switch to the branch:**  ==========  git checkout AppTests\_Android    **Shows all local Branches:**  ==========  $ git branch  **reset single file**  ===================  git checkout -- newUserRegistraion.xml |