

**SWDVC301**

## VERSION CONTROL

### CONDUCT VERSION CONTROL

#### Competence

RQF Level: 3

Learning Hours



80

Credits: 8

Sector: ICT AND MULTIMEDIA

Trade: SOFTWARE DEVELOPMENT

Module Type: Specific

Curriculum: TVET Certificate III in Software Development

Prepared by CYUZUZO Emm.

Purpose statement	This specific module describes the skills required to Apply version control. Upon completion of this module, the learner will be able to: Setup repository, Manipulate files, Ship codes.					
Delivery modality	Training delivery		100%	Assessment		Total 100%
	Theoretical content		15%	Formative assessment	15%	50%
	Practical work:		85%		85%	
	• Group project and presentation	35%				
	• Individual project /Work	50%				
			Summative/Integrated Assessment			50%

## Elements of Competency and Performance Criteria

Elements of competency	Performance criteria
<b>1. Setup repository</b>	1.1 Git is introduced based on version control
	1.2 Git is properly initiated based on Git commands
	1.3 Repository is properly created based on the project.
	1.4 Remote URL is properly set in accordance with Git commands
<b>2. Manipulate files</b>  <b>3. Ship codes</b>	2.1 File changes are properly added according to Git commands
	2.2 File changes are properly committed based on added files
	2.3 Branches are properly managed based on the project
	3.1 Files are properly fetched in accordance with Git instruction
	3.2 Files are properly pushed to remote branch based on committed files
	3.3 Branches are properly merged based on pull request created

## Course content

Learning outcomes	At the end of this module learner will be able to:  1. Setup repository  2. Manipulate files  3. Ship codes
Learning outcome 1: Setup repository	Learning hours: 25hours
Indicative content	
<ul style="list-style-type: none"><li>• Definition of general key terms<ul style="list-style-type: none"><li>✓ Version control</li><li>✓ Git</li><li>✓ GitHub</li><li>✓ Terminal</li></ul></li><li>• <b>Introduction to version control</b><ul style="list-style-type: none"><li>✓ Types of version control<ul style="list-style-type: none"><li>• Local version control</li><li>• Centralized version control system</li><li>• Distributed version control</li></ul></li><li>✓ Well Known version control system<ul style="list-style-type: none"><li>• Git</li><li>• CVS (Concurrent Version System)</li><li>• mercurial</li><li>• SVN(subversion)</li></ul></li><li>✓ Benefits of Version control</li><li>✓ Application of version control</li></ul></li><li>• Description of git<ul style="list-style-type: none"><li>✓ Git Basic concept</li><li>✓ Git architecture</li><li>✓ Git workflow</li><li>✓ Initialisation of Git<ul style="list-style-type: none"><li>• Terminal Basic commands</li><li>• Installation of Git Setup</li></ul></li><li>✓ Configure Git<ul style="list-style-type: none"><li>• Git init command</li><li>• Git config command</li><li>• Git – version command</li></ul></li></ul></li></ul>	

✓ Configure .git ignore file

- **Use of GitHub repository**

- ✓ Description of GitHub
- ✓ Create account on GitHub
- ✓ Create new remote repository
- ✓ Apply git commands related to repository
  - Git clone
  - Git remote

### Resources required for the learning outcome

Equipment	<ul style="list-style-type: none"><li>● Computer</li><li>● projector</li><li>● White board</li></ul>
Materials	<ul style="list-style-type: none"><li>● Internet</li><li>● Electricity</li><li>● Flipchart</li><li>● Marker pen</li></ul>
Tools	Git, GitHub, Text editor(vs code),Terminal(CMD,Gitbash).
Facilitation techniques	<ul style="list-style-type: none"><li>● Demonstration</li><li>● individual and group work</li><li>● practical exercise</li><li>● individualized</li><li>● trainer guided</li><li>● group discussion</li><li>● brainstorming</li></ul>
Formative assessment methods	<ul style="list-style-type: none"><li>● Written assessment</li><li>● Performance assessment</li></ul>

**Learning outcome 2:**  
**Manipulate files**

Learning hours: 25hours

Indicative content

- **Definition of general key terms**
  - ✓ Status
  - ✓ Branch
  - ✓ commit
- **Add file change to git staging area**
  - ✓ Operation on git status command
    - View new untracked file
    - View modified file
    - View deleted file
  - ✓ Operation on git add command
    - Stage all files
    - Stage a file
    - Stage folder
  - ✓ Operation on git reset command
    - Unstage a file
    - Deleting and staging file/folder
  - ✓ Operation on rm command
    - Remove and stage a file
    - Remove and stage a folder
- **Commit File changes to git local repository**
  - ✓ Best practice of creating a commit message
  - ✓ Operation on git commit command
    - Commit a file
    - Edit commit message
  - ✓ Operation on git log command
    - To see simplified list of commit
    - To see a list of commits with more detail
- **Manage branches**
  - ✓ Operations on branches
    - Create branch
    - List branch
    - Delete local and remote branch
    - Switch branch

- Rename branch

### Resources required for the indicative content

Equipment	<ul style="list-style-type: none"> <li>• Computer</li> <li>• projector</li> <li>• White board</li> </ul>
Materials	<ul style="list-style-type: none"> <li>• Internet, Electricity</li> <li>• Flipchart</li> <li>• Marker pen</li> </ul>
Tools	Git, GitHub, Text editor(vs code),Terminal (CMD,Gitbash).
Facilitation techniques	<ul style="list-style-type: none"> <li>• Demonstration</li> <li>• individual and group work</li> <li>• practical exercise</li> <li>• individualized</li> <li>• trainer guided</li> <li>• group discussion</li> <li>• brainstorming</li> </ul>
Formative assessment methods	<ul style="list-style-type: none"> <li>• Written assessment</li> <li>• Performance assessment</li> <li>• Presentation</li> <li>• Project Based Assessment</li> </ul>

## Indicative content

- **Definition of general key terms**
  - pull
  - fetch
  - push
  - pull request
  - merge
- **Fetch file from GitHub repository**
  - ✓ Operation on git fetch command
    - Fetch the remote repository
    - Fetch the specific branch
    - Fetch all the branch simultaneously
    - Synchronize the local repository
  - ✓ Operation on git pull
    - Default git pull
    - Git pull remote branch
    - Git force pull
    - Git pull origin master
- **Push files to remote branch**
  - ✓ Tags used on git push command
  - ✓ operation on git push
    - push on origin master
    - git push force
    - git push verbose
    - delete a remote branch
- **Merge branches on remote repository**
  - ✓ operation on git rebase command
  - ✓ create pull request
  - ✓ operation on git merge
    - merge the specified commit to current active branch
    - merge commits into the master branch
    - git merge branch

Resources required for the indicative content	
Equipment	<ul style="list-style-type: none"> <li>• Computer,</li> <li>• projector</li> <li>• White board</li> </ul>
Materials	<ul style="list-style-type: none"> <li>• Internet</li> <li>• Electricity</li> <li>• Flipchart</li> <li>• Marker pen</li> </ul>
Tools	<ul style="list-style-type: none"> <li>• Git</li> <li>• GitHub</li> <li>• Text editor (vs code)</li> <li>• Terminal ( CMD, Gitbash ).</li> </ul>
Facilitation techniques	<ul style="list-style-type: none"> <li>• Demonstration</li> <li>• individual and group work</li> <li>• practical exercise</li> <li>• individualized</li> <li>• trainer guided</li> <li>• group discussion</li> <li>• brainstorming</li> </ul>
Formative assessment methods(CAT)	<ul style="list-style-type: none"> <li>• Written assessment</li> <li>• Performance assessment</li> <li>• Presentation</li> <li>• Project based assessment</li> </ul>



**Integrated Situation**

**SEZERANO ALPHA Crispin** is a Senior Developer of Innovate company Ltd located at Huye District, he assigned a project to 5 developers to design web application that contains different forms such as: login form, Student Registration form, Entertainment form, courses registration form and book registration form, using html language, but due to Covid-19 pandemic developers were not able to work together on the given task and become difficult to control them. Senior developer decided to assign tasks to each developer respectively and remotely. and he recommended them to work individually on a given task.

He told them to create their own repository related to a given task,

As a one of 5 developers you are hired to choose one task from above, work on it and create Pull Request of your task to be merged on the main branch, use GitHub as version control platform.

Submit Pull Request link to the senior developer.

**Tools, material and equipment are provided**

**This integrated situation is inclusiveness**

**The time allowed to accomplish this task is 4 hours**

## Resources

Tools	<ul style="list-style-type: none"> <li>• Git</li> <li>• Terminal</li> <li>• Web browser</li> <li>• Text editor (sublime text, notepad, notepad++, vscode)</li> <li>• GitHub</li> </ul>
Equipment	<ul style="list-style-type: none"> <li>• Computer</li> <li>• Network devices</li> </ul>
Materials/ Consumables	<ul style="list-style-type: none"> <li>• Internet</li> <li>• Electricity</li> </ul>

Assessable outcomes	Assessment criteria (Based on performance criteria)	Indicator	Observation		Marks allocation
			Yes	No	
1. <b>Setup repository</b> (35%)	Git is properly initiated based on Git commands	Indicator 1. Git setup is installed			6
		Indicator 2. Git is configured			6
	Repository is properly created based on the project.	Indicator 1. GitHub account is created			5

		Indicator 2. Remote Repository is created			6
	Remote URL is properly set in accordance with Git commands	Indicator 1. Remote URL is generated			5
		Indicator 2. URL is configured			7
<b>2.Manipulate files</b> (30%)	File changes are properly added according to Git commands	Indicator1. File is created			7
		Indicator 2. File status is checked			6
		Indicator 3. Untracked, modified and deleted files are added to staging area			5
	File changes are properly committed based on added files	Indicator 1. file is committed to local repository.			5
	Branches are properly managed based on the project	Indicator 1. Branch is created			4
		Indicator 2. Branch is switched			3

1. <b>Ship codes</b>  (35%)	1. Files are properly fetched in accordance with Git instruction	Indicator 1. Pull is done			8
	2. Files are properly pushed to remote branch based on committed files	Indicator 1. Push is done			8
	3. Branches are properly merged based on pull request created	Indicator 1. Pull request is created			9
		Indicator 2. Branch are merge			10
Total marks		100			
Percentage Weightage		100%			
Minimum Passing line % (Aggregate): 70%					

## Learning Unit 1: Setup repository

### Learning Outcomes 1.1 Git is introduced based on version control

- Definition of general key terms
  - ✓ Version control
  - ✓ Git
  - ✓ GitHub
  - ✓ Terminal

A repository in version control is computer storage for maintaining data or software packages. This location contains files, databases, or information organized for quick access over a network or directly. A repo allows consolidating data with a version control system to store metadata for every file and log changes.

### Version control

also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time.

### version control and why is it important

Version control software facilitates coordination, sharing, and collaboration across the entire software development team. It enables teams to work in distributed and asynchronous environments, manage changes and versions of code and artifacts, and resolve merge conflicts and related anomalies.

### Benefits of Version Control

The Version Control System helps manage the source code for the software team by keeping track of all the code modifications. It also protects the source code from any unintended human error and consequences.

**Git** is a version control system that lets you manage and keep track of your source code history. **GitHub** is a cloud-based hosting service that lets you manage Git repositories. If you have open-source projects that use Git, then GitHub is designed to help you better manage them.

### Git a version control software?

Git is an open source distributed version control system that helps software teams create projects of all sizes with efficiency, speed

Git is a distributed version control system that enables software development teams to have multiple local copies of the project's codebase independent of each other.

Can GitHub be used for version control?

### **GitHub a version control software**

is a web-based hosting service for Git repositories which allows you to create a remote copy of your local version-controlled project. This can be used as a backup or archive of your project or make it accessible to you and to your colleagues so you can work collaboratively.

Today, **GitHub** is one of the most popular resources for developers to share code and work on projects together. It's free, easy to use, and has become central in the movement toward open-source software.

### **GitHub best used for**

**GitHub** is an online software development platform. It's used for storing, tracking, and collaborating on software projects. It makes it easy for developers to share code files and collaborate with fellow developers on open-source projects.

### **terminal in GitHub**

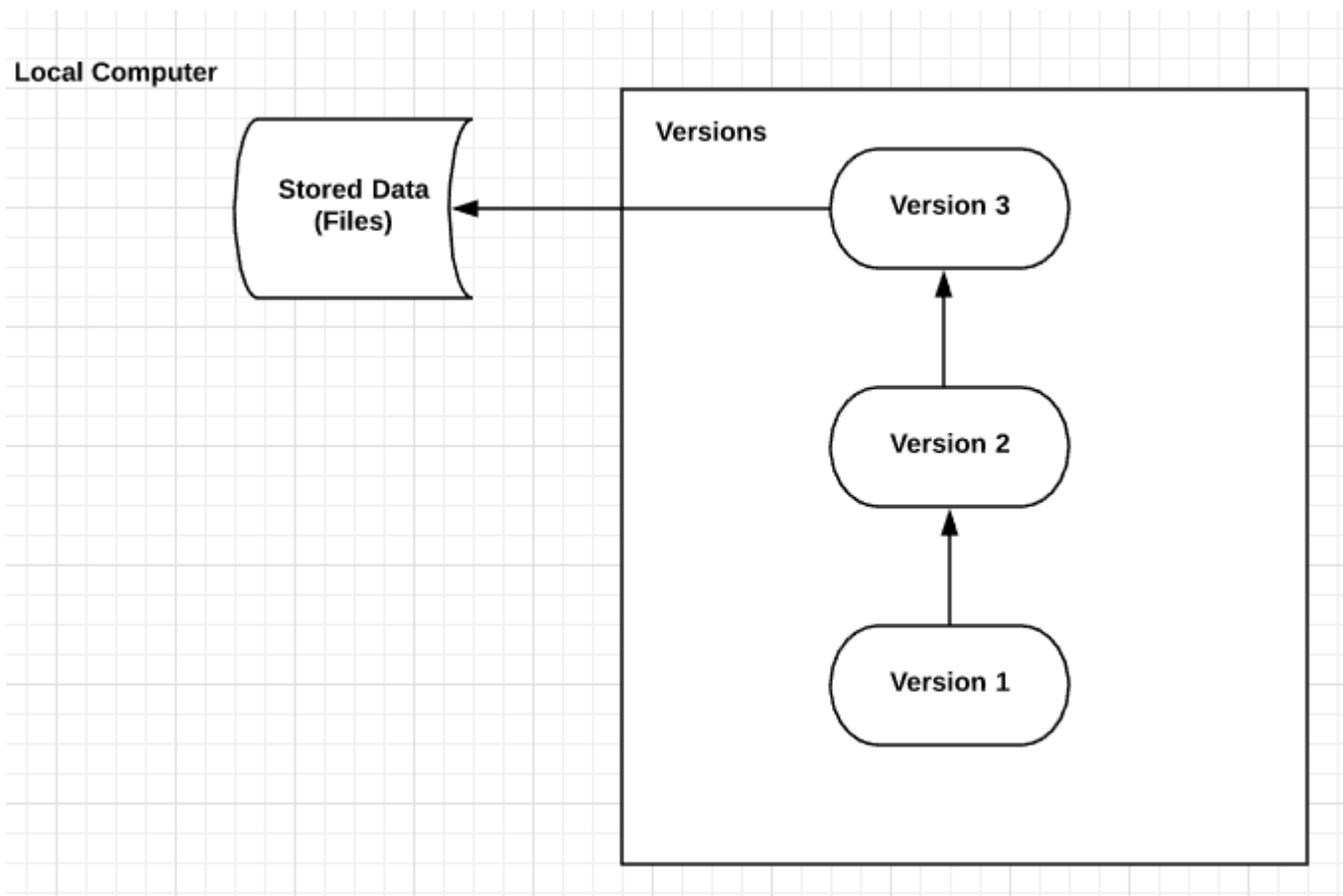
GitHub CLI is a command-line tool that brings pull requests, issues, GitHub Actions, and other GitHub features to your terminal, so you can do all your work in one place. GitHub CLI is an open-source tool for using GitHub from your computer's command line.

**A terminal** in version control is an electronic communication hardware device that handles the input and display of data. A terminal may be a PC or workstation connected to a network, Voice over Internet Protocol (VOIP) network endpoint, mobile data terminal

### **The various types of the version control systems are:**

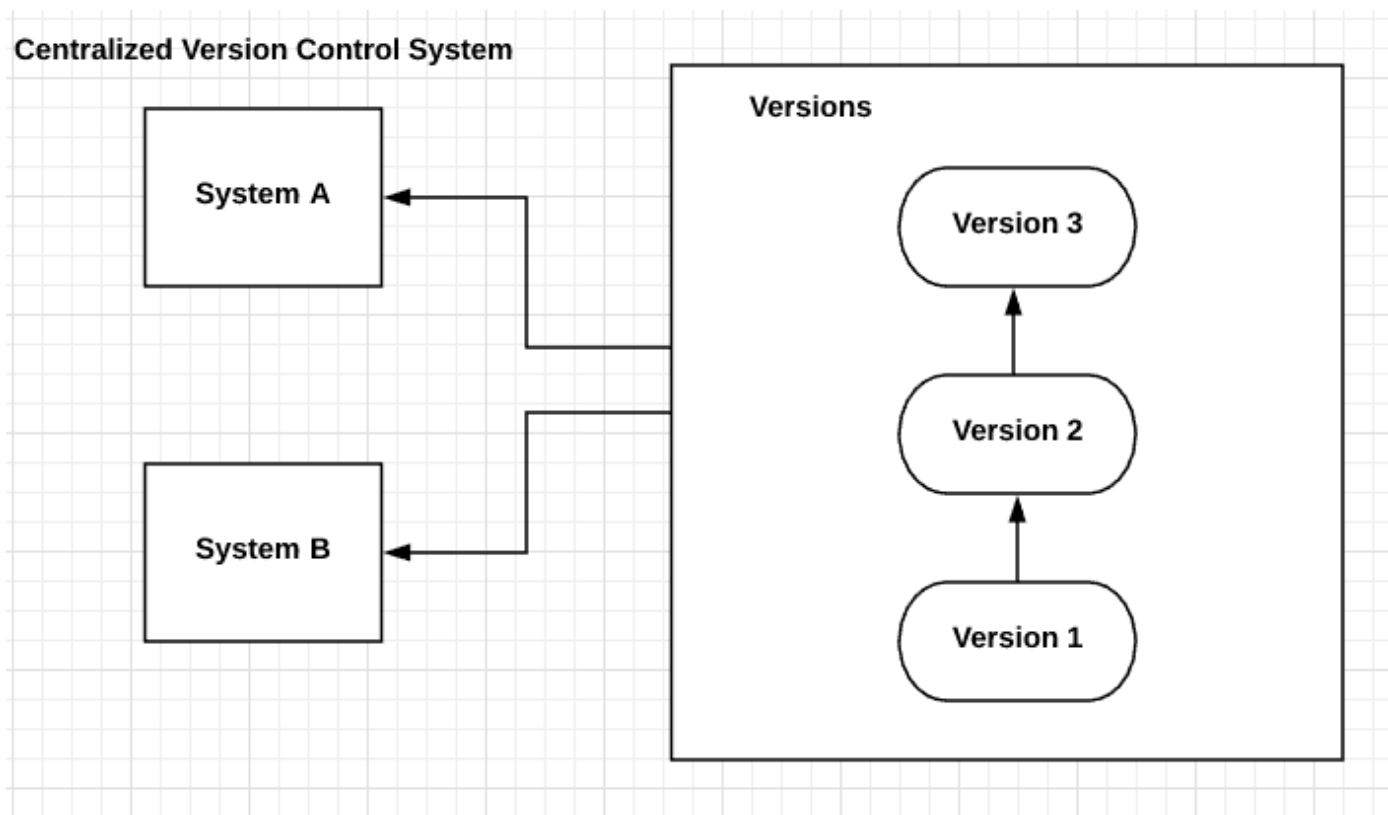
1. Local Version Control System
2. Centralized Version Control System
3. Distributed Version Control System

## 1. Local Version Control System:



Local version control system maintains track of files within the local system. This approach is very common and simple. This type is also error prone which means the chances of accidentally writing to the wrong file is higher.

## 2. Centralized Version Control Systems



In this approach, all the changes in the files are tracked under the centralized server. The centralized server includes all the information of versioned files, and list of clients that check out files from that central place.

Example: Tortoise SVN

## 3. Distributed Version Control System:

Distributed version control systems come into picture to overcome the drawback of centralized version control system. The clients completely clone the repository including its full history. If any server dies, any of the client repositories can be copied on to the server which help restore the server.

Every clone is considered as a full backup of all the data.

Example: Git



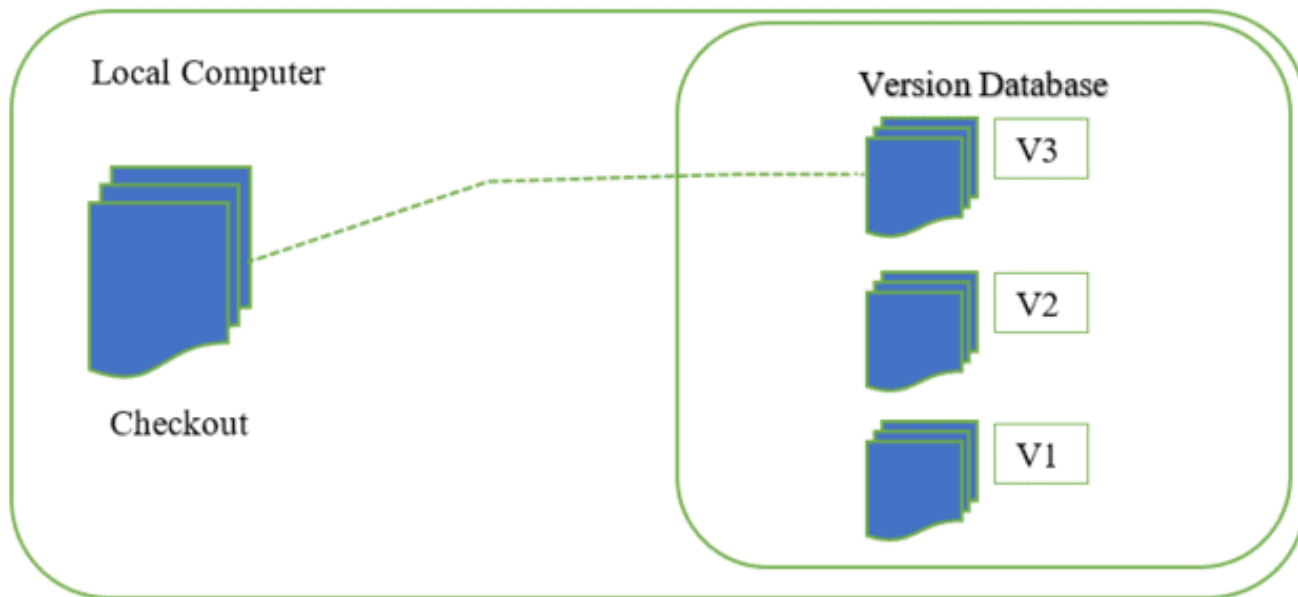
## Git

Git is a Version Control System . VCS is basically software designed to record changes within one or more files over time. It allows us to undo or to cancel all made or pending changes within one or more files. If we're working on a project with many files, VCS enables us to control the whole project. If necessary, this allows us to revert one or more files any of their previous versions or the whole project to a previous version. We can also compare changes to one file between two versions in order to see exactly what was changed in each file, when it was changed and who made the change. We can also see why the change was made.

The types of VCS are:

- Local Version Control System
- Centralized Version Control System
- Distributed Version Control System

### *Local Version Control System*

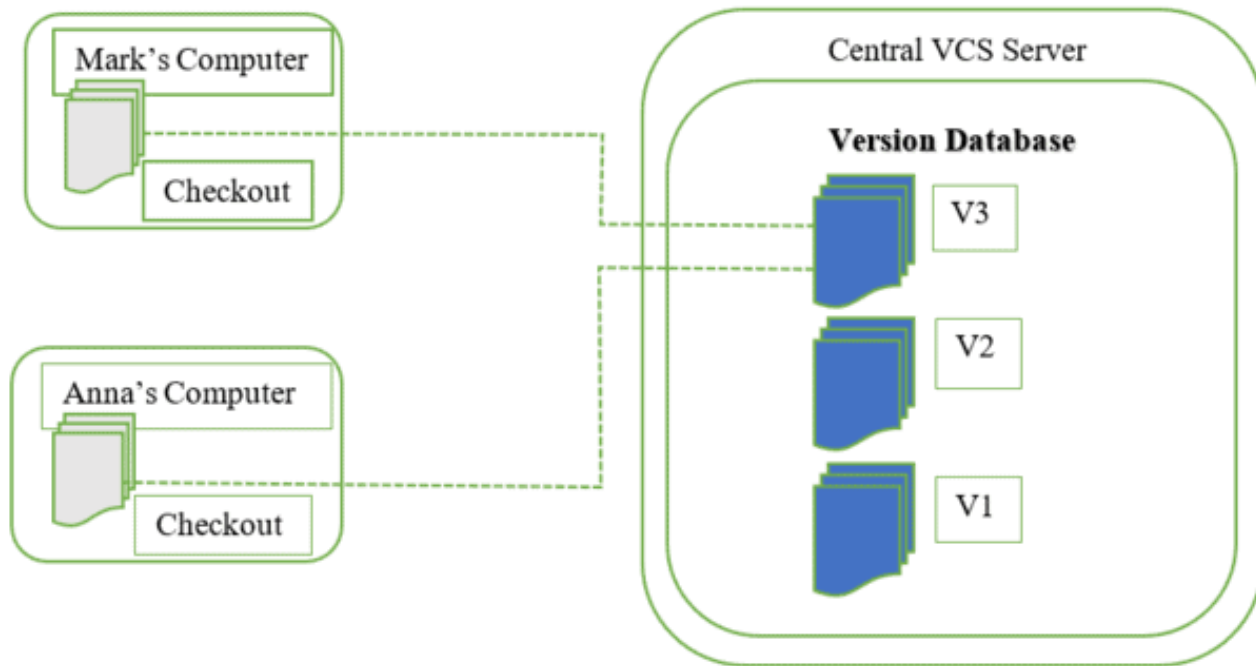


**A local version control system** is a local database located on your local computer, in which every file change is stored as a patch. Every patch set contains only the changes made to the file since its last version. In order to see what the file looked like at any given moment, it is necessary to add up all the relevant patches to the file in order until that given moment.

The main problem with this is that everything is stored locally. If anything were to happen to the local database, all the patches would be lost. If anything were to happen to a single version, all the changes made after that version would be lost.

Also, collaborating with other developers or a team is very hard or nearly impossible.

### **Centralized Version Control System**

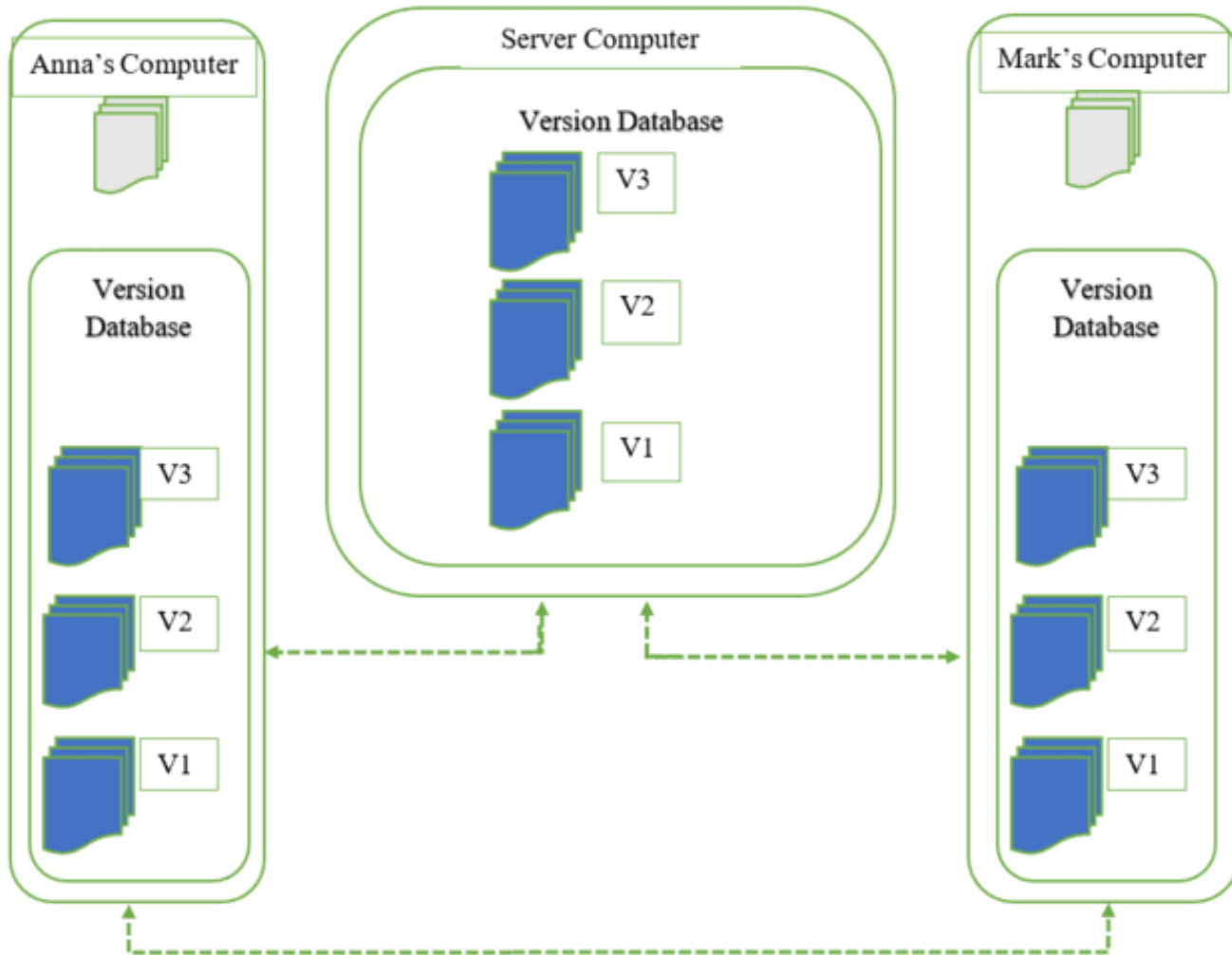


**A centralized version control system** has a single server that contains all the file versions. This enables multiple clients to simultaneously access files on the server, pull them to their local computer or push them onto the server from their local computer. This way, everyone usually knows what everyone else on the project is doing. Administrators have control over who can do what.

This allows for easy collaboration with other developers or a team.

The biggest issue with this structure is that everything is stored on the centralized server. If something were to happen to that server, nobody can save their versioned changes, pull files or collaborate at all. Similar to Local Version Control, if the central database became corrupted, and backups haven't been kept, you lose the entire history of the project except whatever single snapshots people happen to have on their local machines.

## ***Distributed Version Control System***



With distributed version control systems, clients don't just check out the latest snapshot of the files from the server, they fully mirror the repository, including its full history. Thus, everyone collaborating on a project owns a local copy of the whole project, i.e. owns their own local database with their own complete history. With this model, if the server becomes unavailable or dies, any of the client repositories can send a copy of the project's version to any other client or back onto the server when it becomes available. It is enough that one client contains a correct copy which can then easily be further distributed.

**Concurrent Versions System (CVS)** is a program that lets a code developer save and retrieve different development versions of source code. It also lets a team of developers share control of different versions of files in a common repository of files. This kind of program is sometimes known as a version control system

## Subversion

SVN stands for Subversion SVN and Subversion are the same. SVN is used to manage and track changes to code and assets across projects

Mercurial used for

**Mercurial** is a free, distributed version control system. It's also referred to as a revision control system or Mercurial source control. It is used by software development teams to manage and track changes across projects.

## Benefits of Version Control

The Version Control System helps manage the source code for the software team by keeping track of all the code modifications. It also protects the source code from any unintended human error and consequences.

## Description of git

Git is a version control system that developers use all over the world. It **helps you track different versions of your code and collaborate with other developers.**

Version control **enables multiple people to simultaneously work on a single project.**

Git is used to tracking changes in the source code.

The distributed version control tool is used for source code management.

It allows multiple developers to work together.

a project repository?

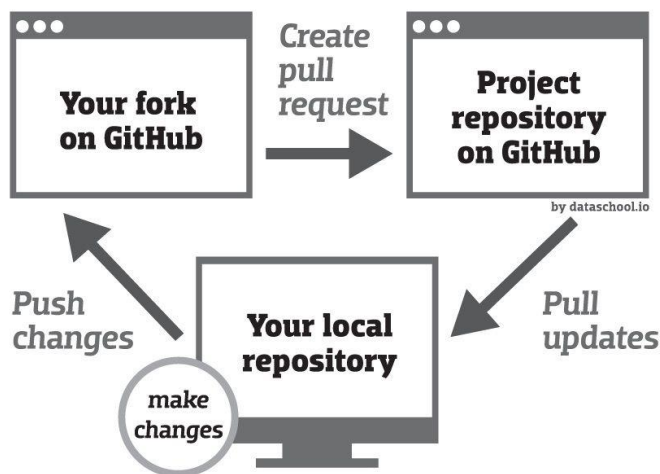
## Learning outcomes: 1.3 Repository is properly created based on the project.

A repository **contains all of the project files (including documentation), and stores each file's revision history.** Repositories can have multiple collaborators and can be either public or private. A Project as documented on GitHub: Project boards on GitHub help you organize and prioritize your work.

### purpose of repository?

Basically, a repository **allows you to populate data in memory that comes from the database in the form of the domain entities.**

The project repository **stores all files to be used by all the projects in the system**



**Learning outcomes: 1.4 Remote URL is properly set in accordance with Git commands**  
**remote in version control**

A remote in Git is a common repository that all team members use to exchange their changes. In most cases, such a remote repository is stored on a code hosting service like GitHub or on an internal server. In contrast to a local repository, a remote typically does not provide a file tree of the project's current state.

## **LEARNING UNIT II : MANIPULATE FILES**

Learning outcome 2: Manipulate files

By the end of the learning outcome, the trainees will be able to:

**2.1 Add file change to git staging area**

**2.2 Commit File changes to git local repository**

**2.3 Manage branches**

**Learning Outcomes 1.1: File changes are properly added according to Git commands**

**file version control**

Version control is **the process by which different drafts and versions of a document or record are managed**. It is a tool which tracks a series of draft documents,

culminating in a final version. It provides an audit trail for the revision and update of these finalised versions.

types of files should be stored in version control?

**At least:**

- Source code files.
- Scripts and other files you need to build software.
- Text formatted documentation, such as README and LICENSE files.
- Tool configuration files, such as clang-format .
- All other text files that your project needs.

**file management?**

File management refers to the structures included in a computer's software that help you store and arrange your electronic data. Computer software with a file management system allows you to create, work with and organize files like pictures, videos, spreadsheets and word processing documents.

**Add, edit, and commit to source files**

When you work on a Bitbucket Cloud repository, you'll need to know how to add new files and edit existing files. From there, you'll commit your changes to the repository, making it possible for you (or anyone else) to refer to that point in the repository.

## **Add and commit with Git**

For a quick reference, here's a few git commands you'll use to work on files in your local repository.

Action	Git command
Add all new files.	<code>git add --all</code>
Remove a file.	<code>git rm &lt;filename&gt;</code>
Commit changes.	<code>git commit -m '&lt;commit_message&gt;'</code>
Get an idea of the git command to use next.	<code>git status</code>

See the Git documentation for more information.

To add and commit files to a Git repository

1. Create your new files or edit existing files in your local project directory.
2. Enter `git add --all` at the command line prompt in your local project directory to add the files or changes to the repository. Enter `git status` to see the changes to be committed.

the best way to prepare files for a commit?

## **Stage Files to Prepare for Commit**

1. Enter one of the following commands, depending on what you want to do: Stage all files: `git add .` Stage a file: `git add example. html` (replace example. ...

2. Check the status again by entering the following command: `git status`.
3. You should see there are changes ready to be committed.

What are three ways to help keep your files organized?

## 5 Ways to Better Organize Your Files

- Naming is the key: Create Perfect Folder Names. ...
- Let them be where they belong: Move Files. ...
- Assign keywords, search quicker: Add Tags. ...
- Organized the hierarchy, lesser the clutter: Create Nested Folders. ...
- Locate files easily: Mark as Favorites.

### Indicative content 2.1: Add file change to git staging area



Summary for the trainer related to the indicative content

### ***Definition of general key terms***

#### **1. Status**

The `git status` command **displays the state of the working directory and the staging area**. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git. Status output does not show you any information regarding the committed project history.



## 2. *Branch*

The git branch command **lets you create, list, rename, and delete branches**. It doesn't let you switch between branches or put a forked history back together again. For this reason, git branch is tightly integrated with the git checkout and git merge commands.

## 3. **Commit**

The "commit" command is **used to save your changes to the local repository**. Note that you have to explicitly tell Git which changes you want to include in a commit before running the "git commit" command. This means that a file won't be automatically included in the next commit just because it was changed.

***Add file change to git staging area***

## Operation on git status command

### View new untracked file

When working with new folders, git does not show the contents of those folders by default when you run git status.

To view untracked files run **git status**

### Example that show the case

```
Trainer@DESKTOP-2S6N6K9 MINGW64 ~/L3SwDB (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   FormStudent.html
```

**Figure 1** Display untracked files by using Git Status

From that image before, there is no untracked files

Let us create a file called **“studentregistration.html”** inside our repository by using **touch File name**

Then after run git status and see what happen

```
Trainer@DESKTOP-2S6N6K9 MINGW64 ~/L3SwDB (master)
$ touch studentregistration.html

Trainer@DESKTOP-2S6N6K9 MINGW64 ~/L3SwDB (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   FormStudent.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        studentregistration.html
```

**Figure 2** Create a file and check that is untracked by using Git Status

For that case it displays studentregistration.html is under untracked because it have been created but not staged means that it is under working stage.

### View modified file

All modified files or changed files are viewed by using **Git status**

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   FormStudent.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    studentregistration.html
```

**Figure 3 View modified files**

### View deleted file

Remember that to delete a file you use **git rm <filename>** once you have deleted a file in git you can view them by using **Git status**.

```
$ git rm third.txt
rm 'third.txt'

Bucky@BUCKY-PC ~/Desktop/Tuna (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:   third.txt
```

## Operation on git add command

### Stage all files

Make sure that your git is configured to your username and email for staging a file

```
Trainer@DESKTOP-2S6N6K9 MINGW64 ~/L3SWDB (master)
$ git config --global user.name "Peacemaker1988"

Trainer@DESKTOP-2S6N6K9 MINGW64 ~/L3SWDB (master)
$ git config --global user.email "jeandelapaix2013@gmail.com"

Trainer@DESKTOP-2S6N6K9 MINGW64 ~/L3SWDB (master)
$
```

```
Trainer@DESKTOP-2S6N6K9 MINGW64 ~/L3SWDB (master)
$ git config --global user.name "Peacemaker1988"

Trainer@DESKTOP-2S6N6K9 MINGW64 ~/L3SWDB (master)
$ git config --global user.email "jeandelapaix2013@gmail.com"

Trainer@DESKTOP-2S6N6K9 MINGW64 ~/L3SWDB (master)
$ git config --list
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.required=true
filter.lfs.process=git-lfs filter-process
credential.helper=manager
core.fscache=true
core.symlinks=false
init.defaultbranch=master
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
user.name=Peacemaker1988
user.email=jeandelapaix2013@gmail.com
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
```

Once You run git **config -list** it displays all configuration of git.

To stage all files you have to run (**git add .**) in order to add all files to the staging area as mentioned in git workflow in outcome 1.

**N.B** :Don't forget to put a dot at the end of git add

### Stage a file

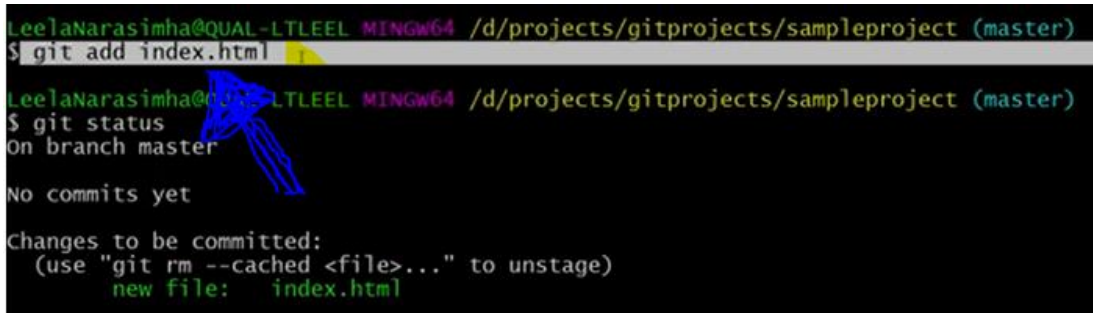
Before staging a file you have to check all existing files and unstaged files for existing files you use **ls command**.

Remember that git add is used once moving a file from workspace to staging area.

For staging a single file use `git add filename.extention`

For file extension use `.html` , `.doc` , `.js`, and others.

Example : `git add index.html` that command will add the index file to staging area as shown on that image.

A terminal window screenshot showing the execution of git commands. The prompt is 'LeelaNarasimha@QUAL-LTLEEL MINGW64 /d/projects/gitprojects/sampleproject (master)'. The first command is '\$ git add index.html'. The second command is '\$ git status', which outputs 'On branch master', 'No commits yet', and 'Changes to be committed: (use "git rm --cached <file>..." to unstage) new file: index.html'. A blue arrow points to the 'new file: index.html' line.

```
LeelaNarasimha@QUAL-LTLEEL MINGW64 /d/projects/gitprojects/sampleproject (master)
$ git add index.html

LeelaNarasimha@QUAL-LTLEEL MINGW64 /d/projects/gitprojects/sampleproject (master)
$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
   new file:   index.html
```

**Figure 4 Stage a single file**

Once you want to stage all unstaged files at the sametime you write **git add . or git add A**

### Stage folder

#### Unstage a File in Git

**In Git, unstaging a file can be done in two ways.**

- 1) `git rm --cached <file-name>`
- 2) `git reset Head <file-name>`

#### 1. Unstage Files using git `rm` command

**One of the methods to unstage git files is using the 'rm' command. It can be used in two ways:**

- 1) **On the brand new file which is not on Github.**
- 2) **On the existing file which exists on Github.**

*Case 1: rm --cached on new file which is not committed.*

`rm --cached <brand-new-file-name>` **is useful to remove only the file(s) from the staging area where this file is not available on GitHub ever. After executing this command, the file rests in the local machine, it just unstaged from the staging area.**

Example:

```
$ git add filetwo.txt
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: filetwo.txt

```
$ git rm --cached filetwo.txt
```

```
$ git status
```

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

filetwo.txt

nothing added to commit but untracked files present (use "git add" to track)

### *Case 2: rm -cached on existing file.*

**If 'rm -cached <existing-file-name>.' command is utilized on the existing file on git then this file will be considered for delete and endures as untracked on the machine. If we make a commit after this command then the file on Github will be deleted forever. We should be very careful while using this command. So, this case is not advised for unstaging a file.**

### *2. Unstage Files using git reset*

**The most effortless way to unstage files on Git is by using the “git reset” command and specify the file you want to unstage.**

`git reset <commit> -- <path>`

**By default, the commit parameter is optional: if you don't specify it, it will be referring to HEAD.**

*What does the git reset command do?*

**This command will** reset the index entries **(the ones you added to your staging area) to their state at the specified commit (or HEAD if you didn't specify any commits).**

**Also, we use the double dashes as argument disambiguation meaning that the argument that you are specifying may be related to two distinct objects: branches and directories for example.**

**As a quick example, let's pretend that you added a file named "README" to your staging area, but now you want to unstage this file.**

```
$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

```
new file: README
```

**In order** to unstage the README file, **you would execute the following command**

```
$ git reset -- README
```

**You can now check the status of your working directory again with "git status"**

```
$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

Untracked files:



(use "git add <file>..." to include in what will be committed)

README

nothing added to commit but untracked files present (use "git add" to track)

### *Unstage all files on Git*

**Previously, we have seen how you can unstage a file by specifying a path or a file to be reset.**

**In some cases, you may want** to unstage all your files **from your index.**

**To unstage all files also you can use the “git reset” command without specifying any files or paths.**

```
$ git reset
```

**Again, let’s pretend that you have created two files and one directory and that you added them to your staging area.**

```
$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

new file: directory/file

**In order to unstage all files and directories, execute “git reset” and they will be removed from the staging area back to your working directory.**

```
$ git reset
```

```
$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

README

directory/

nothing added to commit but untracked files present (use "git add" to track)

### *Remove unstaged changes on Git*

**In some cases, after unstaging files from your staging area, you may want to remove them completely.**

In order to remove unstaged changes, use the “git checkout” command and specify the paths to be removed.

```
$ git checkout -- <path>
```

**Again, let’s say that you have one file that is currently unstaged in your working directory.**

```
$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README

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

**n order to discard changes done to this unstaged file, execute the “git checkout” command and specify the filename.**

```
$ git checkout -- README
```

```
$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

**Alternatively, if you want to discard your entire working directory, head back to the root of your project and execute the following command.**

```
$ git checkout -- .
```

```
$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

### *Unstage Committed Files on Git*

**In some cases, you actually committed files to your git directory (or repository) but you want to unstage them in order to make some modifications to your commit.**

**Luckily for you, there's also a command for that.**

### *Unstage Commits Soft*

To unstage commits on Git, use the "git reset" command with the "--soft" option and specify the commit hash.

```
$ git reset --soft <commit>
```

**Alternatively, if you want to unstage your last commit, you can use the "HEAD" notation in order to revert it easily.**

```
$ git reset --soft HEAD~1
```

**Using the "--soft" argument,** changes are kept in your working directory and index.

**As a consequence, your modifications are kept, they are just not in the Git repository anymore.**

**Inspecting your repository after a soft reset would give you the following output, given that you unstaged the last commit.**

```
$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README

**What happens if you were to hard reset your commit?**

**In this case, all changes would be discarded and you would lose your changes.**

### *Unstage Commits Hard*

**To unstage commits on Git and discard all changes, use the “*git reset*” command with the “*-hard*” argument.**

```
$ git reset --hard <commit>
```

**Note: Be careful when using the reset hard command, you will lose all your changes when hard resetting.**

### *Commit File changes to git local repository*

#### **✓ *Best practice of creating a commit message***

When writing a git commit message, it's important to follow best practices so that your messages are clear and concise. Here are 7 of the best practices to follow.

### *1. Separate subject from body with a blank line*

**The subject line should be used to describe the high-level change being made, while the body can be used for a more detailed description. By separating the two with a blank line, it's easier to see at a glance what the commit is about.**

**It's also important to keep the subject line under 50 characters, so that it can be easily read in most git tools. If you need to go over 50 characters, that's fine, but anything over 72 characters should be wrapped to the next line.**

## *2. Limit the subject line to 50 characters*

**The subject line is the first thing people see when they view your commit, and it should be able to stand on its own as a brief summary of what the commit contains. If the subject line is too long, it will be truncated when viewed in tools like GitHub, making it difficult to understand what the commit is about at a glance.**

**By keeping the subject line short and sweet, you can ensure that your commit messages are easy to read and understand, which will make it easier for people to review and merge your changes.**

### *3. Capitalize the subject line*

**When git generates a log of commits, it uses the subject line to generate a summary of each commit. If the subject line is not capitalized, the summary will be difficult to read. For example, consider the following two commit messages:**

**Subject: Add new feature**

**Subject: add new feature**

The first message is much easier to read than the second. Therefore, it's important to always capitalize the subject line of your git commit messages.

### **4. Do not end the subject line with a period**

When the subject line of a git commit message is ended with a period, it can often be interpreted as an abbreviation. For example, "Fixed bug." would be interpreted as "Fixed bug with no further explanation." This can often lead to confusion for other developers who are trying to understand what was changed in that particular commit.

It's much better to write git commit messages like this:

"Fixed bug where X was not happening."

This makes it clear that the bug has been fixed and provides some context about the issue that was being faced.



### *5. Use the imperative mood in the subject line*

**When you're writing a git commit message, you're essentially writing a command that tells the code to do something. For example, "fix bug" or "add feature."**

**Using the imperative mood in your subject line makes it clear that you're giving a command, which helps keep your messages consistent and easy to understand.**

## *6. Wrap the body at 72 characters*

**When git generates a patch, it uses the first line of the commit message as the subject and the rest of the message as the body. The generated email will have a limited width, so if the body isn't wrapped, it will be very difficult to read.**

**By wrapping the body at 72 characters, you ensure that the generated email will be easy to read. This is especially important when multiple people are working on the same codebase, as they need to be able to quickly understand each other's commits.**

**It's also worth noting that some git clients will wrap the body for you automatically. However, it's still good practice to wrap it manually, as this ensures that the generated email will be readable even if the client doesn't wrap it correctly.**

## 7. Use the body to explain what and why vs. how

**When reading a commit message, I want to know two things: what changed and why. The body of the commit message should explain those two things. The subject line should be a short summary (50 characters is a good rule of thumb) that tells me what changed. The body should tell me why this change was made.**

**The reason for this is simple: the what and the why are more important than the how. The how can be easily gleaned from the diff; what's more important is understanding the motivation behind a change.**

### ✓ *Operation on git commit command*

- *Commit a file*

**git commit:** This command will record the modifications done to the files to a local repository. For simple reference, each commit has a unique ID. It's best practice to add a message with each commit explaining the changes made in a commit and [how to amend a git commit message](#). Adding a commit message helps to find a particular change or understanding the changes.

*Usage:*

```
# Adding a commit with message
$ git commit -m "Commit message in quotes"
```

*In Practice:*

```
$ git commit -m "My first commit message"
[SecretTesting 0254c3d] My first commit message
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 homepage/index.html
```

- *Edit commit message*

## How To Amend Git Commit Message | Change Git Commit Message After Push

### The Git Commit Amend Command

**This command will allow you to change files in your last commit or your commit message. Your old commit is replaced with a new commit that has its own ID.**

**The following syntax is for the amend command:**

```
git commit --amend
```

**Amending a commit does not simply change a commit. It substitutes it with a new commit which will have its own ID.**

### *Commit has not been pushed online*

**In case the commit only exists in your local repository which has not been pushed to GitHub, you can amend the commit message with the git commit --amend command:**

- Navigate to the repository that includes the commit you need to amend on the command line.
- Type git commit --amend and click on Enter
- Later, Edit the commit message and save the commit in your text editor.
  - You can add a co-author by adding a trailer to the commit.
  - You can create commits on behalf of your organization by adding a trailer to the commit.

**The new commit and message will seem on GitHub the next time you push.**

### *How to Amend the latest Git Commit Message?*

**Are you looking for the process of amending the latest Git commit message? This section will explain you clearly. In case the message to be amended is for the latest commit to the repository, then the following commands are to be performed:**

```
git commit --amend -m "New message"
```

```
git push --force repository-name branch-name
```

**Remember that using -force is not supported, as this changes the history of your repository. If you force push, people who have already cloned your repository will have to manually fix their local history.**

### *Amend Older or Multiple Git Commit Message using rebase*

**The easiest way to amend a Git commit message is to use the “git rebase” command with the “-i” option and the SHA of the commit before the one to be amended.**

**You can also choose to amend a commit message based on its position compared to HEAD.**

```
$ git rebase -i <sha_commit>
```

```
$ git rebase -i HEAD~1 (to amend the top commit)
```

```
$ git rebase -i HEAD~2 (to amend one commit before HEAD)
```

**As an example, let’s say that you have a commit in your history that you want to amend.**

**The first thing you would have to do is to identify the SHA of the commit to be amended**

```
$ git log --oneline --graph
```

```
7a9ad7f version 2 commit
```

```
98a14be Version 2 commit
```

```
53a7dcf Version 1.0 commit
```

```
0a9e448 added files
```

```
bd6903f first commit
```

**In this case, we want to modify the message for the second commit, located right after the first commit of the repository.**

**Note: In Git, you don’t need to specify the complete SHA for a commit, Git is smart enough to find the commit based on a small version of it.**

**First, run the “git rebase” command and make sure to specify the SHA for the commit located right before the one to be amended.**

**In this case, this is the first commit of the repository, having an SHA of bd6903f**

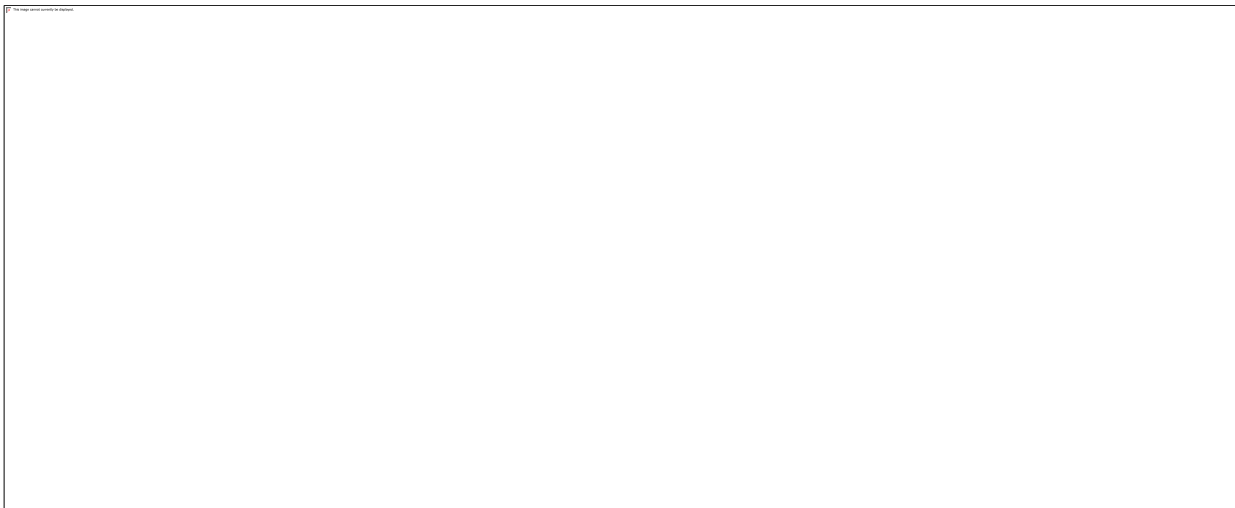
```
$ git rebase -i bd6903f
```

**From there, you should be presented with an interactive window showing the different commits of your history.**



**As you can see, every line is prefixed with the keyword “pick”.**

**Identify the commit to be modified and replace the pick keyword with the “reword” keyword.**



**Save the file and exit the current editor: by writing the “reword” option, a new editor will open for you to rename the commit message of the commit selected.**

**Write an insightful and descriptive commit message and save your changes again.**

```
This is a new commit message.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Fri Nov 29 02:35:11 2019 -0500
#
# interactive rebase in progress; onto bd6903f
# Last command done (1 command done):
#   reword 0a9e448 added files
# Next commands to do (3 remaining commands):
#   pick 53a7dcf Version 1.0 commit
#   pick 98a14be Version 2 commit
# You are currently editing a commit while rebasing branch 'master' on 'bd6903f'.
#
# Changes to be committed:
#   new file:   README
#   new file:   directory/file
#
```

**Save your changes again and your Git commit message should now be amended locally.**

```
$ git log --oneline --graph
```

```
* 0a658ea version 2 commit
* 0085d37 Version 2 commit
* 40630e3 Version 1.0 commit
* 0d07197 This is a new commit message.
* bd6903f first commit
```

**In order for the changes to be saved on the Git repository, you have to push your changes using “git push” with the “-f” option for force.**

```
$ git push -f
```

```
+ 7a9ad7f...0a658ea master -> master (forced update)
```

**That’s it! You successfully** amended the message of one of your Git commits in your repository.

### *Amend Last Git Commit Message*

**If you only want to amend the last Git commit message of your repository, there is a quicker way than having to rebase your Git history.**

**To amend the message of your last Git commit, you can simply execute the “git commit” command with the “--amend” option. You can also add the “-m” option and specify the new commit message directly.**

```
$ git commit --amend      (will open your default editor)
```

```
$ git commit --amend -m <message>
```

**As an example, let’s say that you want to amend the message of your last Git commit.**

```
$ git log --oneline --graph
```

```
* 0a658ea Last commit message
* 0085d37 Version 2 commit
* 40630e3 Version 1.0 commit
* 0d07197 This is a new commit message.
* bd6903f first commit
```

**Execute the “git commit” command and make sure to specify the “--amend” option.**

```
$ git commit --amend
```

### ✓ Operation on git log command

The git log command gives the order of the commit history for a repository. The command aids in getting the state of the current branch by showing the commits that lead to this state.

#### • To see simplified list of commit

*Usage:*

```
# Show git log with date parameters
$ git log --<after/before/since/until>=<date>

# Show git log based on commit author
```



```
$ git log --<author>="Author Name"
```

- **To see a list of commits with more detail**

```
# Show entire git log  
$ git log
```

## Manage branches

### ✓ Operations on branches

**git branch:** To discover what branch the local repository is on, add a new branch, or delete a branch.

#### Create branch

```
# Create a new branch  
$ git branch <branch_name>
```

#### List branch

```
# List all remote or local branches  
$ git branch -a
```

#### Delete local and remote branch

```
# Delete a branch  
$ git branch -d <branch_name>
```

#### *In Practice:*

```
# Create a new branch  
$ git branch new_feature
```

```
# List branches  
$ git branch -a  
* SecretTesting  
new_feature  
remotes/origin/stable  
remotes/origin/staging  
remotes/origin/master -> origin/SecretTesting
```

```
# Delete a branch  
$ git branch -d new_feature  
Deleted branch new_feature (was 0254c3d).
```

## Switch branch

**git checkout:** By using git checkout, you can easily switch branches, whenever the work is to be started on a different branch. The command works on three separate entities: files, commits, and branches.

*Usage:*

```
# Checkout an existing branch
$ git checkout <branch_name>

# Checkout and create a new branch with that name
$ git checkout -b <new_branch>
```

*In Practice:*

```
# Switching to branch 'new_feature'
$ git checkout new_feature
Switched to branch 'new_feature'

# Creating and switching to branch 'staging'
$ git checkout -b staging
Switched to a new branch 'staging'
```

**git merge:** In order to integrate the branches together, we use the git merge command. Also, it combines the changes from one branch to another branch. For instance, it is utilized to merge the changes in the staging branch to the stable branch.

*Usage:*

```
# Merge changes into current branch
$ git merge <branch_name>
```

*In Practice:*

```
# Merge changes into current branch
$ git merge new_feature
Updating 0254c3d..4c0f37c
Fast-forward
 homepage/index.html | 297
 ++++++++++++++++++++++
 1 file changed, 297 insertions(+)
 create mode 100644 homepage/index.html
```

## **Rename branch**

You can rename a local or remote Git branch by using the **-m command**. While this is not a problem for the local branch, for the remote branch you must first **delete the outdated version** and replace it with the new one.

There may be times when you need to rename a Git branch. This is because if the naming is wrong and other developers continue to work with it, you may run into problems. Fortunately, despite the tight integration and various forks, if you want to rename a Git branch, it's not a big issue. To do this, use the **-m command**. The corresponding syntax always follows the same structure:

**“git branch -m <old-name> <new-name>”.**

However, there are differences between branches that you edit locally and those that are already remote. Below we explain the steps for both cases.

### *Rename a local Git branch*

**A local Git branch exists only on your computer. You make changes and tests here without other developers noticing. Renaming it can therefore be done quickly.**

1. In the command line, select the Git branch you want to rename. The command for this is **“git checkout old-name”**.
2. You will get a confirmation that you have selected the correct branch. This will read **“Switched to branch 'old-name'”**.
3. Now perform the actual rename for the local Git branch. The appropriate command for this is: **“git branch -m new-name”**.

Alternatively, you have the option to rename the Git branch **via the master**. To do this, use the following steps:

1. Switch to the master via the command **“git checkout master”**.
2. Now enter the following command if you want to rename a Git branch: **“git branch -m old-name new-name”**.
3. To ensure that the rename was successful, retrieve the current status of the branch using the **“git branch -a”** command.

### *Renaming a remote Git branch*

**In a remote repository, you cannot simply rename a Git branch, as this would lead to complications. Instead, you need to delete the old name and then add the branch with the new name. Fortunately, this is not too hard either and can be done with a few simple commands. As with the local branch, you have two options.**

1. First, make sure the local branch has the correct, new name. The appropriate command is **“git branch -a”**.

2. Now delete the branch with the old, incorrect name from the remote repository. To do this, use the following command: **“git push origin --delete old-name”**.
3. Verify that the old branch has been deleted properly.
4. Now add the branch with the correct name. For this, use the command **“git push origin -u new-name”**.
5. Lastly, perform a **reset of the upstream branch** to ensure that the changes are effective.

However, if you want to rename the remote Git Branch with just one command, you also have the following option.

1. Enter the following command: **“git push origin :old-name new-name”**.
2. Then also perform a **reset of the upstream branch** as described above.

## **References**

- <https://www.nobledesktop.com/learn/git/stage-commit-files>
- <https://www.javatpoint.com/git-branch>
- <https://www.javatpoint.com/git-fetch>