RQF LEVEL 3

SWDVC301

SOFTWARE DEVELOPMENT

Version Control

TRAINEE'S MANUAL

October, 2024

VERSION CONTROL

KOICA Korea International Cooperation Agency

TQUM TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

Copies available from:

- *HQs: Rwanda TVET Board-RTB*
- *Web: www.rtb.gov.rw*
- **KIGALI-RWANDA**

Original published version: October 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer's and trainee's manuals for the TVET Certificate III in Software development, specifically for the module **"SWDVC301: Version Control."**

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda.

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) guiding policies and directives

Under Financial and Technical support of

COORDINATION TEAM

RWAMASIRABO Aimable
MARIA Bernadette M. Ramos
MUTIJIMA Asher Emmanuel

PRODUCTION TEAM

Authoring and Review

MUKESHIMANA Anastase

BIZIMUNGU Damien

Validation

NYABUHORO Elisabeth

HABIGENA Alexandre

KWIZERA Emmanuel

Conception, Adaptation and Editorial works

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

BIZIMANA Eric

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe
SUA Lim
SAEM Lee
SOYEON Kim
WONYEONG Jeong
HAKIZAYEZU Adrien

Financial and Technical support

KOICA through TQUM Project

TABLE OF CONTENT

ACRONYMS

CBT: Competency-Based Training
CMD: command prompt
CVS: Concurrent Version System
Git: Global information tracker
INIT: Initialization
Rm: Remove
RTB: Rwanda TVET Board
SVN: subversion
TQUM Project: TVET Quality Management Project
TVET: Technical and Vocational Education and Training
URL: Uniform resource locator

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in Software development specifically for the module of **"Version Control".** Trainees enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labor market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

MODULE CODE AND TITLE: SWDVC301 VERSION CONTROL

Learning Outcome 1: Setup repository
Learning Outcome 2: Manipulate files
Learning Outcome 3: Ship codes


GitHub
git

Indicative contents
1.1 Introduction to version control **1.2 Description of Git** **1.3 Use of GitHub repository**

Key Competencies for Learning Outcome 1: Setup Repository

Knowledge	Skills	Attitudes
• Description of version control • Description of Terminals used in version control • Description of Git • Description of GitHub	• Using CMD terminal commands for directory management • Installing Git • Preparing Git environment • Configuring .gitignore file • Creating local repository • Creating GitHub account • Creating new remote repository • Applying Git commands related to repository	• Being Problem solver • Have Team work spirit • Have critical thinking • Being self-motivation • Being Adaptability • Being Practical oriented

Duration:20 hrs

Learning outcome 1 objectives:

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

1. Describe clearly version control based on software development process
2. Use correctly terminals based on computer system available
3. Prepare properly Git environment based on Git command
4. Create properly Git Repository based on the project requirements
5. Create correctly GitHub account based on project requirement
6. Create correctly remote repository based on software development standard
7. Set properly Remote URL in accordance with Git commands.

Resources

Equipment	Tools	Materials
• Computer	• Git • GitHub • Text editor (vs code) • Terminal (CMD, Gitbash).	• Internet • Electricity

Indicative content 1.1: introduction to version control

Duration: 5 hrs

Theoretical Activity 1.1.1: Description of version control

Tasks:

1: you are requested to answer the following questions related to the Version Control:

1. What do you understand about Version Control?
2. What is Terminal in software development
3. Describe the Different types of version control systems
4. Where version control is applied
5. What are the benefits of version control?

2: Provide your answers to papers.

3: Present the findings/answers to the whole class or trainer

4: For more clarification, read the key readings 1.1.1. In addition, ask questions where necessary.

Key readings 1.1.1: Introduction to version control

1. Definition of Version control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. As the name Version Control suggests, it is a system that records changes made to a file or a set of files. The system refers to the category of software tools that make it possible for the software team to look after the source code changes whenever needed. The system records all the made changes to a file so a specific version may be rolled if required in the future.

The responsibility of the Version control system is to keep all the team members on the same page. It makes sure that everyone on the team is working on the latest version of the file and, most importantly, makes sure that all these people can work simultaneously on the same project.

The responsibility of the Version control system is to keep all the team members on the same page. It makes sure that everyone on the team is working on the latest version of the file and, most importantly, makes sure that all these people can work simultaneously on the same project.

Let's try to understand the process with the help of this diagram:

There are 3 workstations or three different developers at three other locations, and there's one repository acting as a server. The work stations are using that repository either for the process of committing or updating the tasks.

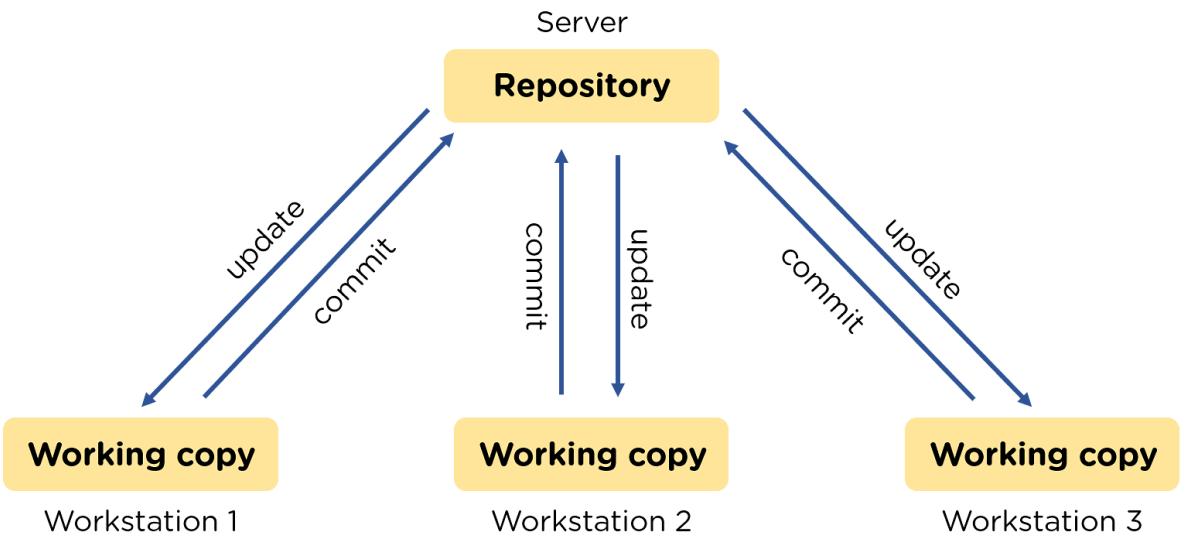


Figure 1: Example of version control

There may be a large number of workstations using a single server repository. Each workstation will have its working copy, and all these workstations will be saving their source codes into a particular server repository.
This makes it easy for any developer to access the task being done using the repository. If any specific developer's system breaks down, then the work won't stop, as there will be a copy of the source code in the central repository.

For example, if you are working on a long report or a collaborative document, Use version control to track changes to the document, review history, and revert to previous versions if necessary. Tools like Google Docs or Microsoft Word with version history features are great for this.
Secondly, Multiple developers work on the same project, each handling different features or fixes use Version Control to Create branches for each feature or fix, allowing developers to work independently without interfering with each other's code. Merge branches into the main codebase once the changes are reviewed and tested.

2. Types of version control system

2.1. Centralized version control

With centralized version control systems, you have a single “central” copy of your project on a server and commit your changes to this central copy.
A centralized version control system offers software development teams a way to collaborate using a central server. In a centralized version control system (CVCS), a server acts as the main repository which stores every version of code.
You pull the files that you need, but you never have a full copy of your project locally. Some of the most common version control systems are centralized, including Subversion (SVN) and Perforce.

Centralized version control systems have many benefits, especially over local version control systems(VCSs).

- Everyone on the system has information about the work that others are doing on the project.
- Administrators have control over other developers.
- It is easier to deal with a centralized version control system than a localized version control system.
- A local version control system facilitates a server software component that stores and manages the different versions of the files.

It also has the same drawback as in the local version control system that it also has a single point of failure.

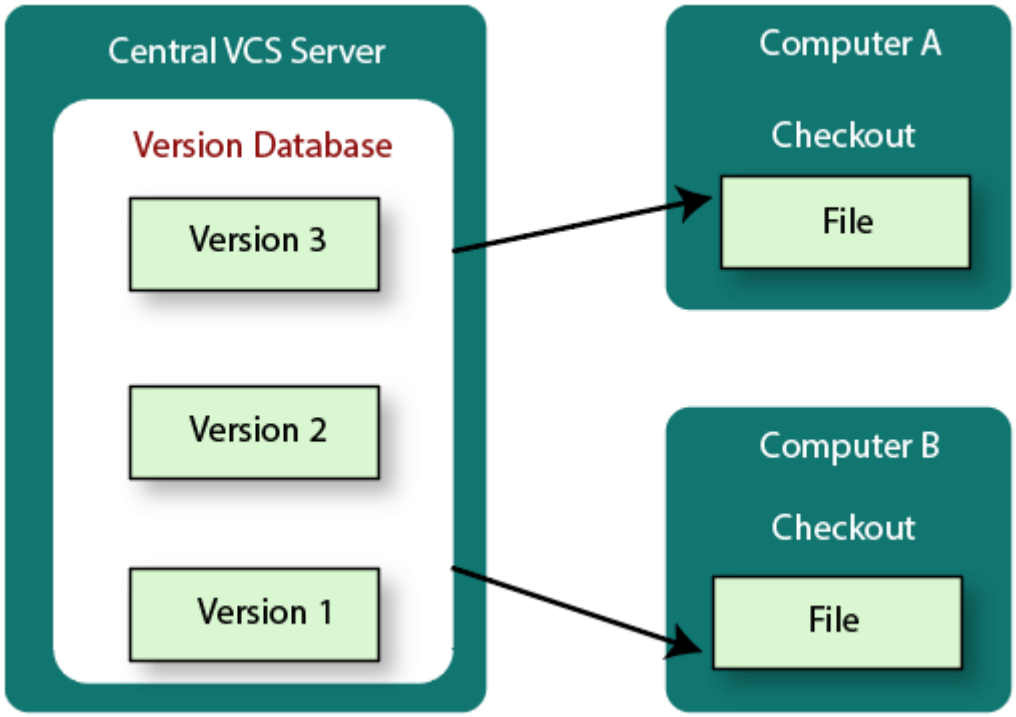


Figure 2: Centralized Version Control

Centralized Version Control System uses a central server to store all the database and team collaboration. But due to single point failure, which means the failure of the central server, developers do not prefer it. Next, the Distributed Version Control System is developed.

In centralized version control, there is a single central repository where all files and changes are stored. Developers check out files from this central repository, make changes, and then check the files back in. The central repository holds the full history of changes.

Examples:

- **SVN (Subversion):** SVN is a centralized version control system designed as a successor to CVS, offering features for version tracking, revision management, and team collaboration with a centralized repository model.
- It is A popular centralized version control system known for its simplicity and ease of use.
- **Perforce**: Often used in large enterprises and gaming industries for its performance and scalability.
- **CVS (Concurrent Version System):** CVS is an older centralized version control system that allows multiple developers to work on the same codebase concurrently, managing file versions and facilitating collaboration.

2.2. Distributed version control

In a Distributed Version Control System (such as Git, Mercurial, Bazaar, or Darcs), the user has a local copy of a repository. So, the clients don't just check out the latest snapshot of the files even they can fully mirror the repository. The local repository contains all the files and metadata present in the main repository.

With distributed version control systems (DVCS), you don't rely on a central server to store all the versions of a project's files. Instead, you clone a copy of a repository locally so that you have the full history of the project. Two common distributed version control systems are Git and Mercurial.

While you don't have to have a central repository for your files, you may want one "central" place to keep your code so that you can share and collaborate on your project with others. That's where Bitbucket comes in. Keep a copy of your code in a repository on Bitbucket so that you and your teammates can use Git or Mercurial locally and to push and pull code.
Distributed version control system allows automatic management branching and merging. It speeds up most operations except pushing and pulling. Distributed version control system enhances the ability to work offline and does not rely on a single location for backups. If any server stops and other systems were collaborating via it, then any of the client repositories could be restored by that server. Every checkout is a full backup of all the data.
In distributed version control, every user has a complete copy of the repository, including its history. This allows for offline work and more robust branching and merging capabilities. Users can work independently and then synchronize changes with others.

Examples:

- **Git**: The most widely used distributed version control system, known for its branching and merging capabilities. Git is the backbone of platforms like GitHub, GitLab, and Bitbucket.
  It is used in software development for tracking changes, enabling collaboration, and providing efficient version control.

- **Mercurial**: Another distributed version control system, similar to Git but with a different command structure and workflow.
- It provides features for tracking changes, managing revisions, and supporting collaboration among developers.

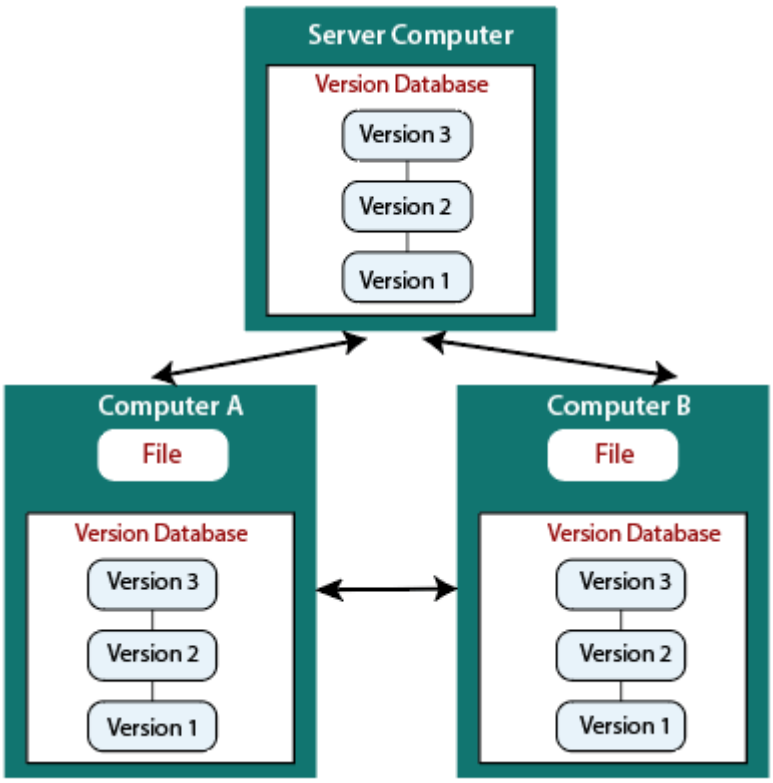


Figure 3: Distributed Version Control

2.3 Local version control

A local version control system is a local database located on your local computer, in which every file change is stored as a patch.

The localized version control method is a common approach because of its simplicity. But this approach leads to a higher chance of error. In this approach, you may forget which directory you're in and accidentally write to the wrong file or copy over files you don't want to.

To deal with this issue, programmers developed local VCSs that had a simple database. Such databases kept all the changes to files under revision control. A local version control system keeps local copies of the files.

The major drawback of Local VCS is that it has a single point of failure.

Local version control systems are simpler and store changes to files locally on a single machine. They are less suited for collaborative work but can be useful for individual developers to keep track of changes on their local files.

Examples:

- **RCS (Revision Control System)**: A simple local version control system that manages individual files and their revisions.
- **SCCS (Source Code Control System)**: An older local version control system that was commonly used for managing source code.

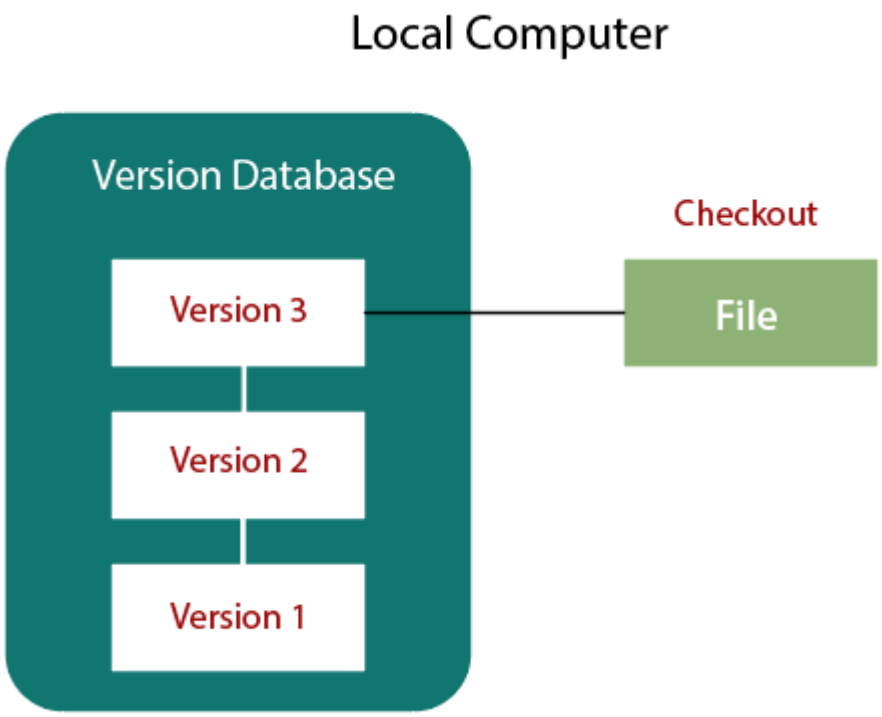


Figure 4: Local Version Control

3.Application of version control

Version control systems find applications in various industries and software development scenarios**:**

Software Development: Used extensively to manage source code, enable collaboration, and maintain code integrity throughout the development process.

Web Development: Crucial for coordinating efforts, sharing code, and managing versions in web development projects.

Mobile App Development: Essential for managing codebases across iOS and Android platforms, ensuring consistency and facilitating collaboration.

Data Science and Machine Learning: Enables reproducibility of experiments, tracks changes to code and data files, and fosters collaboration among data scientists.

Game Development: Facilitates collaboration among artists, programmers, and designers, ensuring synchronized development of game assets and code.

Documentation and Technical Writing: Helps authors track changes, collaborate with reviewers, and maintain a history of revisions for documentation projects.
Open-Source Projects: Fundamental for distributed collaboration, code sharing, and managing contributions and releases in open-source software development.
Configuration Management: Tracks changes to configuration files, infrastructure code, and deployment scripts, ensuring well-documented and reproducible configurations.
Academic Research and Collaborative Writing: Enables multiple researchers or authors to work on the same document, track changes, and maintain a clear version history.
Content Management: Applied to track and manage changes to text-based content, such as articles, blog posts, and documentation.

4.Benefits of Version control

Version control systems offer several benefits to developers and teams:
4.1 Collaboration: Version control enables seamless collaboration, allowing multiple people to work on the same project simultaneously.
4.2 Change Tracking: It tracks changes made to files, providing a complete history of modifications and facilitating code review.
4.3 Reproducibility: Developers can easily recreate previous code versions for debugging and troubleshooting purposes.
4.4 Branching and Merging: Version control supports branching, allowing the creation of separate lines of development that can later be merged.
4.5 Backup and Recovery: It serves as a reliable backup mechanism, reducing the risk of data loss and aiding in disaster recovery.
4.6 Code Reviews: Version control facilitates code review processes, improving code quality and maintaining standards.
4.7 Traceability and Accountability: Detailed logs of changes promote accountability and traceability within the development process.
4.8 Synchronization and Deployment: Version control helps manage different environments and enables efficient code synchronization and deployment.
4.9 Parallel Development: It supports parallel development by allowing multiple developers to work on different features simultaneously.
4.10 Open Source Collaboration: Version control is crucial for open-source projects, fostering collaboration among geographically dispersed developers.

5. Definition of terminal

In computing, a terminal refers to a program or a hardware device that allows users to interact with a computer system, typically through a command-line interface (CLI). The terminal provides a text-based interface where users can input commands and receive textual output from the computer.

In the context of version control, a **terminal** (also known as a command line interface or CLI) is a text-based interface used to interact with version control systems through commands. It allows users to perform version control operations, such as managing repositories, tracking changes, and collaborating with others, directly through typed commands rather than graphical interfaces.

5.1 Most popularly terminals

Here are some of the most popular terminals used across various operating systems:

1. Git Bash (Windows)

- **Overview**: Git Bash is an application for Windows that provides a Bash emulation environment. It comes bundled with Git for Windows and allows you to use Unix-style commands on Windows.
- **Importance**: Git Bash is important because it provides a familiar Bash-like environment on Windows, allowing users to run Git commands and use Unix-like tools, which is helpful for consistency across different operating systems.
- **Features**: Includes Git command-line tools, Unix commands, and Bash scripting capabilities.

2. Terminal (macOS)

- **Overview**: The built-in terminal application on macOS that supports Unix-based commands.
  The Terminal app on macOS is a command-line interface that supports Unix-based commands, including Git commands.
- **Importance**: On macOS, the Terminal provides a native environment for running Git commands and managing repositories. It's essential for macOS users who need to interact with Git and perform version control tasks directly from their system.
- **Features**: Provides a native command-line interface for running Git commands, shell scripting, and interacting with the macOS system.

3. Command Prompt (Windows)

- **Overview**: The default command-line interface on Windows, also known as cmd.exe.
  The Command Prompt (cmd.exe) is the default command-line interface on Windows. Git can be used within this environment if Git is installed and properly configured.
- **Importance**: While less powerful than Git Bash for Unix-like commands, the Command Prompt can still be used to run Git commands. It is important for users who prefer or require the native Windows command-line environment.
- **Features**: Supports basic command-line operations and can be used with Git if properly configured.

4. PowerShell (Windows)

- **Overview**: A more advanced command-line shell and scripting language for Windows. It can be used to run Git commands if Git is installed and configured.
- **Importance**: PowerShell provides more advanced scripting capabilities compared to the Command Prompt and can be used to automate Git-related tasks. It's important for users who need more powerful scripting features.
- **Features**: Provides extensive scripting capabilities, access to .NET framework, and integration with Windows management tasks.

several terminal environments can be used depending on your operating system and personal preference. Each of these terminals allows you to run Git commands and interact with Git repositories.

Why the Terminal is Important for Git:

- **Direct Access**: The terminal provides direct access to Git commands and functionalities without the need for a graphical interface.
- **Efficiency**: Commands can be executed quickly and efficiently, especially when dealing with large repositories or complex workflows.
- **Advanced Features**: The terminal allows users to access advanced Git features and configurations that may not be available in graphical user interfaces.
- **Scripting and Automation**: The terminal supports scripting and automation, making repetitive tasks and batch operations more manageable.
- **Flexibility**: Different terminals provide flexibility depending on the operating system and user preference, ensuring that users can work effectively in their preferred environment.

The terminal is a crucial tool for interacting with Git, offering power, flexibility, and efficiency for version control tasks. The choice of terminal depends on the operating system and the specific needs of the user.

6.Commands used in CMD

- **md or Mkdir command**: it is used to Create a project directory.it is command which are enabled by default, allow you to use a single **mkdir** command to create intermediate directories in a specified path.
- **dir command**: it is used to List out directory contents, to display all files and directories, including hidden ones, in wide format The "**dir**" command is useful for quickly viewing the contents of a directory and getting information about files and directories within it
- **cd command:** it is used Changing the working directory. It allows you to navigate through the file system and switch to a different directory. The **"cd"** command is essential for navigating the file system in Command Prompt and allows you to change directories to access different files and directories.

- **cd.. Command:** it is used Moving up to the parent directory. it as a shorthand way of using the "cd" command to move up one level. The **"cd.."** command is useful for quickly moving up to the parent directory without having to specify the full path.
- **rd or rmdir Command:** it is used Deleting a directory, the **"rd" or "rmdir"** command in Command Prompt (cmd) that are used to remove (delete) an empty directory. Both "**rd" and "rmdir**" are aliases of the same command and can be used interchangeably. But remember to use the appropriate directory name when executing the command.
- **del Command**: it is used Deleting a file, this command will delete all files with the ".txt" extension in the current directory. The **"del"** command permanently deletes files, and they cannot be recovered from the Recycle Bin. Remember to use the appropriate file name(s) or wildcard pattern when executing the command.
- **copy Command:** it is used Copying a file, this command used to copy one or more files from one location to another, but If the destination path is not specified, the file(s) will be copied to the current directory.so that is why is very important to Remember to use the appropriate file name(s) and destination path when executing the command.
- **move Command**: it is used to move a file, the "**move**" command in Command Prompt (cmd) is used to move one or more files or directories from one location to another. Remember to use the appropriate file name(s) or directory name(s) and destination path when executing the command.
- **ren Command**: it is used Renaming a file, the **"ren**" command in Command Prompt (cmd) is used to rename files or directories. Use the "ren" command followed by the current name of the file or directory and the new name you want to assign. For example: **ren current_name new_name:** This command will rename the file or directory with the specified "current_name" to the "new_name. If the file or directory you want to rename has spaces in its name, enclose the names in double quotation marks.
- **type Command:** it is used to View file content, the "**type**" command in Command Prompt (cmd) is used to display the contents of a text file directly. Use the "type" command followed by the name of the text file you want to display its contents. The "type" command is typically used for text files. It may not display the contents of binary files correctly. Remember to use the appropriate file name when executing the command. The "type" command is useful for quickly viewing the contents of a text file without opening it in a separate program.
- **cls Command**: it is used to clear the CMD terminal. The "**cls**" command in Command Prompt (cmd) is used to clear the contents of the command prompt window, providing a clean slate. to clear the contents of the command prompt window, simply type "cls" and press Enter. After executing the command, the command prompt window will be cleared, and you will see a fresh, empty command prompt.so

The "cls" command is useful for clearing the screen and removing the clutter of previous commands and outputs, providing a clean workspace.

- **echo command**: it is used to display a message, the "**echo**" command is useful for displaying messages, checking the value of environment variables, and controlling the echoing of command lines. The echo command can be used to display a message, variable value, or system information on the console. The command can be followed by a message or text string enclosed in double quotes. For example, echo “Hello, World!” will display the message “Hello, World!” on the console.
- **exit Command**: it is used Exiting the CMD terminal, the "**exit**" command in Command Prompt (cmd) is used to close the command prompt window. The "exit" command is useful for quickly closing the command prompt window when you're done with your tasks.

Figure 1

Practical Activity 1.1.2: Using CMD commands

Task:

1: Referring to the previous theoretical activities (1.1.1) you are requested to go to the computer lab to use cmd commands to create a directory, change directory, rename directory, delete directory etc…. This task should be done individually.
2: Launch CMD
3: Referring to the steps provided in key readings 1.1.2, use CMD commands to create a directory, change directory, Rename directory, delete directory etc…. .
4: Present your work to the trainer and whole class

Key readings 1.1.2: Applying CMD commands

1. Use of CMD Commands

1.1. **Launch CMD**

There are several ways to open the Command Prompt:

- Press Windows Key + R, type "cmd" or "cmd.exe" in the Run dialog, and press Enter.
- Press Windows Key, type "Command Prompt" in the search bar, and click on the Command Prompt app.
- Press Windows Key + X, then select "Command Prompt" or "Command Prompt (Admin)" from the Power User menu.

1.2. Understanding the command prompt:

The command prompt displays a current directory, usually starting with C:\Users\YourUsername. This is the location where commands will be executed.
You can type commands directly after the command prompt and press Enter to execute them.

Directory management using CMD commands

1. **md or mkdir:** Make Directory. This command creates a new directory.
    - Example: md MyFolder or mkdir MyFolder (Creates a new folder named "MyFolder" in the current directory)
2. **dir:** Directory Listing. This command lists the files and folders in the current directory.
    - Example: dir (Lists the files and folders in the current directory)
3. **cd:** Change Directory. This command is used to navigate between directories.
    - Example: cd C:\Users (Moves to the "Users" directory on the C drive)
4. **Cd..:** Change to Parent Directory. This command moves up one level in the directory structure.
    - Example: cd.. (Moves to the parent directory of the current directory)
5. **rd or rmdir**: Remove Directory. This command deletes a directory.
    - Example: rd MyFolder or rmdir MyFolder (Deletes the folder named "MyFolder" in the current directory)
6. **del**: Delete. This command deletes a file.
    - Example: del myfile.txt (Deletes the file "myfile.txt" in the current directory)
7. **copy:** Copy. This command copies files from one location to another.
    - Example: copy C:\Folder1\file.txt D:\Folder2\ (Copies "file.txt" from "Folder1" to "Folder" on different drives)
8. **move**: Move. This command moves files or directories to a different location.
    - Example: move C:\Folder1\file.txt C:\Folder2\ (Moves "file.txt" from "Folder1" to "Folder2" on the same drive)
9. **ren:** Rename. This command renames a file or directory.
    - Example: ren myfile.txt newfile.txt (Renames the file "myfile.txt" to "newfile.txt")
10. **type:** Display File Content. This command displays the contents of a text file.
    - Example: type myfile.txt (Displays the contents of the file "myfile.txt")
11. **cls:** Clear Screen. This command clears the CMD window.
    - Example: cls (Clears the CMD window)
12. **echo:** Display Text. This command displays text on the CMD window or writes it to a file.
    - Example: echo Hello, World! (Displays "Hello, World!" on the CMD window)
13. **exit:** Exit CMD. This command exits the CMD prompt and closes the CMD window.
    - Example: exit (Exits the CMD prompt)

These commands are commonly used for various operations in CMD, including directory management, file manipulation, and navigation.

1.3. Steps used to perform CMD Commands related to directory management

- ✓ **md or Mkdir command:**

1.Open the Command Prompt

2.Navigate to the Directory

3.Create the Directory

To perform the dir command, follow these steps:

Open the command prompt by pressing the Windows key and typing "cmd" or "command prompt" in the search bar. Then click on "Command Prompt" or press Enter.

Once the command prompt is open, type "dir" and press Enter.

- ✓ **cd Command**

1.Open Command Prompt

2.Navigate to a Directory

- Change to a Specific Directory
- Change to a Different Drive
- Change to Parent Directory
- Change to User's Home Directory
- List Available Drives

3.Press Enter

4.Verify the Change

- ➔ **Cd.. command**

To implement the cd.. command and navigate up one level in the directory hierarchy, follow these steps:

1) Open the command prompt by pressing the Windows key and typing "cmd" or "command prompt" in the search bar. Then click on "Command Prompt" or press Enter.
2) Once the command prompt is open, type cd followed by the path of the directory you want to navigate to. For example, if you want to navigate to a directory named "Folder1" located in your user directory, you would enter: cd C:\Users\Username\Folder1 and press Enter.
3) Once you are in the desired directory, you can use the cd.. command to navigate up one level at a time. Simply type cd.. and press Enter. This will move you to the parent directory of the current directory.
4) You can use cd.. multiple times to navigate up multiple levels.

- ➔ **Rd or rmdir command**

Steps to implement the rd or rmdir command:

1. Open the command prompt by pressing the Windows key and typing "cmd" or "command prompt" in the search bar. Then click on "Command Prompt" or press Enter.

2. Once the command prompt is open, navigate to the directory containing the directory you want to delete using the cd command. For example, if you want to delete a directory named "Folder1" located on your desktop, enter cd C:\Users\YourUserName\Desktop and press Enter. This will take you to your desktop directory.
3. To delete the directory "Folder1", enter rd /s Folder1 or rmdir /s Folder1 and press Enter. The /s option is used to delete the directory and all its subdirectories recursively without prompting for confirmation. If there are any files or subdirectories in the directory, you will be prompted to confirm the deletion. Type Y and press Enter to confirm.

➔ **Del command:**

1. Open the Command Prompt:
2. Navigate to the directory containing the file(s) you want to delete:
   Replace "C:\path\to\directory" with the actual path to the directory containing the file(s) you want to delete.
3. Use the "del" command followed by the file name(s) or wildcard pattern to delete the file(s).

➔ **Copy command**

1. Open the Command Prompt: Press the Windows key, type "cmd," and press Enter.
2. Navigate to the directory where the source file is located
3. Use the "copy" command followed by the source file name and the destination directory to copy the file

➔ **Move command:**

1. Open the Command Prompt: Press the Windows key, type "cmd," and press Enter.
2. Navigate to the directory where the source file or directory is located
3. 3.Use the "move" command followed by the source file or directory name and the destination directory to move the file or directory

➔ **Ren command**

1. Open the Command Prompt
2. Navigate to the directory where the file or directory you want to rename is located: Use the "cd" command to navigate to the appropriate directory if needed
3. Use the "ren" command followed by the current name of the file or directory and the new name you want to assign.

➔ **Type command**

1. Open the Command Prompt
2. Navigate to the directory where the text file is located
3. Use the "type" command followed by the name of the text file you want to display.

➔ **Cls command**

1. Open the Command Prompt
2. To clear the command window, simply type "cls" and press Enter.

Echo command:

1. Open the Command Prompt: Press the Windows key, type "cmd," and press Enter.
2. To display text in the command window, simply type "echo" followed by the text you want to display
3. to enable or disable the display of commands, you can use the "echo" command with the "on" or "off" parameter.

➔ **Exit command**

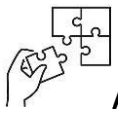
In Windows, the exit command is used to close the command prompt window or terminate a batch file.

To implement the exit command:

1. Open the command prompt by pressing the Windows key and typing "cmd" or "command prompt" in the search bar. Then click on "Command Prompt" or press Enter.
2. Once the command prompt is open, enter any commands you want to execute.
3. When you're ready to exit the command prompt, simply type exit and press Enter. This will close the command prompt window.

Points to Remember

- There are primarily three types of version control systems: Local Version Control Systems, Centralized Version Control Systems, and Distributed Version Control Systems.
- There are most popular terminals commonly used with popular version control systems include: command prompt (CMD), PowerShell, Bash
- To use the CMD (Command Prompt) terminal effectively in Windows, follow these basic steps:
  1. Open CMD
  2. Understanding the commands



Application of learning 1.1.

You are a system administrator responsible for managing directories on a network server. You requested to use the Command Prompt (CMD) to perform creation of directories, changing directories, renaming directories, delete directories for efficient and precise management of the network server's file system, ensuring optimal organization and accessibility.

Indicative content 1.2: Description of Git

Duration: 9hrs

Theoretical Activity 1.2.1: introduction of Git

Tasks:

1. In small groups, you are requested to answer the following questions related to the Git:
   i. What do you understand about git?
   ii. Provide a description of:
   - Features of git
   - Git basic concepts
   - Git architecture and git workflow.
2. Participate in group formulation
3. Present your findings to your classmates and trainer
4. For more clarification, read the key readings 1.2.1. In addition, ask questions where necessary.

Key readings 1.2.1: Description of Git

1. Definition of Git

Git stands for "Global Information Tracker". It's a free, open-source version control system that helps developers track and manage changes to source code. However, the name is derived from the system's ability to track changes to files, not just locally but globally, across a network. GIT works by keeping a record of all the changes made to a project and it allows multiple developers to work on the same project at the same time without interfering with one another. This allows for easy collaboration, especially for large, complex projects**.**

Git is a distributed version control system (VCS) that is widely used in software development. It allows multiple developers to work on a project simultaneously, keeping track of changes made to the codebase and facilitating collaboration.

2. Features of Git

Git is a distributed version control system that is widely used for tracking changes in source code during software development. Here are some key features of GIT:

1. **Distributed Version Control:** GIT is a distributed version control system, meaning that each developer has a complete copy of the entire project history on their local machine. This allows for decentralized collaboration and offline work.

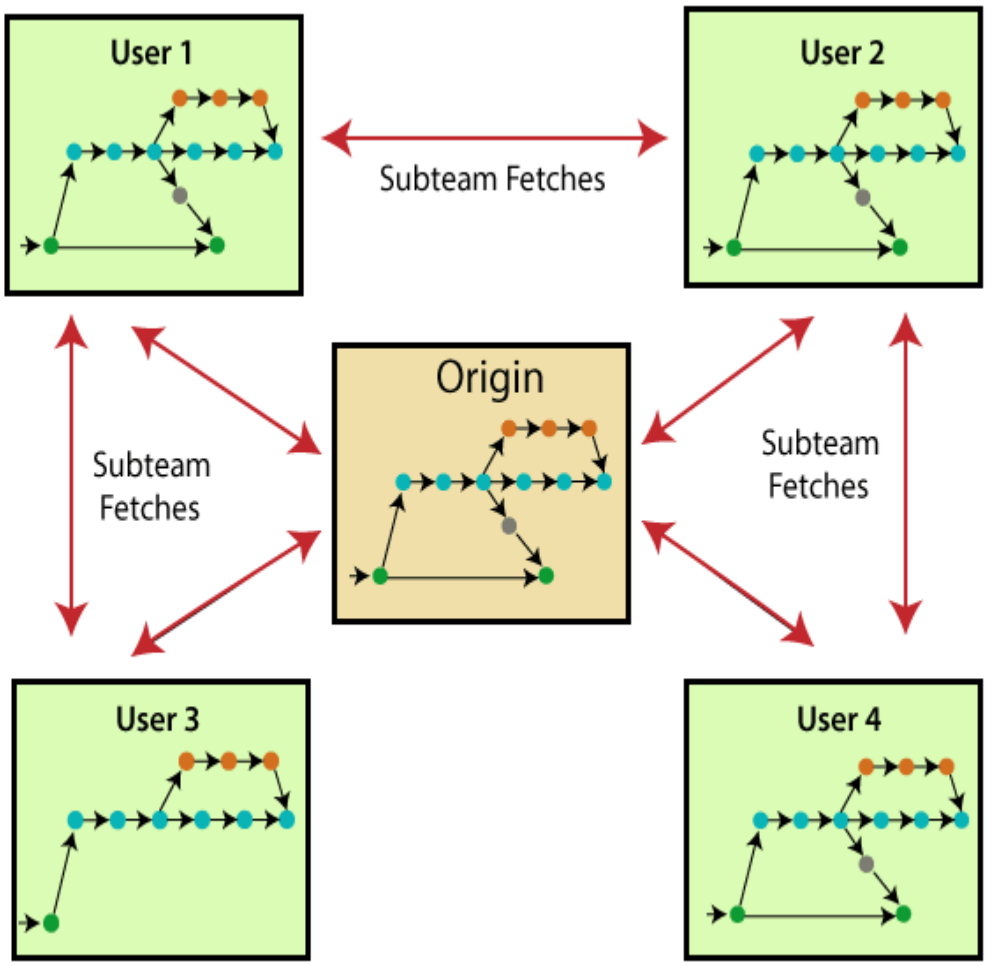


Figure 5: Distribution of Version Control to the users

2. **Branching and Merging:** GIT makes branching easy and encourages a branch-per-feature workflow. Developers can create branches to work on specific features or fixes, and later merge those branches back into the main codebase.
3. **Fast and Lightweight:** GIT is designed to be fast and efficient. It is a lightweight system that doesn't require constant communication with a central server. Most operations are performed locally, making GIT quick and responsive.
4. **Data Integrity:** GIT uses a secure hashing algorithm (SHA-1) to ensure the integrity of the versioned data. Each commit is checked, and the commit history is secured against corruption.
5. **Staging Area (Index):** GIT has a staging area, also known as the index, where changes can be selectively included before committing them. This allows developers to control which changes are included in the next commit.
6. **History Tracking:** GIT maintains a detailed history of changes to the codebase. Developers can view the history, see who made specific changes, and understand how the project has evolved.

7. **Parallel Development:** Multiple developers can work on different features simultaneously, and GIT can intelligently merge their changes. This parallel development is facilitated by GIT's branching and merging capabilities.
8. **Open Source:** GIT is an open-source project, and its source code is freely available. This openness has led to a large and active community, contributing to its widespread adoption.
9. **Compatibility:** GIT is platform-independent and works on various operating systems, including Linux, macOS, and Windows. This makes it easy for teams with diverse environments to collaborate.
10. **Support for Non-linear Development:** GIT supports non-linear development workflows, allowing for complex project structures with features like topic branches, release branches, and more.
11. **Easy Collaboration:** GIT facilitates collaboration among developers. Repositories can be hosted on platforms like GitHub, GitLab, or Bitbucket, enabling easy sharing, collaboration, and contribution from developers around the world.
12. **Integration with Other Tools:** GIT can be easily integrated with various development tools and services. Continuous Integration (CI) platforms, issue-tracking systems, and code review tools often have built-in support for GIT.

Understanding and effectively using GIT is a valuable skill for software developers, as it provides a powerful and flexible version control system for managing codebases of all sizes and complexities.

3. Git Basic concepts

3.1. Repository

In Git, the repository is like a data structure used by VCS to store metadata for a set of files and directories. It contains the collection of the files as well as the history of changes made to those files. Repository in Git is considered as your project folder. A repository has all the project-related data. Distinct projects have distinct repositories.

They are two types of git repositories:

Local Repository: it is a folder in your machine (laptop or computer) where your code is stored. A local source code or project store, means that only you have access to the code and if your computer crashes or you lose that code, then it would be pretty hard to get it back.

Remote repository: it is a folder hosted on a website, like GitHub, for example, and your code there is accessible to not just you, but your whole team! If you lose your code on your computer, your code is still safe here! This is akin to iCloud storage.

3.2. Commit

It is used to record the changes in the repository. It is the next command after the git add. Every commit contains the index data and the commit message. Every commit forms a

parent-child relationship. When we add a file in Git, it will take place in the staging area. A commit command is used to fetch updates from the staging area to the repository.
The staging and committing are co-related to each other. Staging allows us to continue making changes to the repository, and when we want to share these changes to the version control system, committing allows us to record these changes.

Commits are the snapshots of the project. Every commit is recorded in the master branch of the repository. We can recall the commits or revert it to the older version. Two different commits will never be overwritten because each commit has its own commit-id. This commit-id is a cryptographic number created by SHA (Secure Hash Algorithm) algorithm.

3.3. Branch

A branch is a version of the repository that diverges from the main working project. It is a feature available in most modern version control systems. A Git project can have more than one branch. These branches are a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug, you spawn a new branch to summarize your changes. So, it is complex to merge the unstable code with the main code base and also facilitates you to clean up your future history before merging with the main branch.

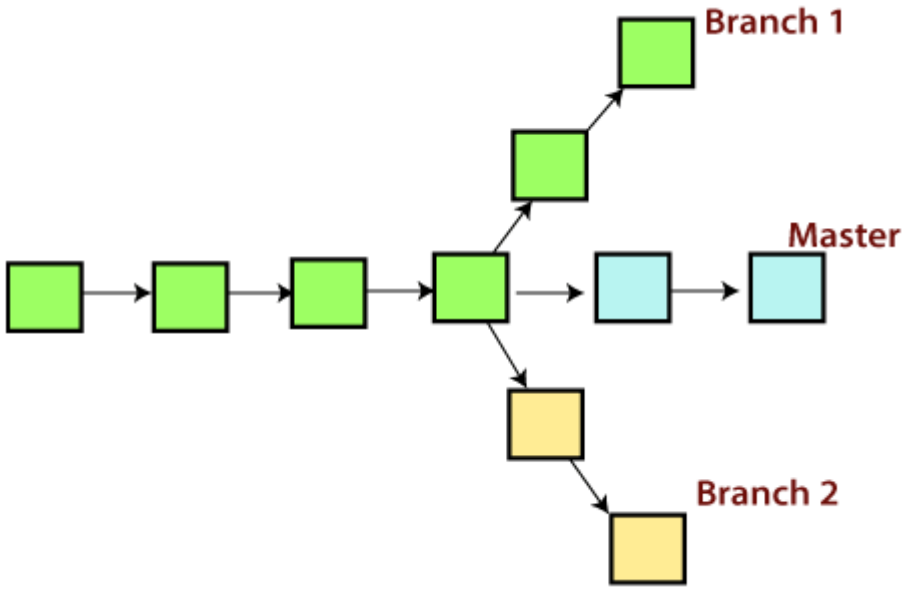


Figure 6: Branches

Git Master Branch

The master branch is a default branch in Git. It is instantiated when the first commit is made on the project. When you make the first commit, you're given a master branch to the starting commit point. When you start making a commit, then the master branch pointer automatically moves forward. A repository can have only one master branch.
Master branch is the branch in which all the changes eventually get merged back. It can be called as an official working version of your project.

3.4. Merge

Merge is the process of combining changes from one branch into another. It takes the changes made in one branch and integrates them into another branch. This is often used to incorporate feature branches back into the main branch.

In Git, the merging is a procedure to connect the forked history. It joins two or more development history together. The git merge command facilitates you to take the data created by git branch and integrate them into a single branch. Git merge will associate a series of commits into one unified history. Generally, git merge is used to combine two branches.

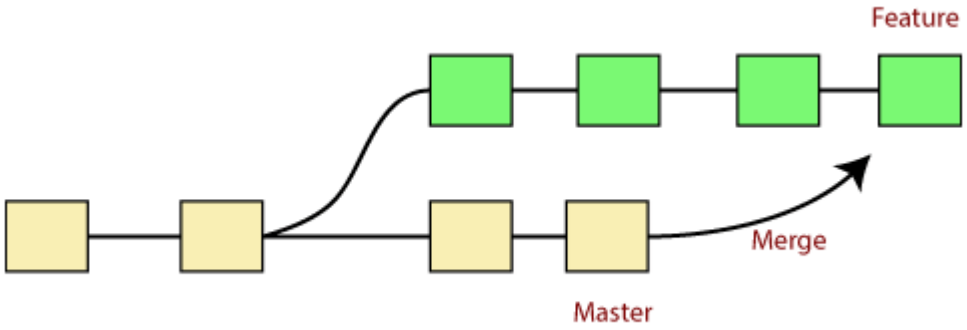


Figure 7: Merging branches

It is used to maintain distinct lines of development; at some stage, you want to merge the changes in one branch. It is essential to understand how merging works in Git.

In the above figure, there are two branches master and feature. We can see that we made some commits in both functionality and master branch, and merge them. It works as a pointer. It will find a common base commit between branches. Once Git finds a shared base commit, it will create a new "merge commit." It combines the changes of each queued merge commit sequence.

3.5. Pull

The term pull is used to receive data from GitHub. It fetches and merges changes from the remote server to your working directory. The Git pull command is used to pull a repository.

Pulling is the process of fetching the latest changes from a remote repository and merging them into your local branch. It combines the fetch (retrieving changes) and merge (incorporating changes) steps.

3.6. Push

Pushing is the process of sending your local commits to a remote repository. It updates the remote repository with your changes, making them available to other developers.

The push term refers to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repository. Pushing is capable of overwriting changes.

4. Git architecture

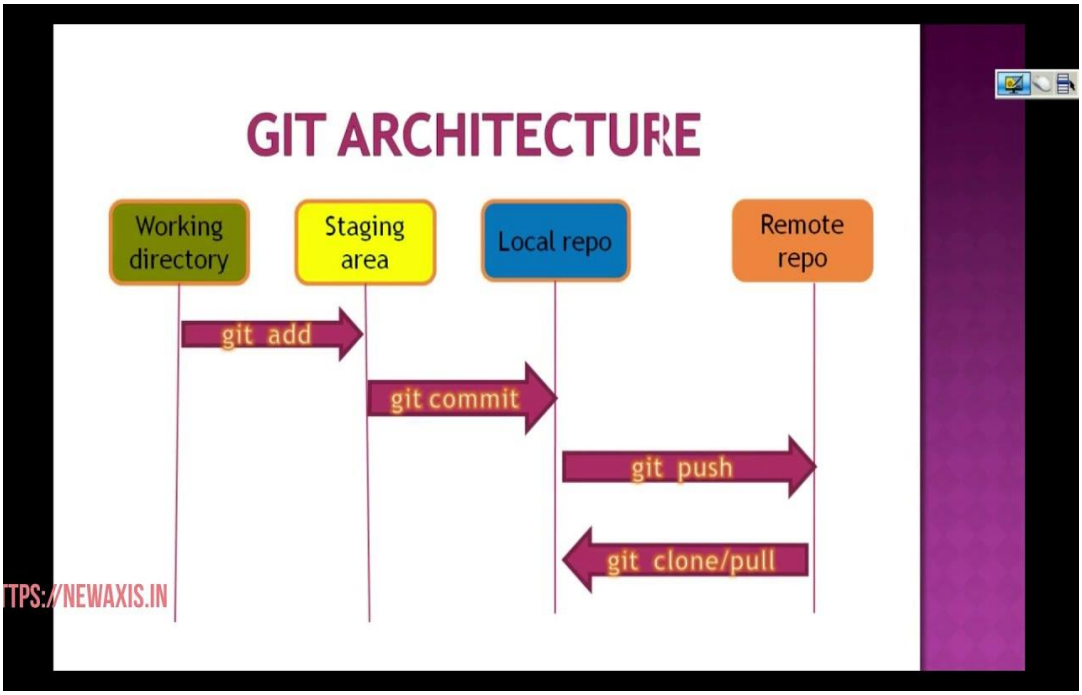


Figure 8: Git Architecture

1. **Working Directory:** The working directory is the directory on a developer's machine where the files of the project are stored. Developers modify files in the working directory as they work on the project.
2. **Index (Staging Area)**: The staging area, also known as the index, is an intermediate area where changes to files are prepared before they are committed to the repository. Developers can selectively choose which changes to include in the next commit by staging them.

3**. Local Repository:** The local repository contains a complete copy of the project's repository, including all files, directories, commit history, branches, and tags. It enables developers to access the project and its history offline.

4**. Remote Repository:** A remote is a version of the repository that is hosted on a different machine or server. It allows multiple developers to collaborate by pushing and pulling changes to and from a shared remote repository.

5. Git workflow

Git workflow refers to the specific set of practices and processes that are followed when using Git, a distributed version control system. It encompasses the way developers collaborate, manage and track changes to their codebase, and coordinate their work effectively.

1.Branch: Create a new branch to work on a specific task or feature. This allows you to isolate your changes from the main branch.
2. Edit: Make changes to files in your working directory.
3. Stage: Selectively choose the modified files you want to include in the next commit and add them to the staging area.
4. Commit: Create a new commit to record the changes in the repository. Provide a meaningful commit message that describes the changes made.
5. Push: Upload or send your local commits to the remote repository, making them available to others.
6. Merge: If you are working on a branch, you can merge your changes back into the main branch once they are tested and ready.
7. Pull: Regularly download changes from the remote repository to stay up to date with the latest changes made by other team members.

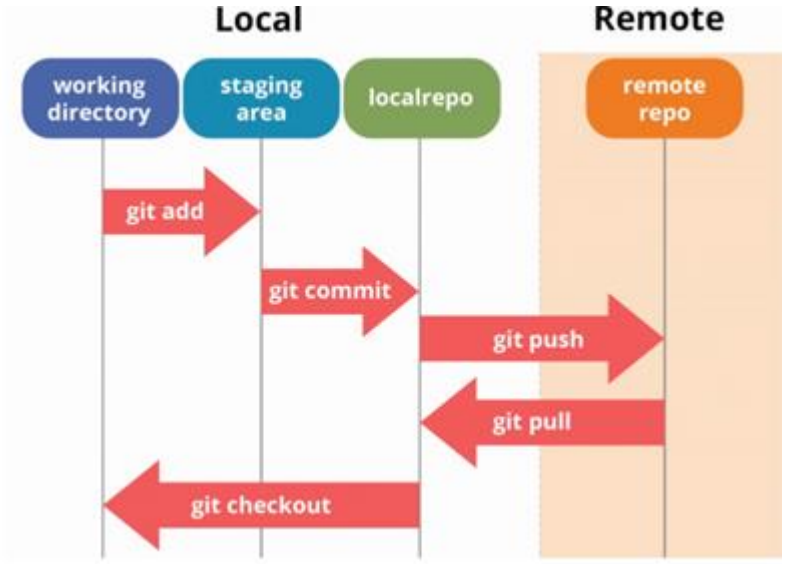


Figure 9: Git workflow simplified

6. Commands used in git configuration

6.1 The Git config commands

Git supports a command called git config that lets you get and set configuration variables that control all facets of how Git looks and operates. It is used to set Git configuration values on a global or local project level.

6.2 Git version command

In version control, "Git version" refers to the specific version or release of the Git software that is being used. Git is a distributed version control system that allows developers to track changes, collaborate on projects, and maintain a history of their code. Git follows a versioning system, where each release or version of Git has a specific number associated with it. The version number usually consists of three parts: simply we run git version for checking current version git we are already running in our computers

6.3 Git init command

It is a command used to create a new repository: Running "git init" in a directory initializes a new Git repository in that location. Git creates a hidden ". git" folder, which contains all the necessary files and subdirectories to manage version control. To create a new repository, you'll use the git init command, also git init is a one-time command you use during the initial setup of a new repository

6.4 Git ignore command

The gitignore file is a configuration file used in version control systems, specifically Git, to specify files, directories, or patterns that should be ignored and not tracked by Git. When you add files or directories to the .git ignore file, Git will exclude them from version control, meaning they will not be staged, committed, or pushed to the repository.

Practical Activity 1.2.2: Installing git set up

Task:

1: Referring to the previous theoretical activities (1.2.1) you are requested to go to the computer lab to install and configure git set up. This task should be done individually.

3: Present out the steps to install git set up.

4: Referring to the steps provided on task 3, install git set up.

5: Present your work to the trainer and whole class

6: Read key reading 1.2.2 and ask clarification where necessary

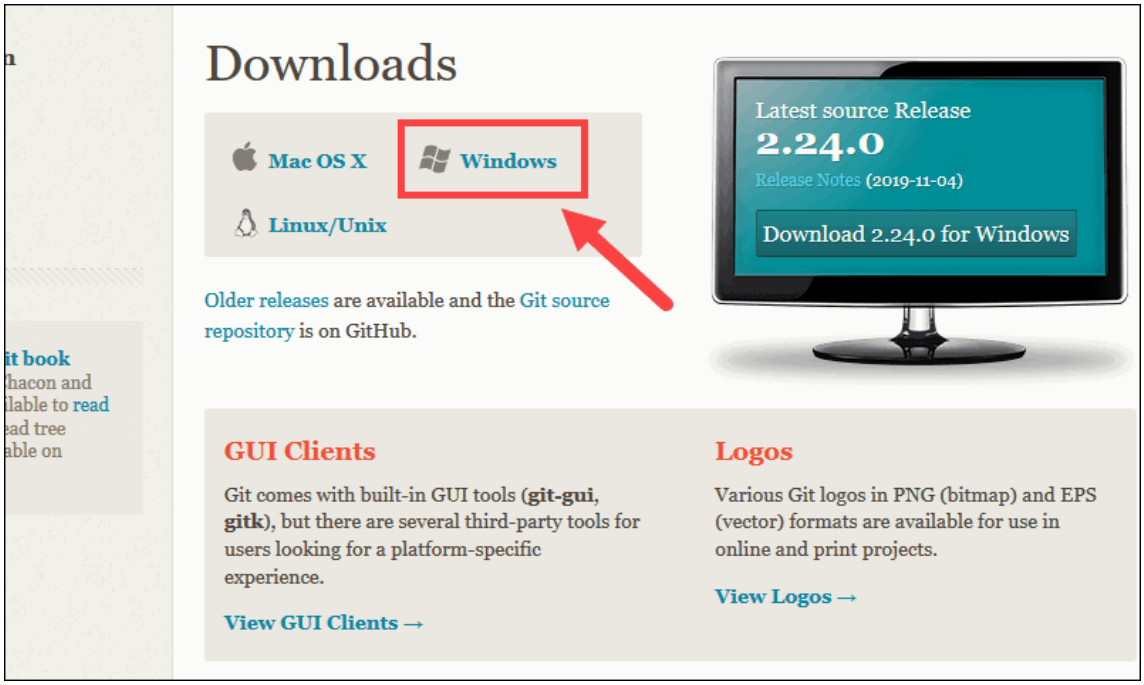
7: Perform the task provided in application of learning 1.2.

Key readings 1.2.2: Preparing Git environment

1. Steps to Install Git for Windows

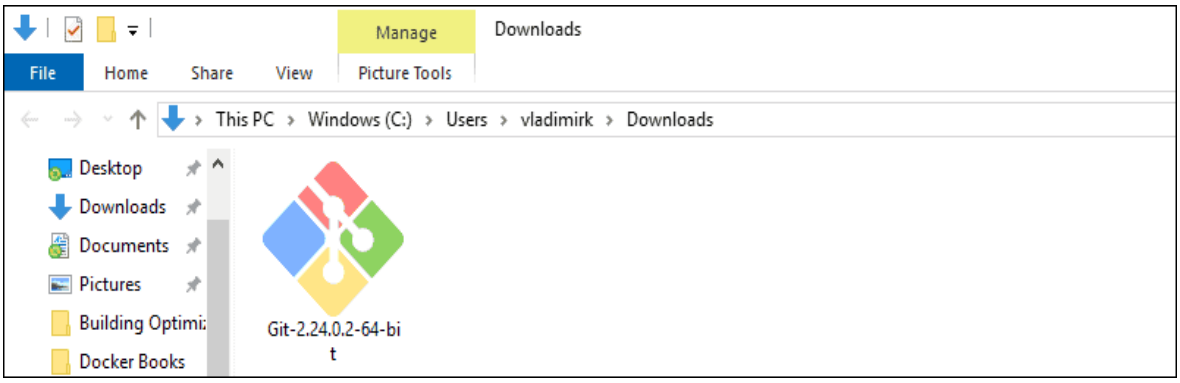
1.1 Download Git for Windows

1. Browse to the official Git website: https://git-scm.com/downloads
2. Click the download link for Windows and allow the download to complete.

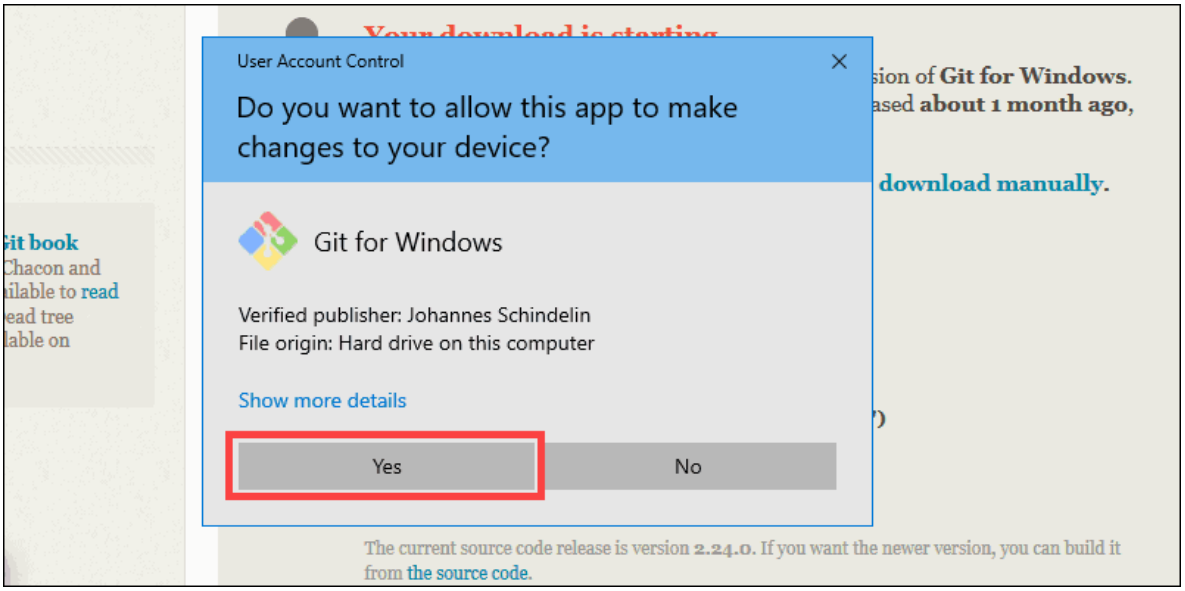


1.2 Extract and Launch Git Installer

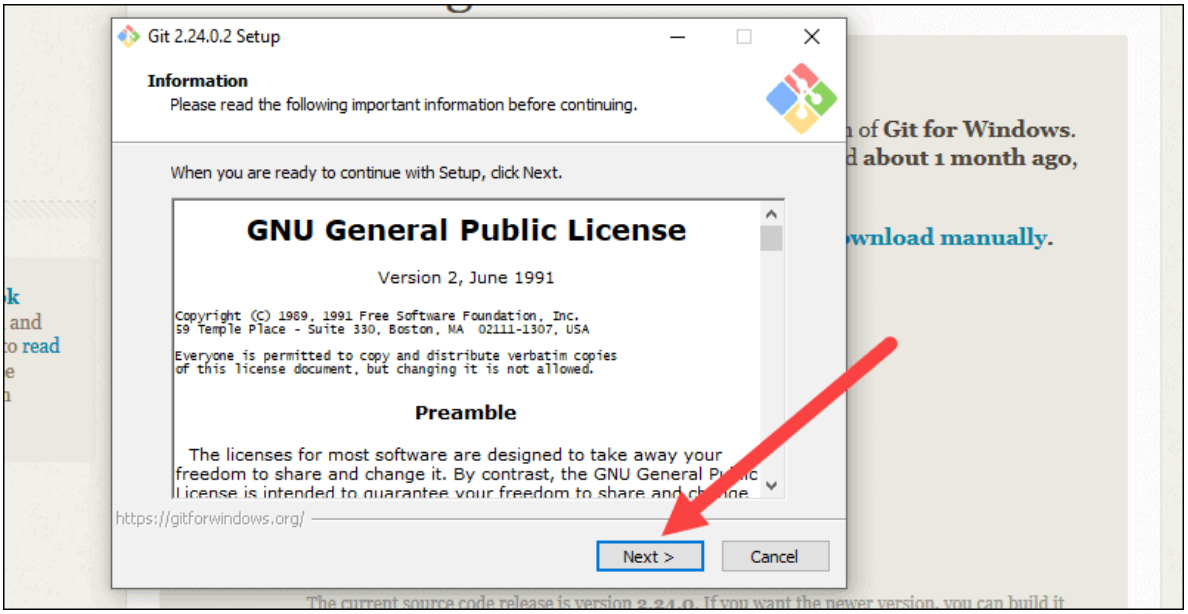
3. Browse to the download location (or use the download shortcut in your browser). Double-click the file to extract and launch the installer.



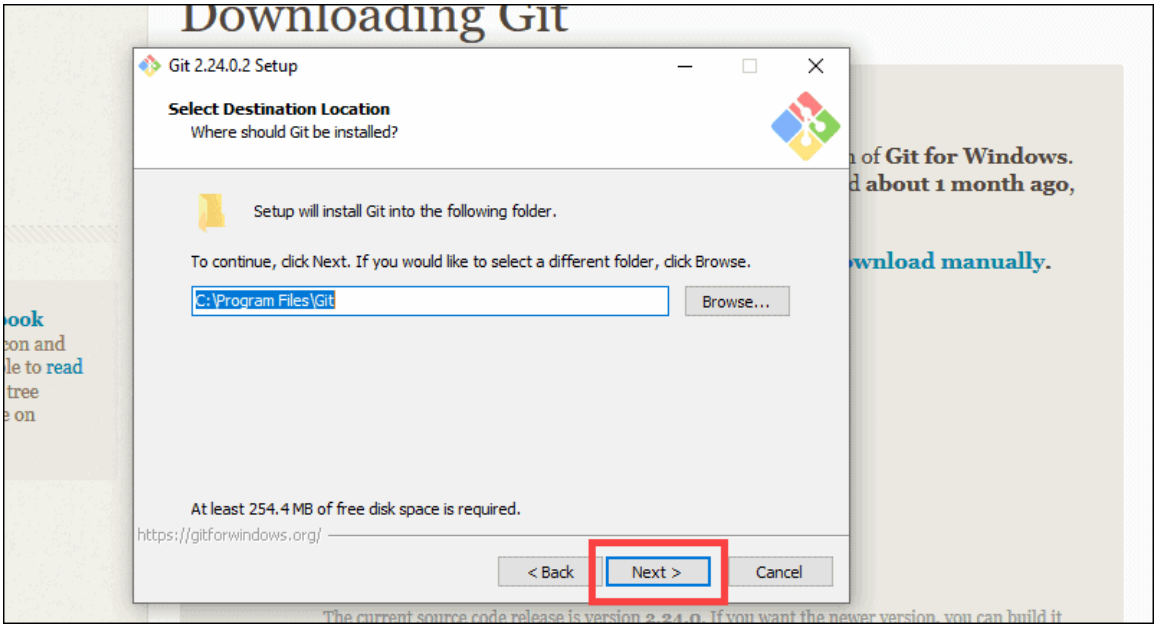
4. Allow the app to make changes to your device by clicking Yes on the User Account Control dialog that opens.



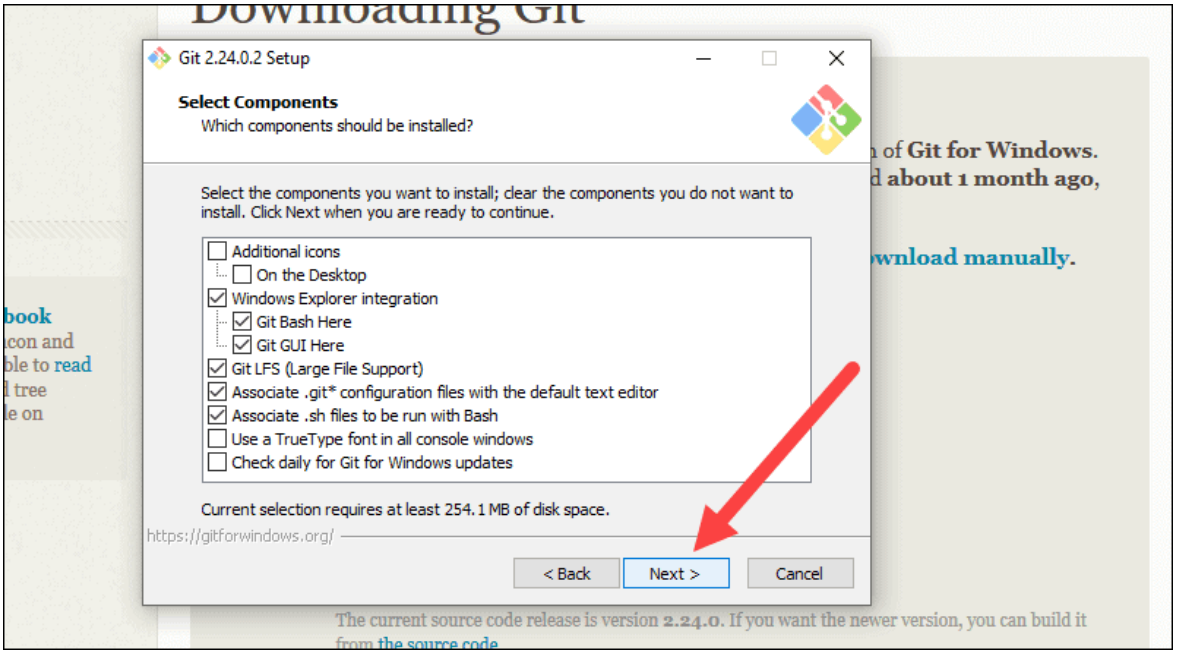
5. Review the GNU General Public License, and when you're ready to install, click Next.



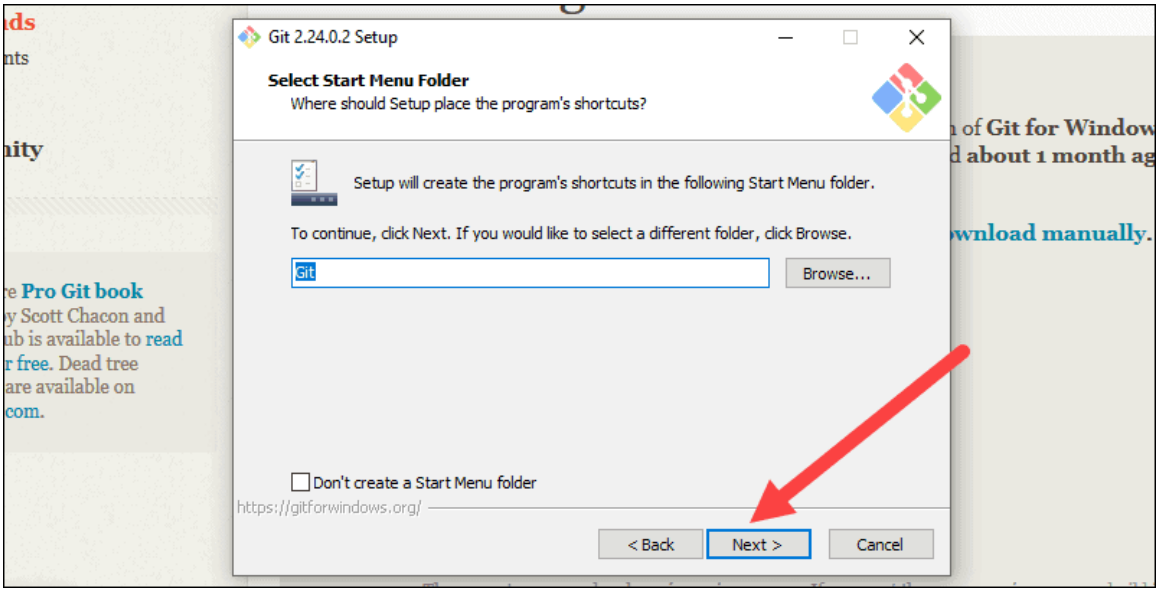
6. The installer will ask you for an installation location. Leave the default, unless you have reason to change it, and click Next.



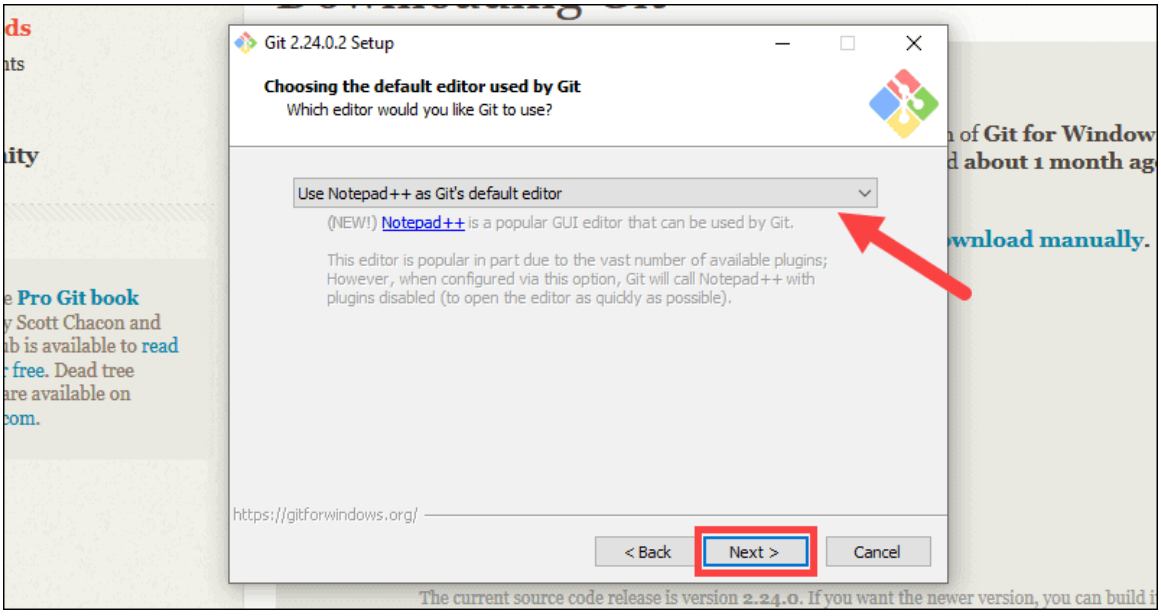
7. A component selection screen will appear. Leave the defaults unless you have a specific need to change them and click Next.



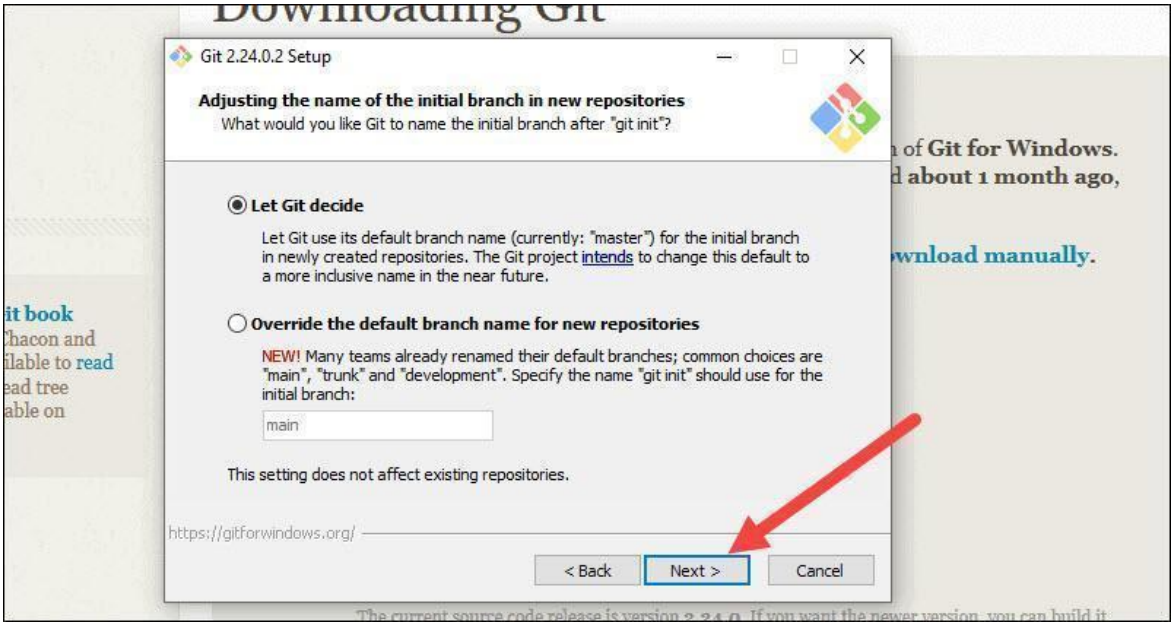
8. The installer will offer to create a start menu folder. Simply click Next.



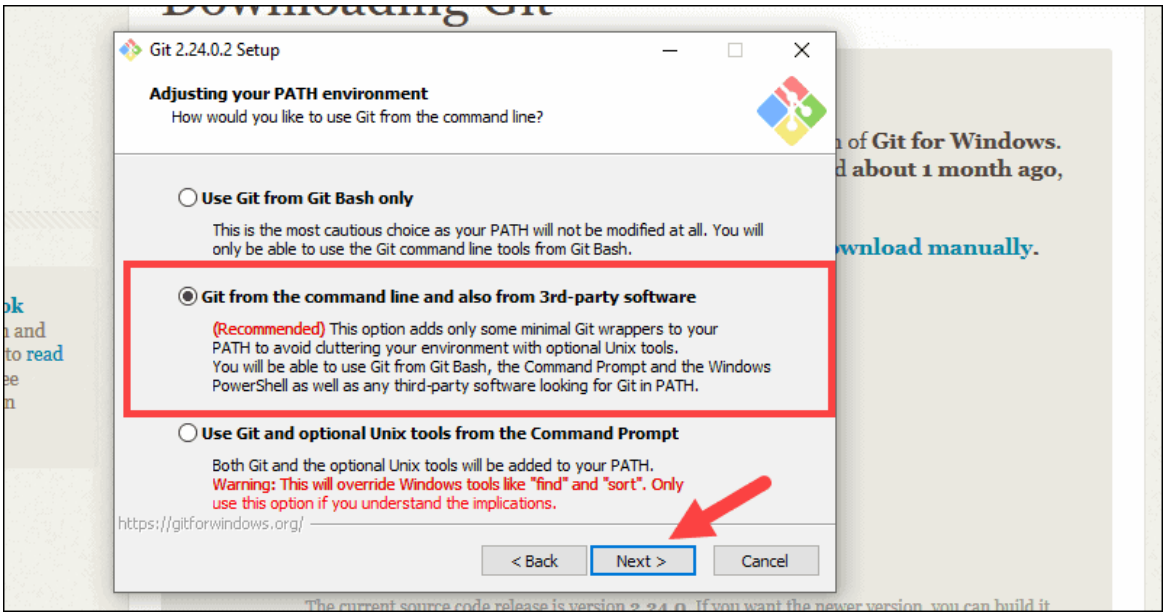
9. Select a text editor you'd like to use with Git. Use the drop-down menu to select Notepad++ (or whichever text editor you prefer) and click Next.



10. The next step allows you to choose a different name for your initial branch. The default is 'master.' Unless you're working in a team that requires a different name, leave the default option and click Next.



11. This installation step allows you to change the PATH environment. The PATH is the default set of directories included when you run a command from the command line. Leave this on the middle (recommended) selection and click Next.

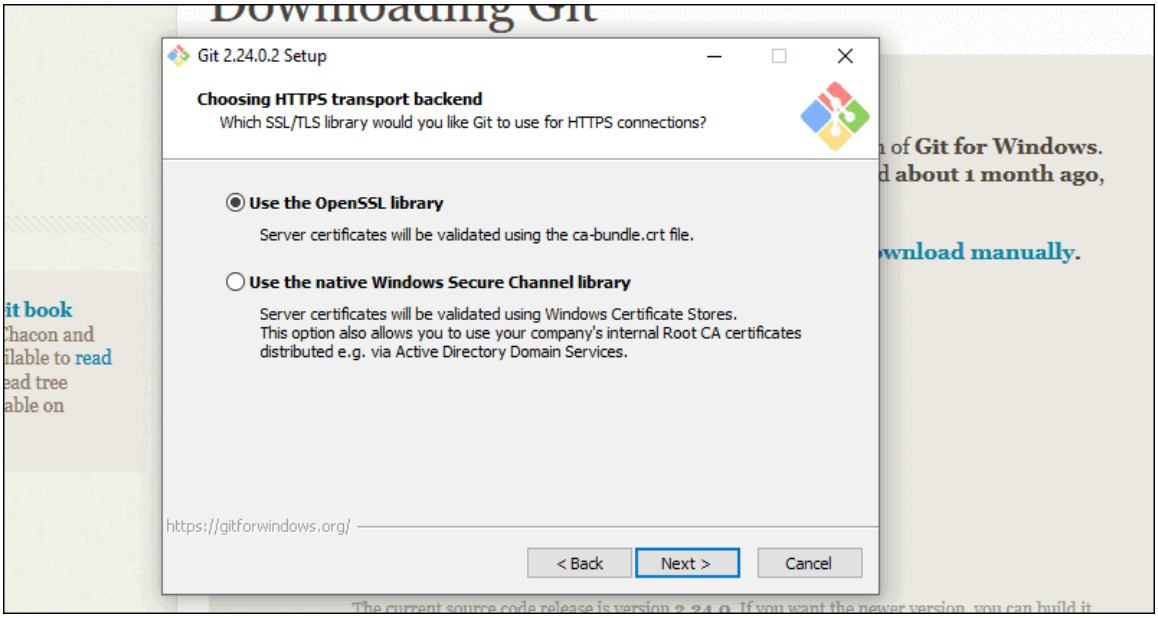


Server Certificates, Line Endings and Terminal Emulators

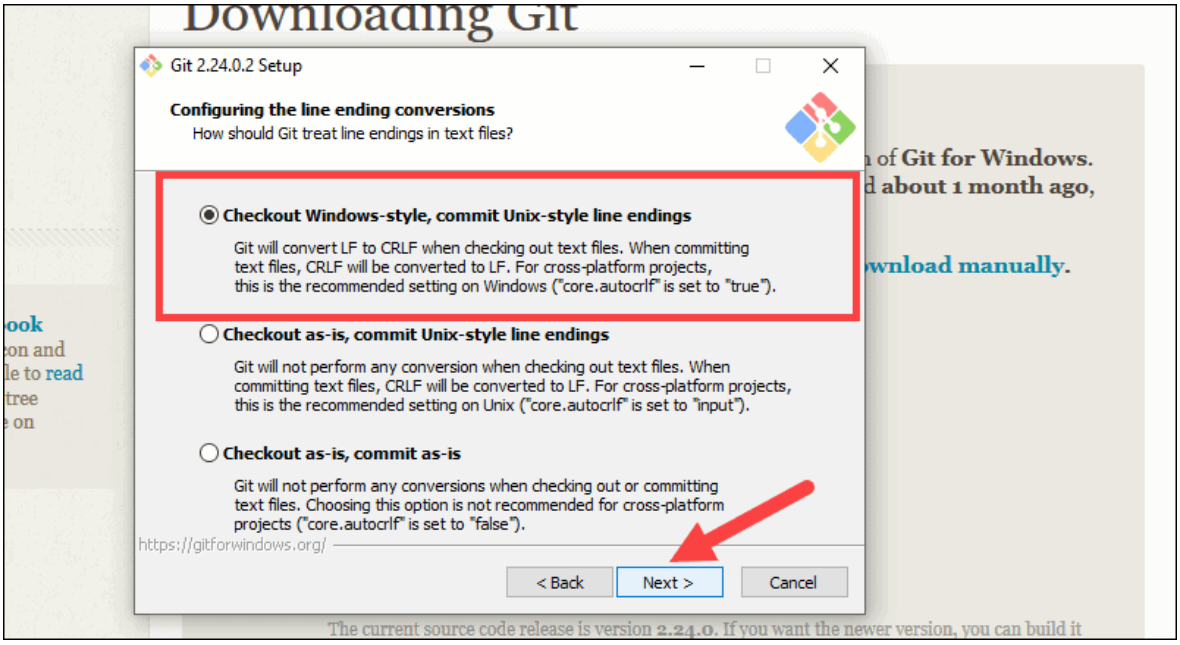
12. The installer now asks which SSH client you want Git to use. Git already comes with its own SSH client, so if you don't need a specific one, leave the default option and click Next.

Note: Check out our comparison of SSH and HTTPS for Git and which one you should use.

13. The next option relates to server certificates. Most users should use the default. If you're working in an Active Directory environment, you may need to switch to Windows Store certificates. Click Next.

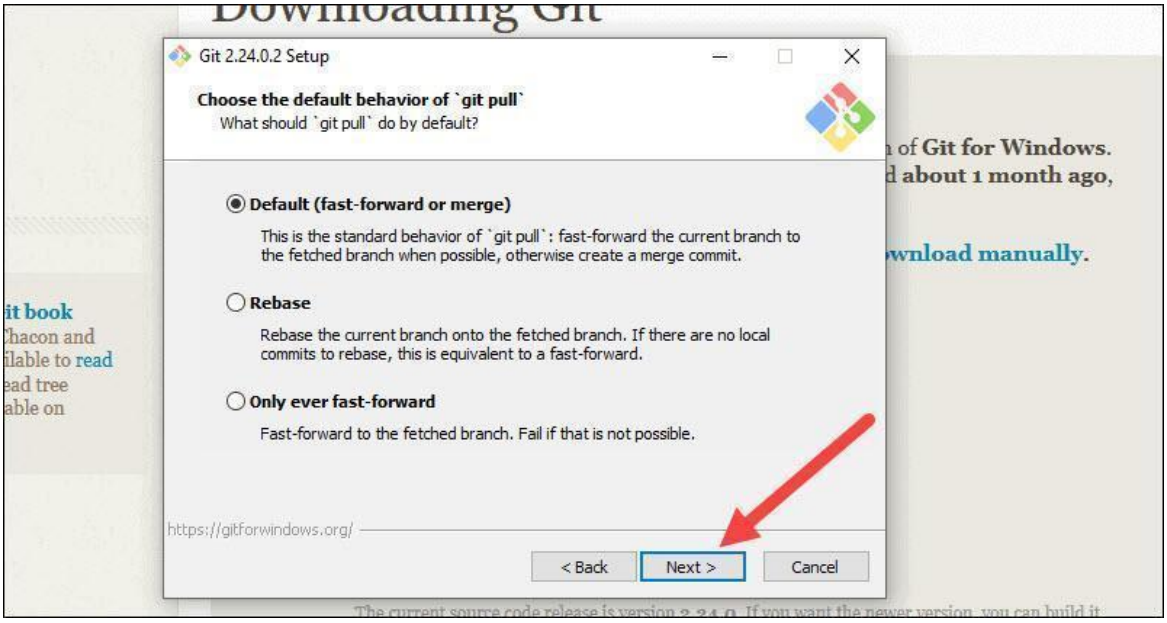


14. The next selection converts line endings. It is recommended that you leave the default selection. This relates to the way data is formatted and changing this option may cause problems. Click Next.

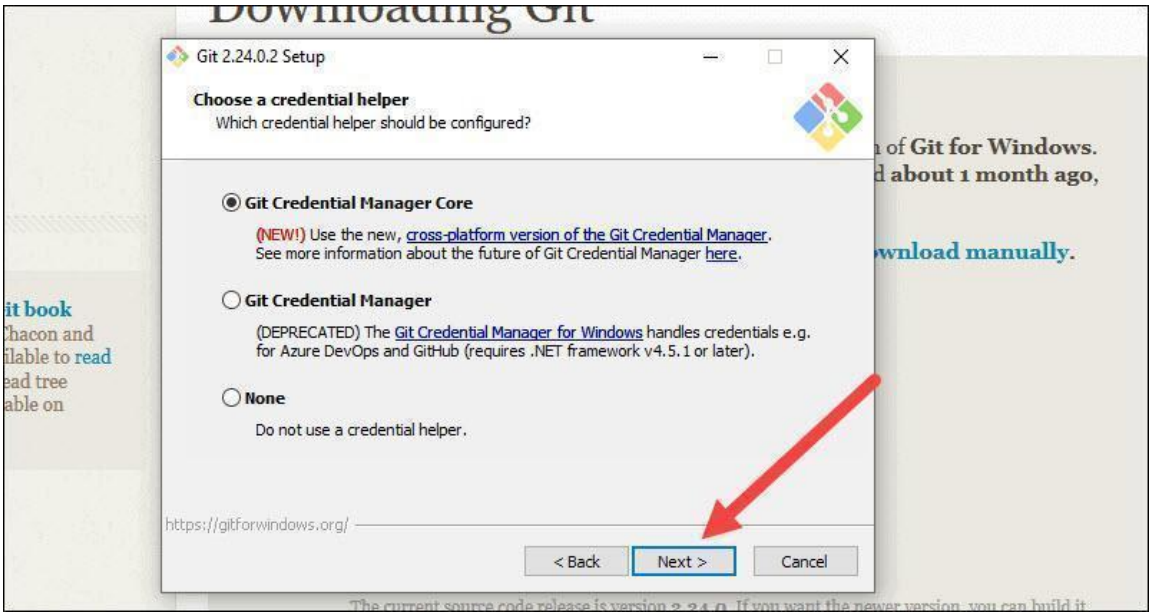


15. Choose the terminal emulator you want to use. The default MinTTY is recommended, for its features. Click Next.

16. The installer now asks what the git pull command should do. The default option is recommended unless you specifically need to change its behavior. Click Next to continue with the installation.

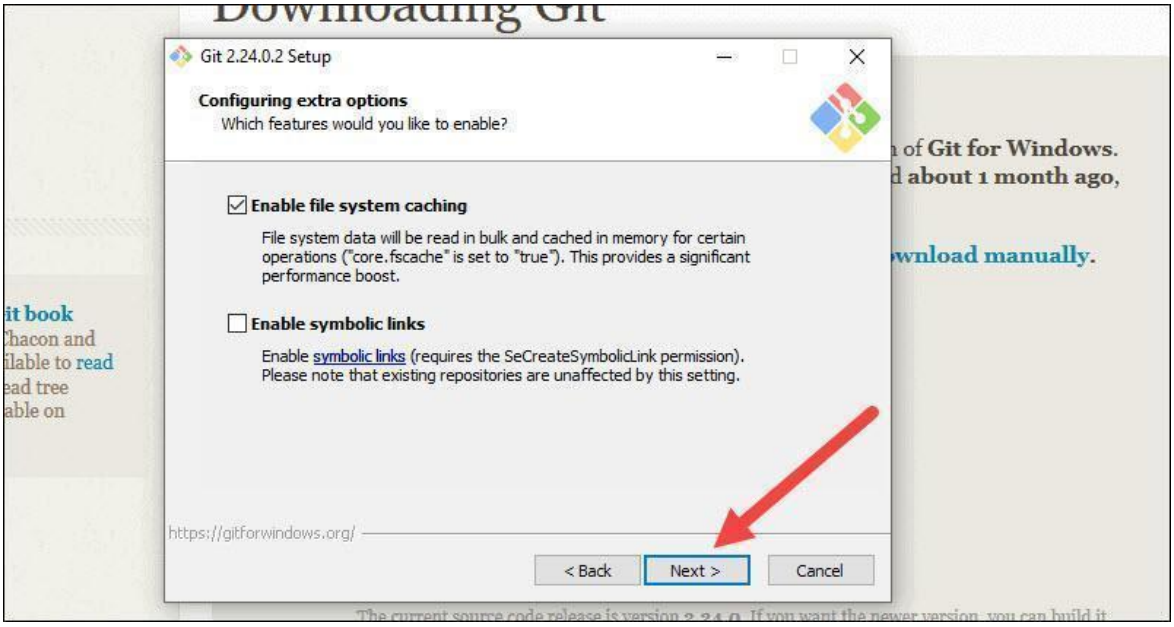


17. Next you should choose which credential helper to use. Git uses credential helpers to fetch or save credentials. Leave the default option as it is the most stable one, and click Next.

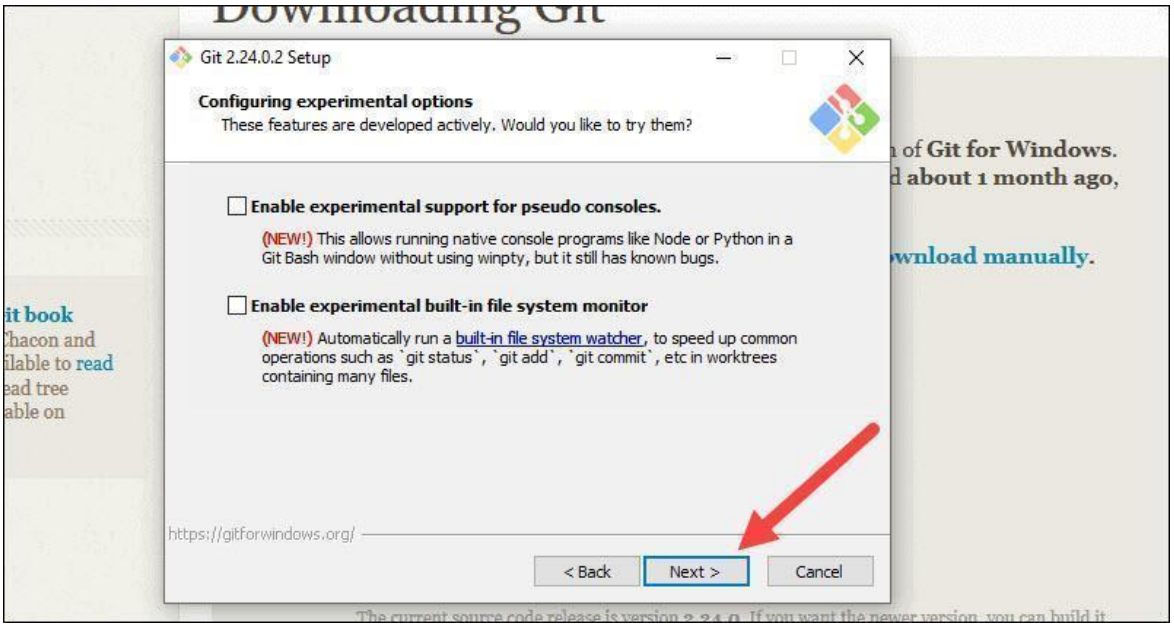


Additional Customization Options

18. The default options are recommended; however this step allows you to decide which extra option you would like to enable. If you use symbolic links, which are like shortcuts for the command line, tick the box. Click Next.

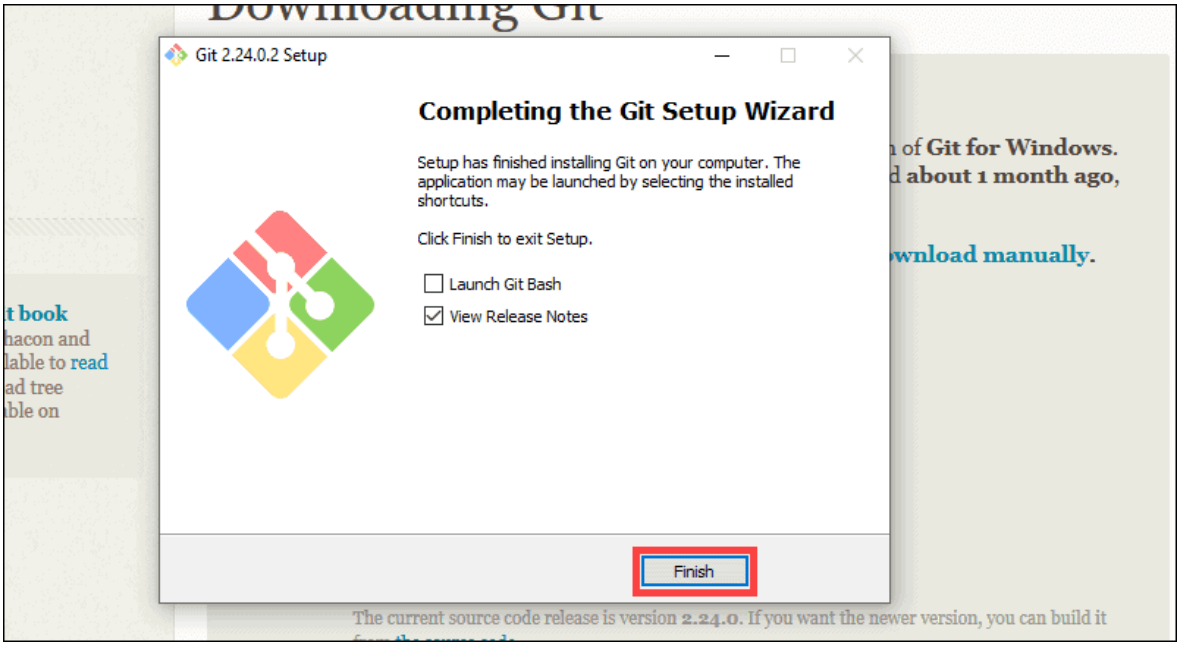


19. Depending on the version of Git you're installing, it may offer to install experimental features. At the time this article was written, the options to include support for pseudo controls and a built-in file system monitor were offered. Unless you are feeling adventurous, leave them unchecked and click Install.



Complete Git Installation Process

20. Once the installation is complete, tick the boxes to view the Release Notes or Launch Git Bash, then click Finish.



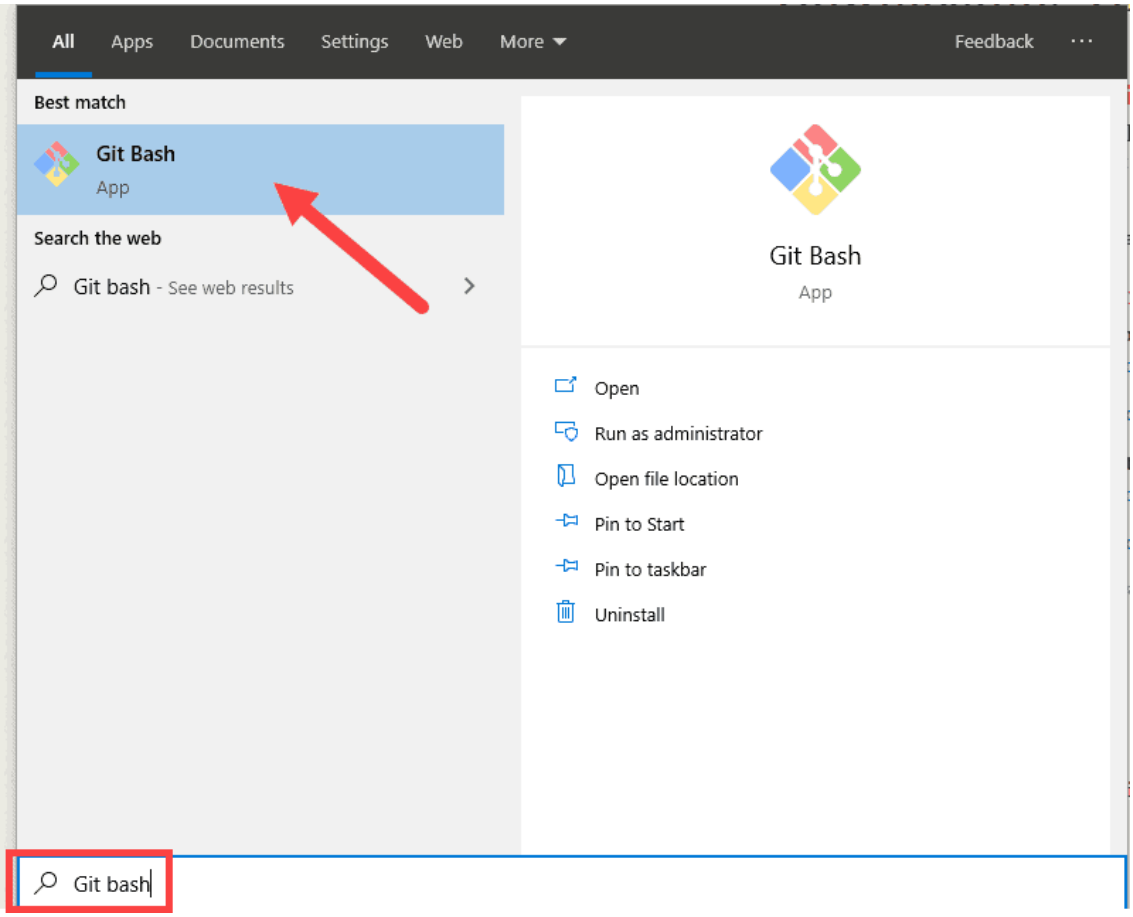
Note: Learn the differences between CLI and GUI.

1.3 How to Launch Git in Windows

Git has two modes of use – a bash scripting shell (or command line) and a graphical user interface (GUI).

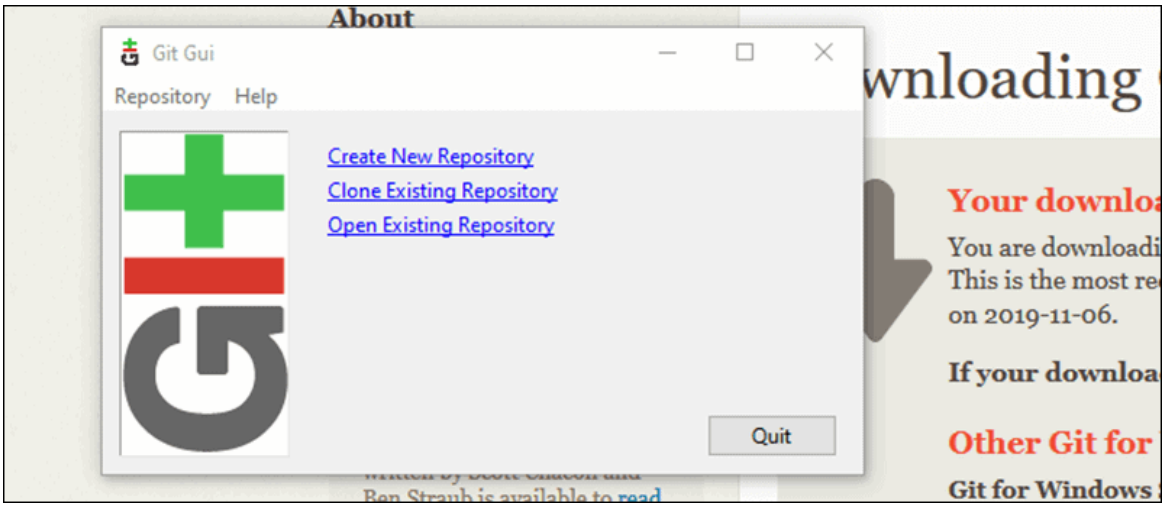
a) Launch Git Bash Shell

To launch Git Bash open the Windows Start menu, type git bash and press Enter (or click the application icon).



b) Launch Git GUI

To launch Git GUI open the Windows Start menu, type git gui and press Enter (or click the application icon).



Steps to perform git configuration:

Performing Git configuration involves setting up your identity (name and email), choosing a text editor, and configuring other settings. Here are the steps to perform Git **configuration:**

1. verify that Git is installed

After the installation is complete, Sarah opens a new terminal or command prompt to verify that Git is installed. She runs the following command:

```
git --version
```

2. Set Your Name and Email:

Use the git config command to set your name and email. This information will be associated with your commits.

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Replace "Your Name" with your actual name and "your.email@example.com" with your actual email address.

3. Configure Your Preferred Text Editor:

Set your preferred text editor for Git commit messages. This is the editor that will open when you make a commit.

For example, if you want to use Visual Studio Code:

```
git config --global core.editor "code --wait"
```

Replace "code" with the command for your preferred text editor.

4. Check Your Configuration:

To view your Git configuration, use the following command:

```
git config --list
```

5. Configure Default Branch Name (Optional):

If you want to use a branch name other than "master" or "main" as the default branch name for new repositories, you can configure it:

```
git config --global init.defaultBranch main
```

Replace "main" with your preferred default branch name.

6. Set Up a Global .gitignore (Optional):

If you want to use a global .gitignore **file across multiple repositories, you can create a global** .gitignore **file and tell Git to use it:**

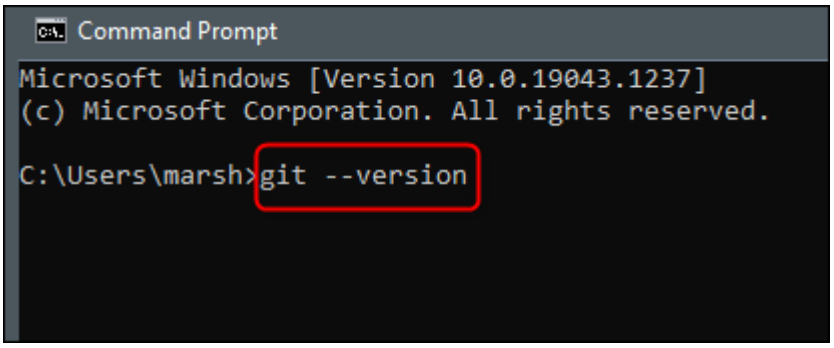
```
# Create a global .gitignore file
touch ~/.gitignore_global

# Open the global .gitignore file in a text editor and add your rules
nano ~/.gitignore_global

# Configure Git to use the global .gitignore file
git config --global core.excludesfile ~/.gitignore_global
```

7. Verify the installation:

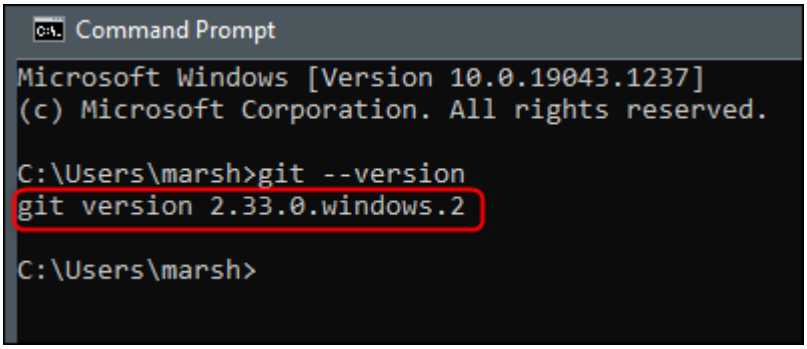
Check Which Version of Git You're Using

The command to check which version of Git you're using is the same on both Windows and Mac. To check your Git version, open Command Prompt (Windows), Terminal (Mac), or the Linux terminal. Once open, run this command:

git --version



The Git version you're currently using will be returned.



Now that you know which version of Git you're using, you can decide if you want to update it or not.

How to Update Git on Windows

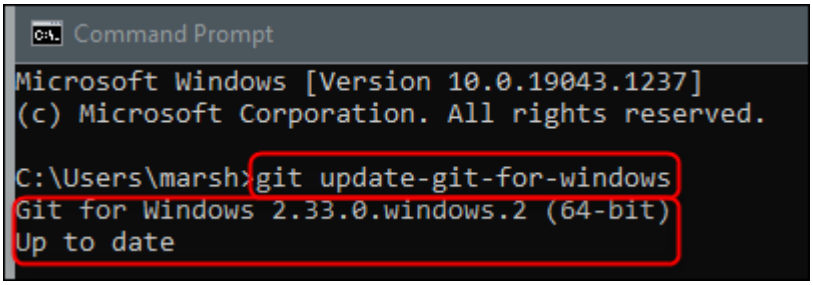
The command you use to update Git on Windows depends on which version of Git you're currently using. If you're using any version from 2.14.2 to 2.16.1, then run this command in Command Prompt:

git update

If you're using any version after 2.16.1, then you'll need to run this command instead:

git update-git-for-windows

Regardless of which command you need to use, your Git version will update or you'll get a message saying you're up to date if you're already using the latest version.



If you're using a version older than 2.14.2, then you'll need to get the latest installer from the download portal and update your Git version the same way as when you installed Git for the first time.

Apply Git configuration

The git config command is used to set or get configuration variables in Git. These variables can control various aspects of Git's behavior, such as user information, default behavior, and repository settings.

Here are some practical Git configurations with examples:

1. Set your name and email:

git config --global user.name "Your Name"

git config --global user.email "your@email.com"

This configuration sets your name and email globally, which will be used for all your Git commits.

2. Configure your preferred text editor:

git config --global core.editor "vim"

This configuration sets Vim as the default text editor for Git. You can replace "vim" with the name of your preferred editor.

3. Set default branch name:

git config --global init.defaultBranch "main"

This configuration sets the default branch name to "main" when creating a new repository.

4. Enable colored output:

git config --global color.ui true

This configuration enables colored output in Git's command-line interface, making it easier to read and interpret.

5. Exclude certain files from being tracked:

git config --global core.excludesfile ~/.gitignore_global

This configuration specifies a global gitignore file that contains patterns for files and directories you want Git to ignore by default.

These are just a few examples of practical Git configurations. You can explore more configuration options and customize Git based on your specific needs.

6. Git Init command

In order to work with code using Git, you need to store your code in a Git repository . Repositories, or repos, are storage containers for a project where you can save different versions of your code.

There are two ways to start working with Git. First, you can clone an existing repository using git clone. This will copy all the code and history from an existing project to your local machine. Second, you can create a new repository using git init, which will have its own versioning system and history.

The git init command creates an empty Git repository. init can be used to convert an existing

project into a Git repository. The init command can also initialize an empty repository for a new project.

What Happens When You Use Git Init

When you run git init, a folder called .git is created in your current working directory (the folder you are viewing). This folder contains all the files and metadata used by the Git version control system. For instance, in this folder you will see a file called HEAD. **The GIt HEAD file** points to the Git commit which you are viewing on your local machine.

The git init command does not change the project in the folder in which you run the command. This is because all the main files git needs are stored within the .git directory that the git init command creates.

The git init command is the first command you'll run if you are starting a new Git project

How to Use Git Init

The git init command is easy to use. You don't need to create a repository on a server to start working with a git repository. Instead, you only have to navigate into your project folder and run the git init command.

Here's the syntax to create a git repo using the git init command:

git **init**

This command will initialize a new Git repository in the current working directory. So, before you run the command, make sure you are in the directory in which you want to initialize a repository.

Alternatively, you can specify the directory in which the new repository should be initialized. The syntax for doing so is as follows:

git **init** <folder>

Suppose we wanted to initialize a repository in a folder called demo-project . We could do so using this code:

git **init** demo-project

When we run this command, a .git folder is created within our *demo-project* folder, instead of in our current working directory.

You can run the git init command in a folder which already has an existing git configuration. This is because git init does not override an existing configuration. So, if you accidentally run git init in an existing Git repository, nothing will happen.

4.7 Configure .gitignore file

Configuring ignored files for a single repository

You can create a .gitignore file in your repository's root directory to tell Git which files and directories to ignore when you make a commit. To share the ignore rules with other users who clone the repository, commit the .gitignore file in to your repository.

GitHub maintains an official list of recommended .gitignor*e* files for many popular operating systems, environments, and languages in the GitHub/gitignore public repository. You can also use gitignore.io to create a .gitignore file for your operating system, programming language, or IDE.

Open Git Bash.

Navigate to the location of your Git repository.

Create a .gitignor*e* file for your repository.

$ touch .gitignore

If the command succeeds, there will be no output.

If you want to ignore a file that is already checked in, you must untrack the file before you add a rule to ignore it. From your terminal, untrack the file.

$ git rm --cached FILENAME

Configuring ignored files for all repositories on your computer

You can also create a global .gitignore file to define a list of rules for ignoring files in every Git repository on your computer. For example, you might create the file at ~/.gitignore_global and add some rules to it.

Open Git Bash.

Configure Git to use the exclude file ~/.gitignore_global for all Git repositories.

$ git config --global core.excludesfile ~/.gitignore_global

Excluding local files without creating a .gitignore file

If you don't want to create a .gitignore file to share with others, you can create rules that are not committed with the repository. You can use this technique for locally-generated files that you don't expect other users to generate, such as files created by your editor.

Use your favorite text editor to open the file called *.git/info/exclude* within the root of your Git repository. Any rule you add here will not be checked in, and will only ignore files for your local repository.

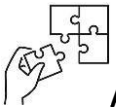
Open Git Bash.

Navigate to the location of your Git repository.

Using your favorite text editor, open the file *.git/info/exclude*.

Points to Remember

- Git's architecture is designed around the concept of a distributed version control system, allowing users to work independently and collaborate seamlessly. There are the key co mponents and concepts of Git's architecture including: Working Directory, Index (Staging Area), Local Repository, Remote Repository.
- Git, a powerful distributed version control system, is built on several fundamental conce pts that form the foundation of its functionality. Here are Git's basic concepts including: Repository, commit, branch, merge, pull, push.
- To install Git on Windows, follow these steps:
    1. Download the Git installer
    2. Run the Git installer
    3. Choose the installation options
    4. Complete the installation
    5. Verify the installation
- Basic commands to Configure GIT
    1. Git init command
    2. Git config command
    3. Git – version command



Application of learning 1.2.

XYZ Company, as part of your responsibilities, you need to showcase your expertise in Git for our software development project. This involves installing the current version of Git, verifying the installation, initializing a Git repository in the XYZ project folder, and configuring the.gitignore file to exclude dot env file paths for security purposes

Indicative content 1.3: Use of GitHub repository

Duration: 6 hrs

Theoretical Activity 1.3.1: Description of GitHub

Tasks:

1. In small groups, you are requested to answer the following questions related to the GitHub:
    I. What do you understand about GitHub?
    II. Provide an explanation of:
        - The features of GitHub
        - Benefits of GitHub
        - GitHub account
        - Remote repository
        - Git repository commands as used in GitHub.
2. Provide the answer for the asked questions and write them on papers.
3. Present the findings/answers to the whole class.
4. For more clarification, read the key readings 1.3.1. In addition, ask questions where necessary.

Key readings 1.3.1: Description of GitHub

1. Definition of GitHub

GitHub is a large platform for code hosting. It supports version controlling and collaboration and allows developers to work together on projects. It offers both distributed version control and source code management (SCM) functionality of Git. It also facilitates collaboration features such as bug tracking, feature requests, task management for every project.

Essential components of the GitHub are:

- Repositories
- Branches
- Commits
- Pull Requests
- Git (the version control tool GitHub is built on)

2. Benefits of GitHub

GitHub can be separated as the Git and the Hub. GitHub service includes access controls as well as collaboration features like task management, repository hosting, and team management.

The key benefits of GitHub are as follows.

- It is easy to contribute to open source projects via GitHub.
- It helps to create an excellent document.
- You can attract the recruiter by showing off your work. If you have a profile on GitHub, you will have a higher chance of being recruited.
- It allows your work to get out there in front of the public.
- You can track changes in your code across versions.

3. Features of GitHub

GitHub is a place where programmers and designers work together. They collaborate, contribute, and fix bugs together. It hosts plenty of open source projects and codes of various programming languages. It is a web-based platform that uses Git, the open-source version control software, to help developers manage and share their code. Here are some of its key features:

1. Collaboration
2. Bug tracking
3. Branches
4. Git repositories
5. Project management
6. Team management
7. Code hosting
8. Track and assign tasks

3.1. Collaboration

Collaboration is the process of two or more people, entities or organizations working together to complete a task or achieve a goal. Collaboration is similar to cooperation

3.2. Integrated issue and bug tracking

Bug and issue tracking systems are often implemented as a part of integrated project management systems. This approach allows including bug tracking and fixing in a general product development process, fixing bugs in several product versions, automatic generation of a product knowledge base and release notes.

3.3. Graphical representation of branches

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. You can think of them as a way to request a brand new working directory, staging area, and project history.

3.4. Git repositories hosting

A Git repository is a central storage location for managing and tracking changes in files and directories. It is a crucial component of the Git version control system, which enables collaborative development and allows multiple developers to work on a project simultaneously.

3.5. Project management

Project management is the application of processes, methods, skills, knowledge and experience to achieve specific project objectives according to the project acceptance criteria within agreed parameters. Project management has final deliverables that are constrained to a finite timescale it achieves by allowing multiple developers to collaborate and share the tasks which will be merged to produce a full package of project.

3.6. Team management

For each repository that you administer on GitHub, you can see an overview of every team or person with access to the repository. From the overview, you can also invite new teams or people, change each team or person's role for the repository, or remove access to the repository.

3.7. Code hosting

GitHub's code hosting capabilities make it a powerful platform for collaborative software development, making it easier for developers and teams to work together on code projects, whether they are open source or private repositories.

3.8. Track and assign tasks

GitHub's "Track and Assign Tasks" feature allows you to keep organized and assign work to team members. You can create tasks, known as issues, with descriptions and due dates. You can assign these issues to specific team members who are responsible for them. Notifications keep everyone in the loop when tasks are assigned or updated. You can use project boards to visually track task progress. Overall, this feature helps teams stay on top of their work, delegate responsibilities, and make sure nothing falls through the cracks.

3.9. Conversations

git hub allows different developers to have conversation by allowing them to text each other a message.

4. GitHub vs. Git

Git is an open-source distributed version control system that is available for everyone at zero cost. It is designed to handle minor to major projects with speed and efficiency. It is developed to co-ordinate the work among programmers. The version control allows you to track and work together with your team members at the same workspace.

While GitHub is an immense platform for code hosting, it supports version controlling and collaboration. It allows developers to work together on projects.

It offers both distributed version control and source code management (SCM) functionality of Git. It also facilitates collaboration features such as bug tracking, feature requests, task management for every project.

GitHub	Git
It is a cloud-based tool developed around the Git tool.	It is a distributed version control tool that is used to manage the programmer's source code history.
It is an online service that is used to store code and push from the computer running Git.	Git tool is installed on our local machine for version controlling and interacting with online Git service.
It is dedicated to centralize source code hosting.	It is dedicated to version control and code sharing.
It is managed through the web.	It is a command-line utility tool.
It provides a desktop interface called GitHub desktop GUI.	The desktop interface of Git is called Git GUI.
It has a built-in user management feature.	It does not provide any user management feature
It has a market place for tool configuration.	It has a minimal tool configuration feature.

5. Git commands related to repository

There are several Git commands related to managing and working with repositories. Here are some common of these commands:

1. `Git init`: Initializes a new Git repository in the current directory, creating a new `.git` folder to store version control information.

2. `Git clone <repository-url>`: Creates a local copy of a remote repository. The repository URL can be obtained from the hosting platform (e.g., GitHub, GitLab).

3. `Git remote add <name> <repository-url>`: Adds a remote repository to your local repository. The `<name>` is an alias for the remote repository, and the `<repository-url>` is the URL of the remote repository.

4. `Git remote -v`: Lists the remote repositories associated with your local repository, along with their URLs.

5. `Git pull <remote> <branch>`: Fetches changes from a remote repository and merges them into the current branch in your local repository.

6. `Git push <remote> <branch>`: Pushes your local commits to a remote repository. The `<remote>` is the remote repository alias, and the `<branch>` is the branch name.

7. `Git branch`: Lists all the branches in your local repository. The current branch is indicated with an asterisk.
8. `Git branch <branch-name>`: Creates a new branch with the specified name.
9. `Git checkout <branch-name>`: Switches to the specified branch.
10. `Git checkout -b <branch-name>`: Creates a new branch with the specified name and switches to it.
11. `Git merge <branch-name>`: Merges changes from the specified branch into the current branch.
12. `Git status`: Shows the current status of your local repository, including modified files, new files, and branch information.
13. `Git log`: Displays a log of commits in reverse chronological order, showing commit hashes, authors, dates, and commit messages.
14. `Git add <file>`: Adds a file to the staging area, preparing it for committing.
15. `Git commit -m "<commit-message>"`: Commits the changes in the staging area with a descriptive commit message.
16. `Git push --tags`: Pushes any tags that have been created to the remote repository.

Practical Activity 1.3.2: Using GitHub

Task:

1: Referring to the previous theoretical activity (1.3.1) you are requested to go to the computer lab to use GitHub. This task should be done individually.
2: Apply safety precautions.
3: Present out the steps to use GitHub.
4: Referring to the steps provided on task 3, use GitHub.
5: Present your work to the trainer and whole class.
6: Read key reading 1.3.2 and ask clarification where necessary
7: Perform the task provided in application of learning 1.3.2.

Key readings 1.3.2: Creating account and repository on GitHub

1. Create GitHub Account

After installing Git on your machine, the next step is to create a free GitHub account by following these steps:

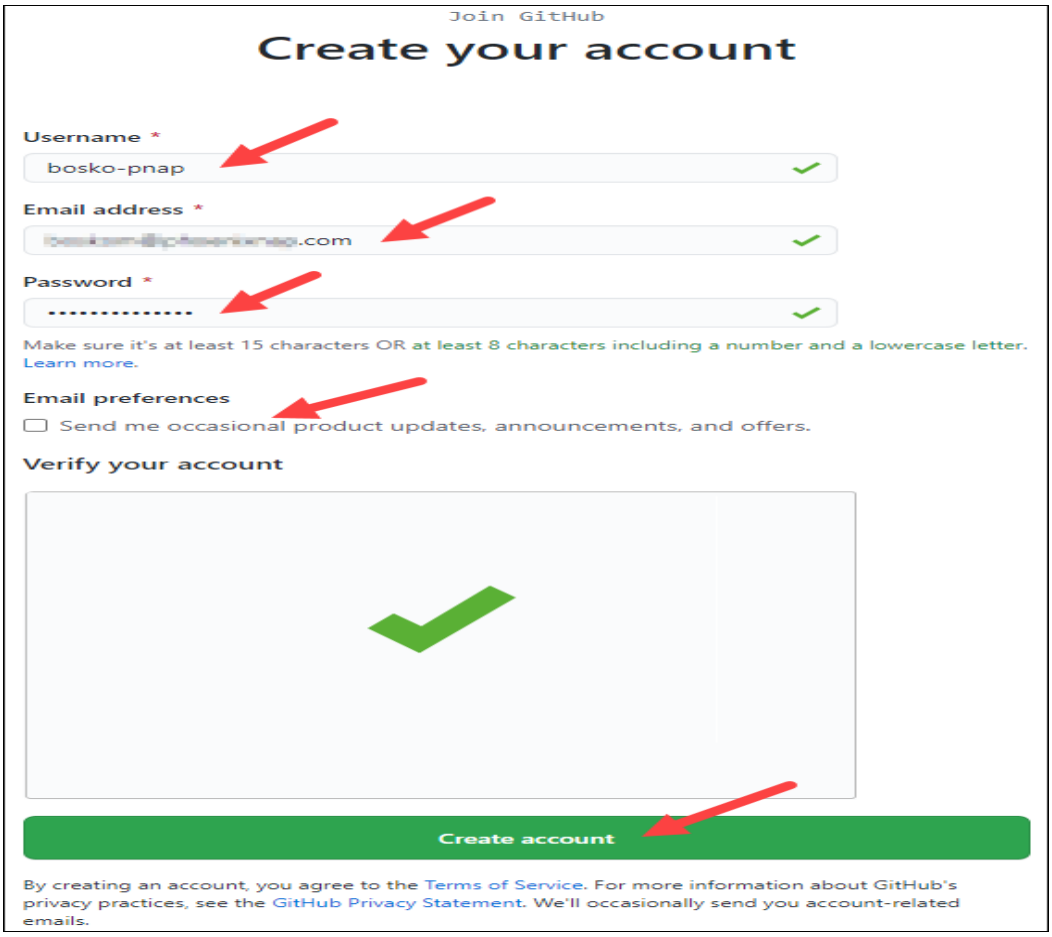
1. Visit the official account creation page by Joining GitHub

2. Pick a **username**, enter your **email address**, and choose a **password**.

3. Opt for or opt out of receiving updates and announcements by checking/unchecking the **Email preferences** checkbox.

4. Verify you're not a robot by solving the **Captcha** puzzle.

5. Click **Create account**.



6. GitHub sends a **launch code** to the specified email address. Copy-paste the code in the designated field.

7. Optionally, enter account personalization details when asked or **Skip**, and click **Continue**.

You have now successfully created a GitHub account.

2. Create a Local Git Repository

After installing or updating Git, the next step is to **create a local Git repository**.

To create a Git repository, follow the steps below:

1. Open a Git Bash terminal and move to the directory where you want to keep the project

on your local machine. For example:

cd ~/Desktop

mkdir myproject

cd myproject/

In this example, we changed the directory to **Desktop** and created a subdirectory called *myproject*.

2. Create a Git repository in the selected folder by running the git init command. The syntax is:

git init [repository-name]

```
Boško@Bosko MINGW64 ~/Desktop/myproject
$ git init
Initialized empty Git repository in C:/Users/Snesko/Desktop/myproject/.git/

Boško@Bosko MINGW64 ~/Desktop/myproject (master)
$
```

Now you have successfully created a local Git repository.

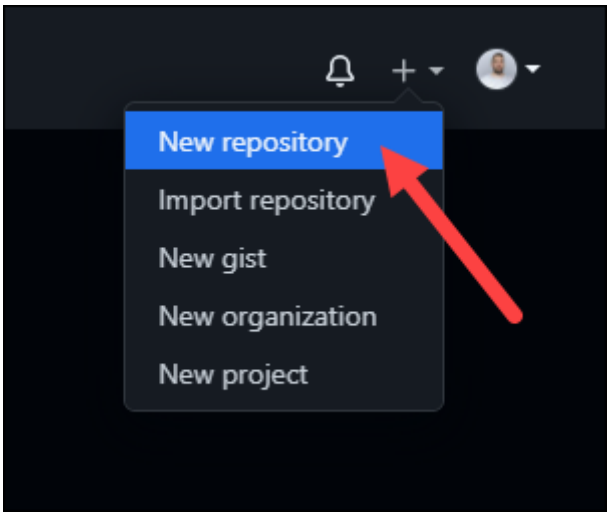
3. Create New Repository on GitHub

GitHub allows you to keep track of your code when you're working with a team and need to modify the project's code collaboratively.

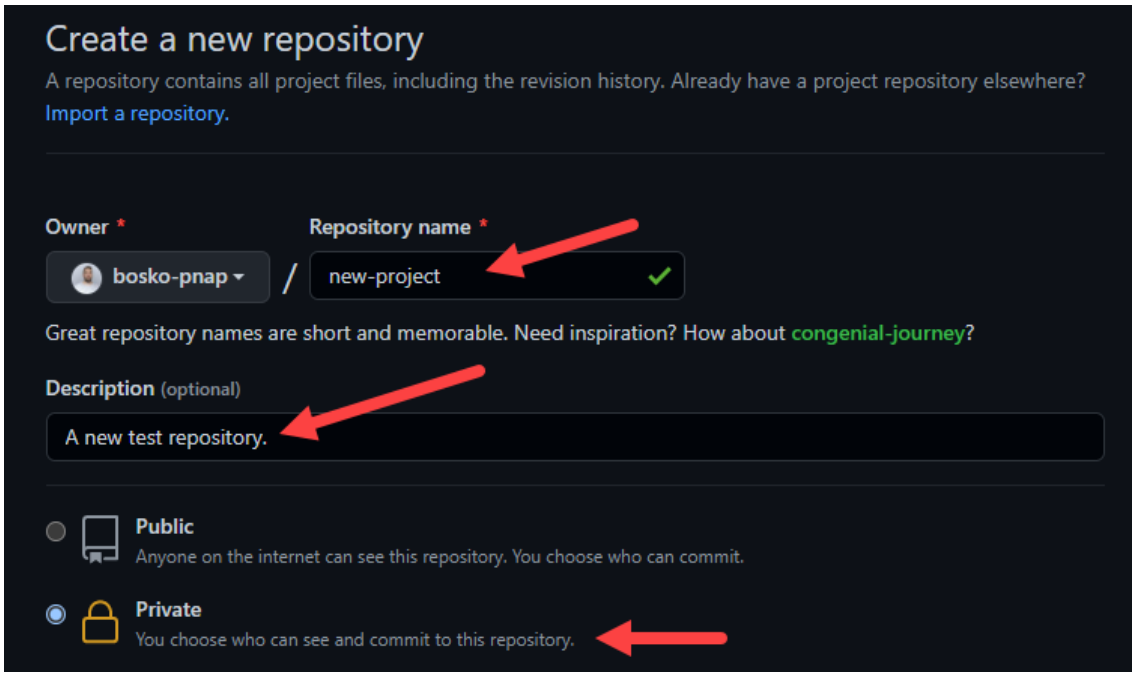
Follow these steps to create a new repository on GitHub:

1. Log in and browse to the GitHub home page.

2. Find the **New repository** option under the **+** sign next to your profile picture, in the top right corner.



3. Enter a name for your repository, provide a brief description, and choose a privacy setting.



4. Click the **Create repository** button.

GitHub allows you to add an existing repo you have **created locally**.

4.Git commands related to repository

1. **git init:** Initializes a new Git repository in the current directory.

Example:

$ git init

2. **git clone <repository>:** Creates a copy of a remote repository on your local machine.

Example:

$ git clone https://github.com/user/repository.git

3. **git remote add <name> <url>:** Adds a remote repository to your local repository.

Example:

$ git remote add origin https://github.com/user/repository.git

4. **git remote remove <name>:** Removes a remote repository from your local repository.

Example:

$ git remote remove origin

5. **git remote -v:** Lists the remote repositories associated with your local repository.

Example:

$ git remote -v

6. **git fetch <remote>:** Fetches the latest changes from a remote repository without merging them.

Example:

$ git fetch origin

7. **git pull <remote> <branch>:** Fetches the latest changes from a remote repository and merges them into the current branch.

Example:

$ git pull origin main

8. **git push <remote> <branch>:** Pushes your local changes to a remote repository.

Example:

$ git push origin main

9. **git branch:** Lists all the branches in your repository.

Example:

$ git branch

10. **git branch <branch_name>:** Creates a new branch in your repository.

Example:

$ git branch new-feature

11. **git checkout <branch>:** Switches to an existing branch in your repository.

Example:

$ git checkout main

12. **git checkout -b <branch_name>:** Creates and switches to a new branch in your repository.

Example:

$ git checkout -b new-feature

13. **git merge <branch>:** Merges a branch into the current branch.

Example:

$ git merge feature-branch

14. **git add <file>:** Adds a file to the staging area to be included in the next commit.

Example:

$ git add myfile.txt

15. **git commit -m "<message>":** Commits the changes in the staging area with a descriptive message.

Example:

$ git commit -m "Fix bug #123"

16. git log: Displays a log of commits in reverse chronological order.

Example:

$ git log

17. **git status:** Shows the current state of your repository, including any untracked, modified, or staged files.

Example:

$ git status

18. **git diff:** Shows the differences between the working directory and the staging area.

Example:

$ git diff

Steps to follow when performing git-config command

1. Open a terminal or command prompt
2. Check existing configurations (optional):
3. Set global or local configurations
4. Modify existing configurations
5. Remove configurations

6. Confirm configurations

Steps to perform git commands related to repository:

1. Git Clone

1. **Open Terminal or Command Prompt:** Open the Terminal (on macOS and Linux) or Command Prompt (on Windows) on your computer. This will be the interface where you'll enter Git commands.

2. **Navigate to the Desired Directory:** Use the `cd` command to navigate to the directory where you want to clone the repository. For example, if you want to clone the repository into the "Documents" folder**,** use the following command**:**

cd Documents

3. **Clone the Repository:** Use the `git clone` command followed by the URL of the repository you want to clone. The URL can be obtained from the repository's GitHub page. For example, if the repository URL is `https://github.com/username/repository.git`, use the following command:

git clone https://github.com/username/repository.git

4. **Provide Credentials (if required):** If the repository is private and requires authentication, Git may prompt you to enter your GitHub username and password or access token. Enter the required credentials to proceed with the cloning process.

5. **Wait for Cloning to Complete:** Git will begin cloning the repository into the current directory.

2. Git Remote

To apply Git commands related to remote repositories, such as **`git remote`,** follow these steps:

1. **Navigate to the Local Repository:** Open the Terminal or Command Prompt and navigate to the directory of your local Git repository using the **`cd`** command.

2. **Check Existing Remotes**: To see the list of existing remote repositories associated with your local repository, use the **`git remote`** command. This will display the names of the remote repositories. For example: **git remote**

3. **Add a Remote**: To add a new remote repository, use the **`git remote add`** command followed by a name and the URL of the remote repository. The name can be anything you choose, and the URL should be the address of the remote repository. For example:

git remote add origin https://github.com/username/repository.git

In this example, **`origin`** is the name given to the remote repository, but you can choose

a different name if you prefer.

4. **Rename a Remote:** If you want to rename an existing remote repository, use the **`git remote rename`** command followed by the current name and the new name. For example, to rename the remote repository from **`origin`** to **`new-origin`,** use the following command:

git remote rename origin new-origin

5. **Remove a Remote:** If you want to remove an existing remote repository, use the **`git remote remove`** command followed by the name of the remote repository. For example, to remove the remote repository named **`origin`,** use the following command:

git remote remove origin

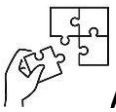
6. **Get Detailed Information:** To get more detailed information about a specific remote repository, use the **`git remote show`** command followed by the name of the remote repository. For example, to get detailed information about the remote repository named **`origin`,** use the following command: **git remote show origin**

Points to Remember

- GitHub, widely used platform for software development and collaboration, offers a r ange of features that facilitate version control, project management, code sharing, a nd team collaboration. There are some key features of GitHub including: Collaboratio n, Integrated issue and bug tracking, Graphical representation of branches, Git reposi tories hosting, Project management, Team management.
- GitHub accounts are user profiles on the GitHub platform that provide individuals an d organizations with access to various features and functionalities for managing and collaborating on software projects.

- **To use GitHub effectively, follow these steps:**

1. Create GitHub account
2. Create repository in your account
3. Perform git-config command
4. Perform git commands related to repository



Application of learning 1.3.

Alex, a software developer, is starting a new project to build a personal finance tracking application. Alex wants to use GitHub to manage the project's version control and collaborate with other developers. How Alex goes through the process to perform these tasks:

1. Set Up the Project Directory
2. Initialize Git Repository
3. Create a GitHub Repository
4. Link Local Repository to GitHub
5. Add Initial Files
6. Push to GitHub
7. Collaborate with Other Developers

Learning outcome 1 end assessment

Theoretical assessment

1. **Read the Following statement and answer by true if correct or false otherwise**

Version control systems (VCS), including distributed versions like Git, are essential tools in software development that enable tracking changes over time, facilitate collaboration, and maintain a comprehensive version history. Through concepts like branching and merging, developers can work on separate features or fixes independently and later integrate them into the main codebase. Key commands like git pull to fetch and merge changes, git status to view repository state, and git branch to manage branches, all contribute to efficient project management by providing snapshots of the project at specific points and allowing seamless collaboration.

a. Version control systems (VCS) track changes to files over time, enabling collaboration and version history.
b. Distributed version control systems (DVCS) allow each user to have a complete copy of the repository with its full history.
c. Branching in version control allows developers to work on separate features or fixes without interfering with the main codebase.
d. Merging in version control combines changes from one branch or fork into another, integrating separate lines of development.
e. Commits in version control systems are snapshots of the entire project at a specific point in time, including all tracked files.
f. git pull fetches changes from a remote repository and merges them into the local repository.
g. git status displays information about the current state of the repository, such as modified files and branch status.
h. git branch is used to create, list, delete, or manipulate branches in a Git repository.

2. Match Git commands with its corresponding description

Command	**Description**
Git Commit	1. Manages connections to remote repositories.
Git Clone	2. Copies a repository from a remote source to your local machine.
Git Push	3. Fetches and merges changes from a remote repository to your local repository.
Git Pull	4.Removes files from the working directory and stages the removal for the next commit
Git Remote	4.Records changes to the repository
Git Status	5. Uploads local repository content to a remote repository.
Git Config	6. Sets configuration options for Git on your local machine.
Git rm	7.Shows the current state of the repository, including tracked/untracked files and changes

3.Read the following sentences and Fill the gap with the missing word.

1. The command git _____ is used to create a new branch in Git.
2. A Git _____ is a pointer to a specific commit in the repository's history.
3. git checkout _____ is used to switch between branches in Git.
4. To list all branches in a Git repository, you can use git branch _____.
5. A Git _____ is a copy of a repository that lives on your computer instead of on a website's server.
6. git merge _____ is used to integrate changes from one branch into another.
7. git remote _____ is used to manage connections to remote repositories.
8. git push origin _____ is used to push the changes of the current branch to a remote repository.
9. A centralized version control system offers ……a way to collaborate using a central server.
10. Distributed version control system allows …… management branching and merging.

4 . Read this statement and answer by **true** if correct or **false** otherwise

Git Bash is a command-line interface for interacting with Git, a version control system. The git init command initializes a new, empty repository, while git clone is used to clone an existing remote repository. The git status command shows the state of the working directory, including untracked files, and git version displays the installed Git version.

a) Git bash is one among version control system that exist
b) Git init is used to clone remote repository
c) Git status is used to initialize an empty repository
d) In git you can view untracked files by using git version command

Practical assessment

Mugabe XY holds the position of Senior Developer at Innovate Company Ltd, situated in Ruhango District. He has delegated a project to four developers, tasking them with designing a web application featuring various forms: login, student registration, course registration, and book registration, all using HTML. However, due to geographical constraints and other commitments, the developers found it challenging to collaborate effectively on the project. Consequently, the Senior Developer opted to assign tasks to each developer individually and remotely, recommending them to work autonomously on their designated tasks by referring to this repository structure.

```
├── login.html
├── student_registration.html
├── course_registration.html
├── book_registration.html
├── README.md
└── .gitignore
```

As one of the developers at the company, you have been assigned the following task:
Create a remote repository on GitHub using your full name as the repository name.
Clone the repository to your local computer.
Within the repository, create the required forms mentioned above for subsequent commits.

References

JavaTpoint. (n.d.). Git Version Control System. Retrieved from https://www.javatpoint.com/git-version-control-system

JavaTpoint. (n.d.). IntelliJ IDEA Version Control. Retrieved from https://www.javatpoint.com/intellij-idea-version-control

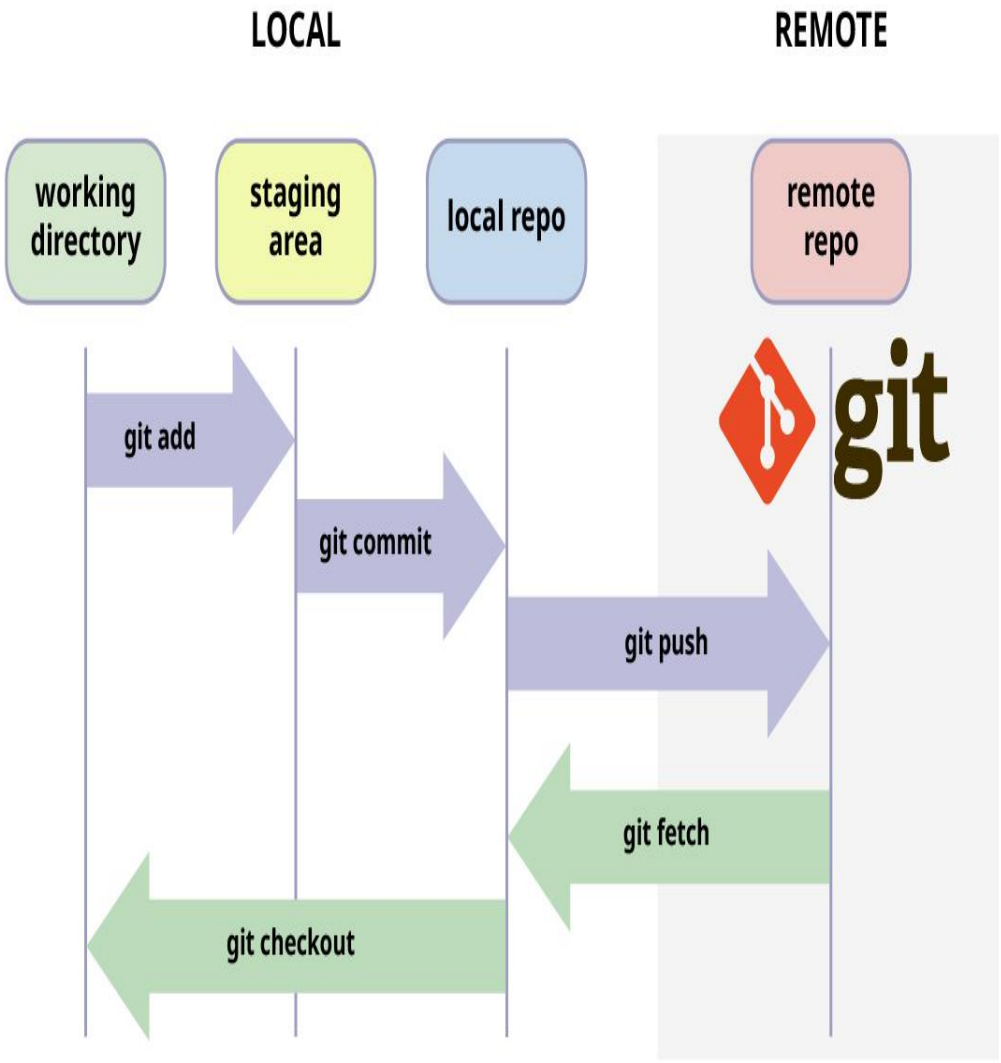
Red Hat Developers. (2023, August 2). Beginner's guide to Git version control. Retrieved from https://developers.redhat.com/articles/2023/08/02/beginners-guide-git-version-control#

JavaTpoint. (n.d.). Git. Retrieved from https://www.javatpoint.com/git

JavaTpoint. (n.d.). How to Install Git on Windows. Retrieved from https://www.javatpoint.com/how-to-install-git-on-windows

TutorialsPoint. (n.d.). Git Environment. Retrieved from https://www.tutorialspoint.com/git/git_environment.htm

JavaTpoint. (n.d.). GitHub. Retrieved from https://www.javatpoint.com/github


LOCAL
REMOTE
working directory
staging area
local repo
remote repo
git
git add
git commit
git push
git fetch
git checkout

Indicative contents
2.1: Adding file change to Git staging area **2.2: Commit file changes to Git local repository and manage branches**

Key Competencies for Learning Outcome 2: Manipulate files

Knowledge	Skills	Attitudes
• Description of git status command • Description of git commands operations • Description of commit message • Introduction of branches operations	• Applying git status commands • Performing Operation on git add command • Using staging area • Performing File management commands • Applying git commit command • Applying Operations on git branches • Adding file change to git staging area	• Being Adaptability • Being Practical oriented • Have Communication skills • Have Creativity • Have critical thinking • Being Problem solver • Have Team work

Duration:20 hrs

Learning outcome 2 objectives:

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

1. Describe clearly git status command based on git command.
2. Describe properly git command operations based on the project requirements
3. Introduce clearly operations on branches based on project requirements.
4. Add correctly file change to git staging area based on operations.
5. Apply correctly git commit command based on project content
6. Performing properly file management commands based on project requirements.

Apply clearly operations on branches based on project standard .

Resources

Equipment	Tools	Materials
● Computer	● Git ● GitHub ● Text editor (vs code) ● Terminal (CMD, Gitbash).	● Internet ● Electricity

Indicative content 2.1: Add file change to Git staging area

Duration:10 hrs

Theoretical Activity 2.1.1: Description of staging area

Tasks:

1. In small groups, you are requested to answer the following questions related to the staging area:
    I. Describe the operations of these commands:
        - git status command
        - git add command
        - git reset command
        - Rm command
2. Participate in group formulation
3. Present your findings to your classmates and trainer
4. For more clarification, read the key readings 2.1.1. In addition, ask questions where necessary.

Key readings 2.1.1.: Description of staging area

1. The staging area

In Git, staging area also known as the index, is a crucial intermediate step in the Git workflow that allows users to prepare and organize changes before committing them to the repository. When modifications are made to files in the working directory, the staging area acts as a holding area where users can selectively choose which changes to include in the next commit. By using the `git add` command, users can move specific changes from the working directory to the staging area, effectively staging them for the next commit. This process enables users to review, fine-tune, and organize their changes before creating a commit, promoting a structured and controlled approach to version control.

2. Git status command

2.1 Definition of git status command

The "git status" command is a command used in the Git version control system to displays information about the current state of the repository, such as the status of tracked and untracked files, the branch being worked on, and any changes that have been made.

Mostly, it is used to display the state between Git Add and Git commit command. We can check whether the changes and files are tracked or not.

2.2 Operations on git status command

1. **View new untracked files**:

When you run `git status`, it will display a section titled "Untracked files" that lists any new files in your working directory that Git is not currently tracking. These files have not been staged or included in any previous commits. It provides you with a list of files that you may want to consider adding to your repository.

2. **View modified files:**

In the "Changes not staged for commit" section of the `git status` output, you will see a list of modified files in your working directory that have not been staged. These files have changes compared to the last commit. Git will show you the names of the modified files so that you can review the changes made to them.

3. View deleted files:

Similarly, in the "Changes not staged for commit" section, `git status` will also display any deleted files that have not been staged. These are files that were present in the last commit but have been deleted in your working directory since then. Git will show you the names of the deleted files, allowing you to review the deletions.

The git add command is used to add changes or new files to the staging area in Git. It prepares the changes or files to be included in the next commit.

3. Git add command

3.1 Definition of git add command

The **"git add"** command is used in Git to add changes or new files to the staging area. The staging area is a temporary storage space where you can gather and prepare changes before committing them to the Git repository.

It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way changes are not actually recorded until you run git commit.

Here are a few common usages of the "git add" command:

Adding specific files: You can add specific files to the staging area by specifying their names or paths. For example, to add a file named "example.txt", you would run:

git add example.txt

Adding all changes: You can add all modified and new files in the current directory and its subdirectories to the staging area using the following command:

git add

Adding changes interactively: Git provides an interactive mode for selectively staging changes. You can use the following command to launch the interactive mode:

git add -i

This allows you to choose which changes to add by selecting them from a menu.

After using the "git add" command, the changes or files you specified will be moved to the staging area. You can then review the changes using "git status" and proceed with committing them using the "**git commit"** command.

The git add command is used to add file contents to the Index (**Staging Area**).This command updates the current content of the working tree to the staging area. It also prepares the staged content for the next commit. Every time we add or update any file in our project, it is required to forward updates to the staging area.

3.2 Operations on git add command

1. Stage all files:

When you use the **`git add .`** command, Git scans the current directory and its subdirectories, identifying any modified and new files. It adds these files to the staging area, which is a space where you can prepare your changes for the next commit.

By staging all files, you are telling Git to include all modifications and additions made to the files in the next commit. This operation allows you to commit all changes made to your project at once, without having to specify individual files.

2. Stage a file:

When you use the **`git add <file>`** command, Git adds the specified file to the staging area. This means that Git records the current state of the file, including any modifications you made to it since the last commit.

Staging a file allows you to selectively include changes from specific files in your commits. It helps you organize your commits by grouping related changes together. Only the staged files will be included in the next commit when you run **`git commit`.**

3. Stage a folder:

Git does not have a direct command to stage a folder. However, when you use the **`git add <folder>`** command, Git scans the specified folder and its subdirectories. It identifies any modified and new files within the folder and adds them to the staging area.

Staging a folder allows you to include changes from multiple files within a specific directory in a single commit. It provides a convenient way to organize your changes when you have made modifications to multiple files within a folder.

In summary, staging files using the `git add` command allows you to prepare your changes for the next commit. By staging files, you are telling Git to include those changes in the commit, making them a part of your project's history. Whether you stage all files, a specific file, or a folder, it helps you manage and organize your changes effectively before committing them to your repository.

4. Git reset command

4.1 Definition of git reset command

The **"git reset"** command is used in Git to move the HEAD and branch pointer to a specific commit or to unstage changes. It allows you to manipulate the commit history and the staging area. The behavior of "git reset" varies depending on the options and arguments used.

Here are a few common usages of the "git reset" command:

1. **Resetting the HEAD and branch pointer**: You can move the branch pointer and the HEAD to a specific commit, effectively discarding commits. For example, to reset the branch pointer to a commit identified by its SHA-1 hash, you would run:

   git reset <commit-SHA>

   By default, "git reset" moves the branch pointer to the specified commit and leaves the changes made in the discarded commits as uncommitted modifications.
2. **Unstaging changes:** If you have added changes to the staging area using "git add" and you want to remove them from the staging area, you can use the "--mixed" option with "git reset". This option is the default behavior if you don't specify any mode. For example: **git reset HEAD**

   This command moves the changes from the staging area back to the working directory, leaving the commit history and working directory unchanged.
3. **Discarding changes:** To completely discard changes in both the working directory and the staging area, you can use the "--hard" option with "git reset". For example: **git reset --hard HEAD**

   This command resets the branch pointer, the staging area, and the working directory to the specified commit, effectively discarding all changes made after that commit.

It's important to note that the "git reset" command modifies the commit history, so caution should be exercised when using it. It's recommended to create a backup or ensure that you have a clear understanding of the consequences before using it.

4.2 Operations on git reset command

1. Unstage a file:

When you want to remove a file from the staging area without discarding the changes made to that file, you can use the **`git reset`** command. Specifically, the **`git reset <file>`** command allows you to unstage a specific file.

By running **`git reset <file>`,** Git moves the specified file from the staging area back to the working directory. This means that the file is no longer marked for the next commit. However, the changes made to the file are preserved, allowing you to make further modifications or stage it again later if needed.

Unstaging a file with **`git reset`** provides flexibility in managing your staged changes, allowing you to selectively remove files from the staging area while retaining the modifications made to them.

2. Deleting and staging a folder:

Git does not have a direct command to delete or stage a folder. However, you can achieve the desired effect by combining different Git commands, including **`git reset`**.

To delete a folder and stage the deletion, you can follow these steps:

1. Use the **`git rm -r <folder>`** command to recursively remove the folder and its contents from both the working directory and the Git repository. This permanently deletes the folder and its files.

2. Run **`git reset`** to unstage the deletion. For example, **`git reset <folder>` or `git reset** .` (to unstage all changes).

3. At this point, the folder deletion is unstaged, and the folder and its contents still exist in your working directory. You can choose to commit the deletion or make further modifications before committing.

By combining **`git rm -r`** and **`git reset`,** you can effectively delete a folder and stage the deletion, providing control over the removal of folders from your repository.

Using the `git reset` command with these operations allows you to unstage files from the staging area while preserving their changes and delete folders from the repository while having the option to modify or stage them again before committing. It offers flexibility in managing your staging and deletion actions within Git.

5. Rm command

5.1 Definition of rm command

The "git rm" command is used in Git to remove files from the Git repository. It is primarily used to delete files that are tracked by Git and stage the deletion for the next commit.

Here are a few common usages of the **"git rm"** command:

1. Removing a file from the repository: You can use the following command to remove a file from the Git repository and stage the deletion: **git rm <file>**
   Replace **<file>** with the name or path of the file you want to remove. This command not only removes the file from the current working directory but also stages the deletion for the next commit.
2. Removing a file from the repository without deleting it locally: If you want to remove a file from the Git repository but keep it in your local working directory, you can use the "--cached" option with "git rm". For example: **git rm --cached <file>**
   This command removes the file from the repository but leaves it intact in your local working directory. The file will be untracked, and future changes to the file will not be tracked by Git.

3. Removing multiple files or using shell patterns: You can use shell patterns or specify multiple files to remove multiple files at once. For example:

git rm *.txt

This command removes all files with the **".txt"** extension from the repository and stages the deletions.

After using the **"git rm"** command, the file(s) will be removed from the repository, and you will need to commit the changes using **"git commit"** to make the deletion permanent in the Git history.

5.2 Operations on rm command

1. Remove and stage a file:

The `**git rm**` command allows you to remove a file from both your working directory and the Git repository, while also staging the removal. When you run the following command:

git rm <file>

Replace `<file>` with the name or path of the file you want to remove.

This operation removes the specified file from your working directory and stages the removal, which means that the file will be marked for deletion in the next commit. The file will no longer be present in your repository or working directory after you commit the changes.

Removing and staging a file using `git rm` is useful when you want to permanently delete a file from your project and include the deletion in the commit history.

2. Remove and stage a folder:

Git does not have a direct command to remove and stage a folder. However, you can achieve the desired effect by combining different Git commands, including the `git rm` command.

To remove and stage a folder, you can follow these steps:

1. Use a file system command (e.g., **`rm -r`** on Unix-like systems or `rd /s /q` on Windows) to remove the folder and its contents from your working directory.
2. Run **`git rm -r --cached <folder>`** to stage the removal of the folder:

   git rm -r --cached <folder>

   Replace `<folder>` with the name or path of the folder you want to remove.

   This command removes the specified folder from the Git repository and stages the removal. The `--cached` flag ensures that the folder is only removed from the Git repository, not from your working directory.

Practical Activity 2.1.2: Adding file change to Git staging area

Task:

1: Read key reading 2.1.2 and ask clarification where necessary
2: Referring to the previous theoretical activity (2.1.1) you are requested to go to the computer lab to add file change to git staging area. This task should be done individually.
3: Apply safety precautions.
4: Present out the steps to add file change to git staging area.
5: Referring to the steps provided on task 3, add file change to git staging area
6: Present your work to the trainer and whole class

Key readings 2.1.2: Applying Git commands to change files

1. Adding file change to staging area.

To add file changes to the Git staging area, follow these steps:

1. Check the Status:

- Before adding file changes, check the status of your repository using **`git status`** to see which files have been modified.

2. Add Changes:

- Use the **`git add`** command followed by the filename to stage specific changes. For example, to stage a single file, use `git add <filename>`.

3. Stage All Changes:

- To stage all changes in the working directory, you can use `git add ` to add all modified files or `git add --all` to include all changes, including deleted files.

4. Review Staged Changes:

- Verify the changes staged for commit by running `git status` again. The staged changes will be listed under "Changes to be committed."

5. Commit Staged Changes:

- Once the desired changes are staged, commit them to the repository using `git commit -m "Your commit message"` to create a new commit with the staged modifications.

2. Operations on git status command

2.1 View new untracked file

To view new untracked files in Git, you can use the `git status` command. Here are the steps to do it practically:

1. Open your **terminal** or **command prompt**.

2. Navigate to the root directory of your Git repository using the **`cd` command**. For **example:**

cd /path/to/your/repository

3. Run the `**git status**` command:

git status

This command will display the current status of your repository, including any untracked files.

4. Look for the section labeled "Untracked files". It will list all the files in your working directory that are not tracked by Git.

For example:

Untracked files:

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

new_file.txt

In this example**, `new_file.txt`** is an untracked file.

By running `**git status`,** you can easily see the list of untracked files in your repository. This helps you identify any new files that Git is not currently tracking. If you want to include these untracked files in your Git repository, you can use the `git add` command to stage them for the next commit.

Example2 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
```

From that image before, there is no untracked files

Let us create a file called **"studentregistration.html"** inside our repository by using **touch Flile 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
```

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

2.2 View modified file

Let's assume you have a Git repository with some files, and you have made changes to one of the files. Follow these steps to view the modified file:

1. Open your terminal or command prompt.

2. Navigate to the root directory of your Git repository using the `cd` command. For example: cd /path/to/your/repository

3. Run the `git status` command:

This command will display the current status of your repository, including any modified files.

4. Look for the section labeled "Changes not staged for commit". It will list all the modified files in your working directory.

For example:

Changes not staged for commit:

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

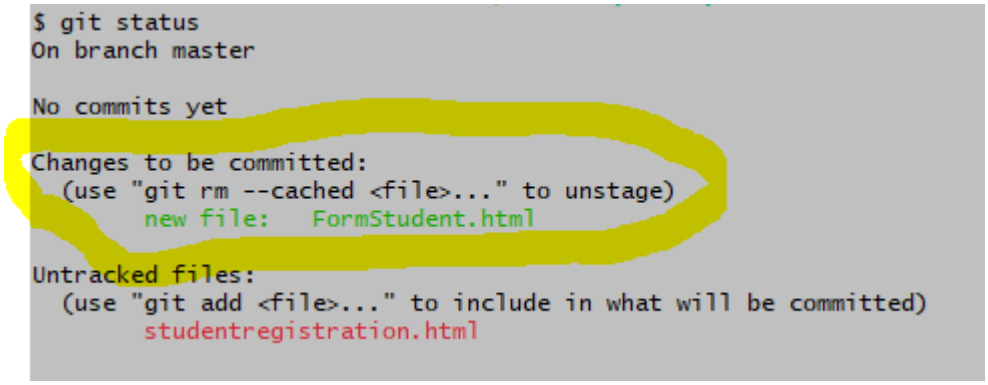
(use "git restore <file>..." to discard changes in working directory)

modified: file.txt

In this example, `file.txt` is the modified file.

By running **`git status`,** you can easily identify the modified file in your repository. Git will display the file name and indicate that it has been modified. This allows you to keep track of changes made to your files and helps you decide how to proceed, such as staging the modifications for the next commit using the `git add` command

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



2.3 View deleted file

To view deleted files in Git, you can use the `git status` command. Here's a clear example of how to do it practically:

Assuming you have a Git repository and you have deleted a file. Follow these steps to view the deleted file:

1. Open your terminal or command prompt.

2. Navigate to the root directory of your Git repository using the `cd` command. For example:

cd /path/to/your/repository

3. Run the `**git status**` command:

git status

This command will display the current status of your repository, including any deleted files.

4. Look for the section labeled "Changes not staged for commit". It will list all the deleted files in your working directory.

For example:

Changes not staged for commit:

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

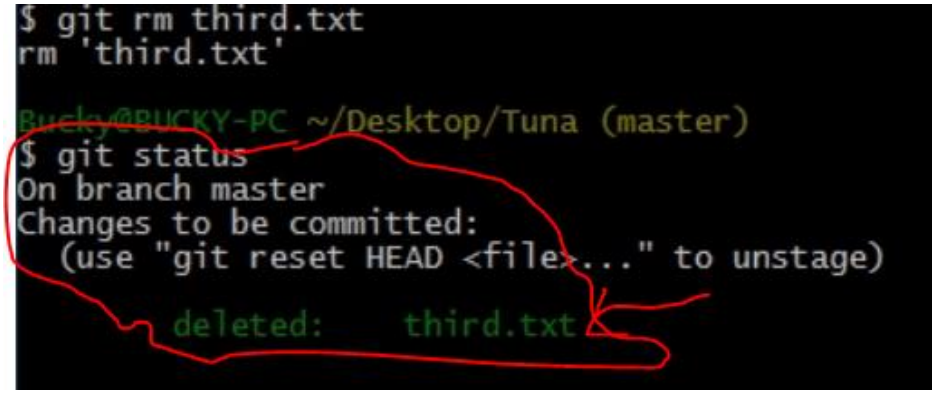
(use "git restore <file>..." to discard changes in working directory)

deleted: deleted_file.txt

In this example, `deleted_file.txt` is the deleted file.

By running `git status`, you can easily identify the deleted file in your repository. Git will display the file name and indicate that it has been deleted. This helps you keep track of the changes made to your files and allows you to decide how to proceed, such as committing the deletion using the appropriate Git command.

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.**



3.Operation on git add command

3.1 Stage all files

Git add files

Git add command is a straight forward command. It adds files to the staging area. We can add single or multiple files at once in the staging area. It will be run as:

$ git add <File name>

The above command is added to the git staging area, but yet it cannot be shared on the version control system. A commit operation is needed to share it. Let's understand the below scenario.

We have created a file for our newly created repository in **NewDirectory**. To create a file, use the touch command as follows:

$ touch newfile.txt

And check the status whether it is untracked or not by git status command as follows:

$ git status

The above command will display the untracked files from the repository. These files can be added to our repository. As we know we have created a newfile.txt, so to add this file, run the below command:

$ git add newfile.txt

Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ touch newfile.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master

No commits yet

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

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

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git add newfile.txt
```

From the above output, we can see **newfile.txt** has been added to our repository. Now, we have to commit it to share on Git.

3.2 Git Add All

We can add more than one files in Git, but we have to run the add command repeatedly. Git facilitates us with a unique option of the add command by which we can add all the available files at once. To add all the files from the repository, run the add command with **-A** option. We can use '.' Instead of **-A** option. This command will stage all the files at a time. It will run as follows:

$ git add -A

Or

$ git add .

The above command will add all the files available in the repository. Consider the below scenario:

We can either create four new files, or we can copy it, and then we add all these files at once. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git add newfile.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ touch newfile1.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ touch newfile2.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ touch newfile3.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master

No commits yet

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

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        newfile1.txt
        newfile2.txt
        newfile3.txt
```

In the above output, all the files are displaying as untracked files by Git. To track all of these files at once, run the below command:

$ git add -A

The above command will add all the files to the staging area. Remember, the **-A** option is case sensitive. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git add -A

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   newfile.txt
        new file:   newfile1.txt
        new file:   newfile2.txt
        new file:   newfile3.txt
```

In the above output, all the files have been added. The status of all files is displaying as staged.

3.3 Adding all files by extension

In some cases, you may be interested in adding all files that have a specific extension: ***.txt** or ***.js** for example.

To add all files having a specific extension, you have to use the **"git add"** command followed by a wildcard and the extension to add**.**

$ **git add *.txt**

$ **git add *.js**

As an example, let's say that you have created two Javascript files and one text file.

```
C:\git-add (feature -> origin)
λ git status
On branch feature
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file.js
        folder/script.js
        instructions.txt

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

In order to add all Javascript files, we are going to **use the wildcard syntax followed by "*.js".**

$ git add *.js

```
C:\git-add (feature -> origin)
λ git add *.js

C:\git-add (feature -> origin)
λ git status
On branch feature
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file.js
        new file:   folder/script.js

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

Using dot with git add

the "." symbol stands for "**current directory** "

As a consequence, if you don't use it at the top of your project hierarchy, you will only add files in the current working directory.

To illustrate this concept, let's say that you have two new files: "**root-file**" in your project top folder and "**new-file**" in a folder named "**folder**".

```
C:\git-add (feature -> origin)
λ git status
On branch feature
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        folder/
        root-file

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

If you navigate to your new folder and execute the **“git add”** command with the dot syntax, **you will notice that you only add files located in this directory.**

$ **cd folder**

$ git add

```
C:\git-add\folder (feature -> origin)
λ git add .

C:\git-add\folder (feature -> origin)
λ git status
On branch feature
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   new-file

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

As a consequence, you might miss some of your files in your commit.

To avoid this problem, you can use the dot syntax combined with the absolute path to your project top folder.

$ **git add <path>/.**

```
C:\git-add\folder (feature -> origin)
λ git add C:\git-add\.

C:\git-add\folder (feature -> origin)
λ git status
On branch feature
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   new-file
        new file:   ../root-file
```

3.4 Stage a file

The operation of staging a file using the **"git add"** command allows you to selectively include changes made to a specific file in the staging area. To stage a file, you specify the path to that file as an argument to the **"git add"** command. For example:

git add path/to/file.txt

This command stages the changes made to the "file.txt" file, enabling you to include those changes in the next commit. Staging a file individually allows you to isolate and commit specific modifications without affecting other files in your working directory.

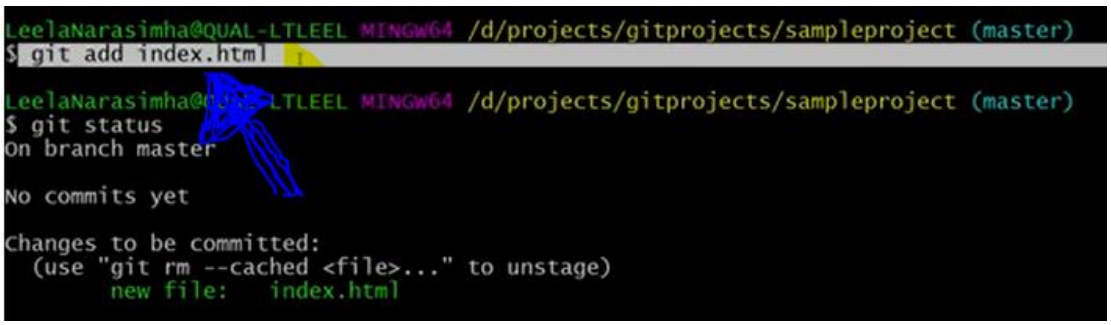
Before staging a file, you have to check all existing files and unstagged files for existing files you use **ls command**.

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

For stagging a single file use **git add filename.extension**

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

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



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

3.5 Stage folder

The operation of staging a folder using the **"git add"** command enables you to include changes made to multiple files within a directory and its subdirectories in the staging area. By executing the command:

git add path/to/folder/

To stage a folder in Git, you can use the **`git add`** command with the folder's path. Here's how you can do it practically:

1. Open your terminal or command prompt.

2. Navigate to the root directory of your Git repository using the **`cd`** command. For example:

cd /path/to/your/repository

3. Run the following command to stage a folder:

git add <folder>

Replace `<folder>` with the name or path of the folder you want to stage.

For example, if you want to stage a folder named "myfolder", the command would be:

git add myfolder

This command stages the specified folder and all its contents for the next commit.

4. Verify the staged changes by running the `git status` command:

git status

You will see that the folder and its contents are now staged for the next commit.

For example:

Changes to be committed:
(use "git restore --staged <file>..." to unstage)
new file: myfolder/file1.txt
modified: myfolder/file2.js
deleted: myfolder/file3.html

By using `git add <folder>`, you can stage a specific folder and its contents in your repository. This allows you to include all changes within that folder in a single commit.

Remember to review the changes and commit them using the appropriate Git command when you are ready.

4. Operations on git reset command

Git Reset

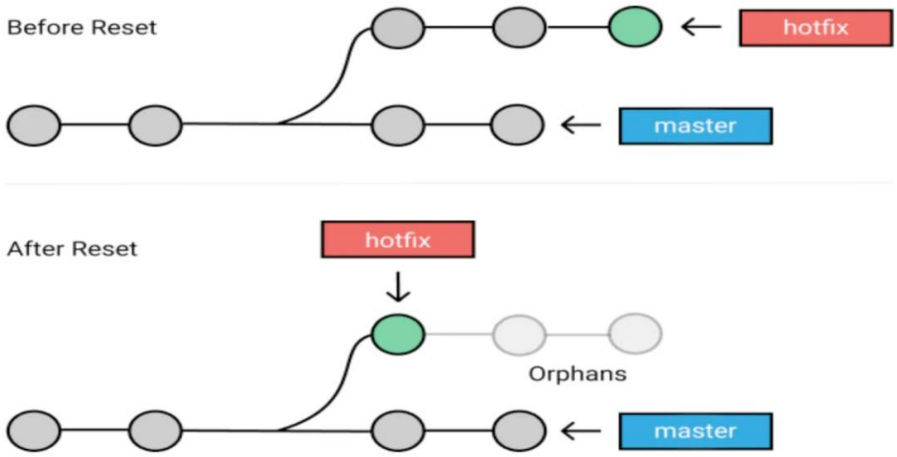


Figure 10: Git Reset diagram

The **git reset** command is used to undo the changes in your working directory and get back to a specific commit while discarding all the commits made after that one.

For instance, imagine you made ten commits. Using git reset on the first commit will remove all nine commits, taking you back to the first commit stage.

Before using **git reset**, it is important to consider the type of changes you plan to make; otherwise, you will create more chaos than good.

git reset repositoryName

The default option is git reset --mixed, which updates the current branch tip and moves anything in the staging area back to the working directory. We'll take a closer look at all three, but first let's create a basic Git repo with the following structure and show a simple git reset command in action:

git-reset-repositoryName/

file1.ext

dir1/

dir1file1.ext

Assuming we have already made our initial commit, let's add some text to file1.ext and dir1file1.ext and stage and commit them both in separate commits.

Next, let's make one more change to file1.ext and only stage the changes (but not commit), then we'll run a git log followed by a git status to check out the state of things:

$ **git log --oneline**

d66f707 (HEAD -> master) Change 2

32c2d09 Change 1

38e2a6e Initial commit

Next let's run git status:

$ **git status**

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: file1.ext

As we can see from our output, we have a commit history reflecting three commits, along with one staged file sitting in the staging index. Note that git diff can also be useful to check the state of things. Let's run a basic git reset command and check our log and status once more:

$ **git reset**

Unstaged changes after reset:

M file1.ext

$ **git log --oneline**

d66f707 (HEAD -> master) Change 2

32c2d09 Change 1

38e2a6e Initial commit

$ **git status**

On branch master

Changes not staged for commit:

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

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

modified: file1.ext

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

A quick glance at the output tells us that a basic git reset without a specific commit parameter left the commit history unchanged, while unstaging our modified file and moving it back to the working directory.

It's important to note that our changes still exist, as the working directory was left untouched. Git reset merely moved the change out of the staging area because it defaults to the --mixed option, in contrast with --hard which will wipe out the changes in the working directory as well.

4.1 Unstage a file

In Git, to unstage a file that you've previously added to the staging area, you can use the "git reset" command. Here's how you can do it:

1. First, ensure you are in the root directory of your Git repository.
2. To check the status of your repository and see which files are staged and unstaged, use the following command:

   git status

3. If you see the file you want to unstage under the "Changes to be committed" section (staged changes), you can unstage it using the following command:
   git reset HEAD <file>
   For example, if you want to unstage a file named "example.txt," you would run:
   git reset HEAD example.txt
   This command moves the changes from the staging area back to the working directory, effectively "unstaging" the file.
4. After unstaging the file, you can verify its status using **git status** again.

Example:

4.2 Deleting and staging file/folder

Deleting and staging a file/folder involves two separate steps: deleting the file/folder from your working directory and then staging the deletion for the next commit. Here's how you can do it:

1. To delete a file from your working directory, you can use the standard file system command, such as **rm** on Unix-based systems or **del** on Windows. For example, to delete a file named "example.txt"
   on Unix: rm example.txt
   On Windows: del example.txt
   Alternatively, you can use the Git command to delete the file, which will both remove it from the working directory and stage the deletion:
   git rm <file>
   For example:
   git rm example.txt

2. After deleting the file, it is removed from the working directory, but its deletion is not yet committed. To stage the deletion for the next commit, you need to run:
   git add <file>
   For example:
   git add example.txt

3. At this point, the file deletion is staged and will be part of the next commit. To complete the process, commit the changes:
   git commit -m "Deleted example.txt"
   Replace the commit message with an appropriate message for your changes.
   Please note that the **git rm** command is used to delete a file from both the working directory and the repository. If you want to keep the file in your working directory but remove it from the repository (staged deletion), you can use the **git rm --cached <file>** command instead of **git rm <file>**.

5. Operations on rm command

5.1 Remove and Stage a File:

1. First, ensure you are in the root directory of your Git repository.

2. To remove the file from both the working directory and the repository (staged deletion), use the following command:
   git rm <file>

   For example, if you want to remove a file named "example.txt":
   git rm example.txt
3. After running the **git rm** command, the file is removed from the working directory and staged for deletion. To complete the process, commit the changes:
   git commit -m "Deleted example.txt"
   Replace the commit message with an appropriate message for your changes.

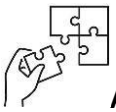
5.2 Remove and Stage a Folder (Directory):

1. First, ensure you are in the root directory of your Git repository.
2. To remove the folder and its contents from both the working directory and the repository (staged deletion), use the following command:
   git rm -r <folder>
   For example, if you want to remove a folder named "examples":
   git rm -r examples
   The **-r** flag tells Git to recursively remove the directory and its contents.
3. After running the **git rm -r** command, the folder and its contents are removed from the working directory and staged for deletion. To complete the process, commit the changes:
   git commit -m "Deleted 'examples' folder"
   Replace the commit message with an appropriate message for your changes.
   Remember that once you've committed the changes, they become a part of your Git history, and you won't be able to recover the deleted files or folders easily. So, use these commands with caution and make sure you have a backup if needed.

Points to Remember

- **Git status** is a useful command in Git that provides information about the current state of the repository and helps users track changes to files. There are operations related to `git status` including: View new untracked files, View modified files, View deleted files.
- The **`git add`** operation in Git is a fundamental command used to stage changes in a repository, preparing them to be included in the next commit. There are operations related to the `git add` command including: Stage all files, Stage a file, Stage a folder.
- **To add file changes to the Git staging area, follow these steps:**
  1. Check the Status
  2. Add Changes
  3. Stage All Changes

4. Review Staged Changes
5. Commit Staged Changes



Application of learning 2.1.

You are a software developer working on a coding project, who utilize Git for version control. While adding a new feature to the project, you need to create a file called "feature.html" and made modifications to an existing file called "main.html". To be sure that the previous tasks are performed well you can also review the changes before committing them. Using Git, check the status of your project with "git status" and identify the "feature.html" file as untracked, and "main.html" as modified. Use “git diff main.html" to view the differences in "main.html". During this process, you discover an unnecessary file and deleted it.

Indicative content 2.2: Commit File changes to git local repository and manage branch

Duration: 10hrs

- **Theoretical Activity 2.2.1: Introduction of commit file change to git local repository**

Tasks:

1. In small groups, you are requested to answer the following questions related to the commit file change to git:
   I. What do you understand about commit message?
   II. What are the best practices for creating a commit message, operations related to git log?
   III. Can you explain the operations involved in the `git commit` command in Git?
2. Participate in group formulation
3. Present your findings to your classmates and trainer
4. For more clarification, read the key readings 2.2.1. In addition, ask questions where necessary.

Key readings 2.1.1.: Description of commit file change

1. Definition of Commit message

In Git, a commit message is a brief description that explains the changes made in a particular commit. When you make changes to files in your Git repository and are ready to save those changes as a new version, you create a commit. Each commit represents a snapshot of the changes you've made to the files at a specific point in time.

The commit message serves to document the purpose and context of the changes made in that commit. It helps other developers (including your future self) to understand the reasons for the changes and the intention behind them. Writing clear and descriptive commit messages is a best practice in software development, as it promotes collaboration and makes it easier to navigate through the project history.

A typical commit message includes the following components:

1. Summary line: A concise, one-line description of the changes made in the commit. This summary line is usually limited to around 50 characters and ends with a period.
2. Description (optional): A more detailed explanation of the changes. This part is not mandatory, but it can be helpful for providing additional context and details about the commit.

Here's an example of a commit message:

Fix issue with login validation

The login form was not properly validating user credentials, leading to an error message loop. This commit fixes the issue by adding proper validation checks for username and password inputs**.**

In this example, the summary line is "Fix issue with login validation," which briefly describes the purpose of the commit. The description provides more details about the problem and the solution.

When you create a commit using the **git commit** command, your text editor will open, allowing you to write the commit message. Alternatively, you can use the **-m** flag to add the commit message directly from the command line, like this:

git commit -m "Fix issue with login validation"

It's important to follow good commit message practices to maintain a clean and informative version history for your projects. This will make it easier for you and your team to track changes, identify the purpose of each commit, and manage the development process effectively.

2. Best practice of creating a commit message

2.1 General Commit Message Guidelines

As a general rule, your messages should start with a single line that's no more than about 50 characters and that describes the change set concisely, followed by a blank line, followed by a more detailed explanation.

The same recommendations apply whether you are working on a GitHub commit, Gitlab commit, or your local git server. Follow these guidelines when writing good commit messages:

Keep it short (less than 150 characters total)

- Committing fewer changes at a time can help with this Use the imperative mood
- This convention aligns with commit messages generated by commands like git merge and git revert
- Consistency enhances speed of reading comprehension
- Tends to be more concise than the other moods

Add a title

- Less than 50 characters
- Use Title case (i.e. "Add Logging" instead of "add logging")

Add a body (optional)

- Less than 100 characters
- Explain WHAT the change is, but especially WHY the change was needed
- Leave a blank line between the title and body
- Separate paragraphs in the body with blank lines

- ❖ Use a hyphen (-) for bullet points if needed
- ❖ Use hanging indents if needed

Bad commit examples:

a. Debugging

b. I've added a delete route to the accounts controller

Good commit Examples:

a. Enable Logging Globally

b. Add Account Delete Route

c. Needed for account deletion workflow on frontend

When you use the **"git commit"** command in Git, there are several operations that take place. Here is an explanation of the operations applied on the "git commit" command:

Staging: Before you commit changes, you need to stage the modified files or new files to be included in the commit.

Creating the commit: Once you have staged the changes, the "git commit" command creates a new commit using the staged changes.

Recording the commit message: When you run "git commit," Git prompts you to provide a commit message. This message describes the changes made in the commit and serves as documentation for future reference.

Committing to the local repository: The commit is then saved to the local Git repository, which is typically located in the git directory within your project.

Creating a new commit object: Git creates a new commit object that contains a reference to the commit's parent or parents.

Advancing the branch pointer: After the commit is created, Git advances the pointer of the current branch to the new commit, making the branch point to the latest commit.

The **"git log"** command in Git allows you to view the commit history of a repository. When you run **"git log,"** several operations are performed to display a list of commits in chronological order. Here are the operations related to the "**git log"** command:

- ✓ Retrieving commit history
- ✓ Displaying commit information
- ✓ Showing commit references
- ✓ Commit filtering and formatting
- ✓ Paging and navigation

Practical Activity 2.2.2 : Committing File changes to git local repository and manage branches

Tasks:

1: Read key reading 2.2.2 and ask clarification where necessary

2: Referring to the previous theoretical activity (2.2.1) you are requested to go to the computer lab to commit file changes to git local repository and manage branches. This task should be done individually.

3: Apply safety precautions.

4: Present out the steps to commit file changes to git local repository and manage branches.

5: Referring to the steps provided on task 3, commit file changes to git local repository and manage branches.

6: Present your work to the trainer and whole class.

Key readings 2.2.1: Operation on git commit command

1. Commit file change

Here are the steps to commit file changes to a Git local repository:

1.Check Status: Before committing changes, it's a good practice to review the status of your repository using the command:

```
git status
```

This will show you which files have been modified, added, or deleted.

2.Stage Changes: Use the "git add" command to stage the changes you want to commit. You can either stage specific files or stage all changes. For example:

- To stage a specific file:

```
git add filename
```

- To stage all changes:

```
git add .
```

3.Verify Staging: Confirm that the changes you want to commit are staged correctly by checking the status again:

```
git status
```

Staged changes will appear in green.

4.Commit Changes: Once you've staged the desired changes, commit them to the local repository along with a descriptive commit message using the "git commit" command:

```
git commit -m "Your commit message here"
```

Replace "Your commit message here" with a concise description of the changes you're committing

5.Verify Commit: After committing the changes, you can verify that they have been successfully committed by reviewing the commit history:

```
git log
```

This will display a list of commits, including the one you just made. Press "q" to exit the log.

2. Operations on git commit command

Commit a file

It is used to record the changes in the repository. It is the next command after the git add. Every commit contains the index data and the commit message. Every commit forms a parent-child relationship. When we add a file in Git, it will take place in the staging area. A commit command is used to fetch updates from the staging area to the repository.

The staging and committing are co-related to each other. Staging allows us to continue in making changes to the repository, and when we want to share these changes to the version control system, committing allows us to record these changes.

Commits are the snapshots of the project. Every commit is recorded in the master branch of the repository. We can recall the commits or revert it to the older version. Two different commits will never overwrite because each commit has its own commit-id. This commit-id is a cryptographic number created by **SHA (Secure Hash**

Algorithm) algorithm.

Let's see the different kinds of commits.

To apply a commit file operation in Git, you typically follow these steps:

1. **Stage your changes**: Before committing your changes, you need to stage the files you want to include in the commit. You can use the git add command to stage specific files or directories.
2. **Review the changes**: Once you've staged the changes, you can use the git status command to review the modifications you've made. This command shows the staged changes and any untracked files.
3. **Create a commit**: After reviewing the changes, you're ready to create a commit. Use the git commit command to create a new commit with a commit message describing the changes.
4. **Repeat the process**: If you have additional changes to include in the commit, you can repeat steps 1-3. Stage the new changes using git add, review the modifications with git status, and create a new commit with git commit.
5. **Push or share your commits**: Once you've finished creating your commits, you can push them to a remote repository to share them with others or keep a backup. Use the git push command to push your commits to a remote branch.

3. The git commit command

The commit command will commit the changes and generate a commit-id. The commit command without any argument will open the default text editor and ask for the commit message. We can specify our commit message in this text editor. It will run as follows:

1. **$ git commit**

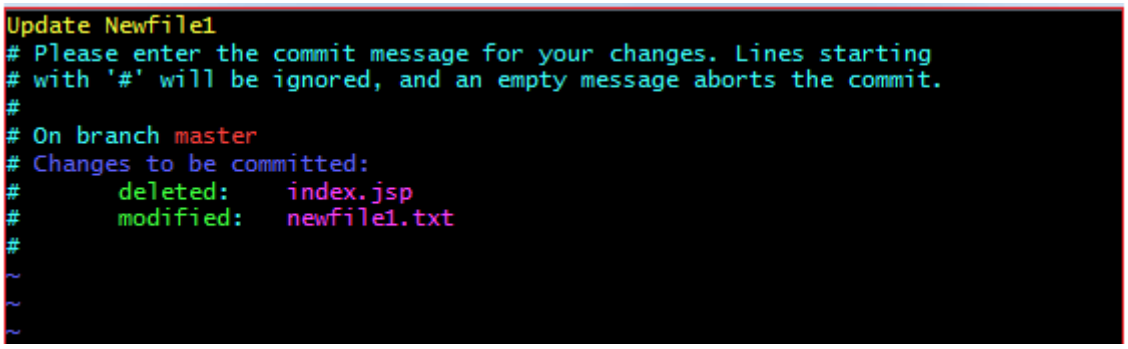
The above command will prompt a default editor and ask for a commit message. We have made a change to **newfile1.txt** and want it to commit it. It can be done as follows:

Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git commit
[master e3107d8] Update Newfile1
 2 files changed, 1 insertion(+)
 delete mode 100644 index.jsp
```

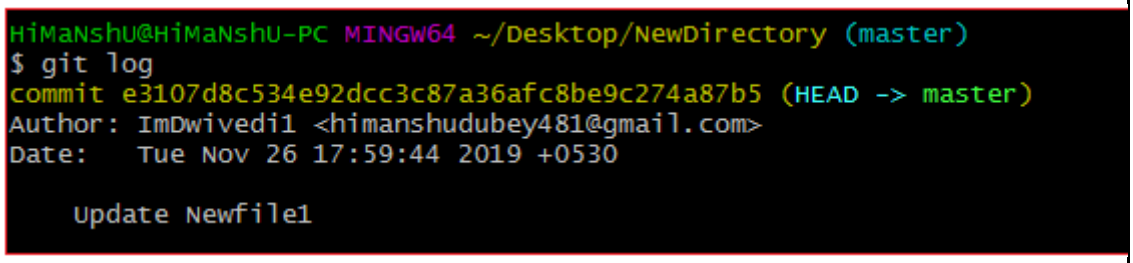
As we run the command, it will prompt a default text editor and ask for a commit

message. The text editor will look like as follows:



Press the **Esc** key and after that '**I**' for insert mode. Type a commit message whatever you want. Press **Esc** after that '**:wq**' to save and exit from the editor. Hence, we have successfully made a commit.

We can check the commit by git log command. Consider the below output:



We can see in the above output that log option is displaying commit-id, author detail, date and time, and the commit message.

Git commit -a

The commit command also provides **-a** option to specify some commits. It is used to commit the snapshots of all changes. This option only considers already added files in Git. It will not commit the newly created files.

Consider below scenario:

We have made some updates to our already staged file newfile3 and create a file **newfile4.txt**. Check the status of the repository and run the commit command as follows:

1. $ **git commit -a**

   Consider the output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ touch newfile4.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory
        modified:   newfile3.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        newfile4.txt

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

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git commit -a
[master fc66f84] updated newfile3
 1 file changed, 1 insertion(+)
```

The above command will prompt our default text editor and ask for the commit message. Type a commit message, and then save and exit from the editor. This process will only commit the already added files. It will not commit the files that have not been staged. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        newfile4.txt

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

As we can see in the above output, the newfile4.txt has not been committed.

Git commit -m

The -m option of commit command lets you to write the commit message on the command line. This command will not prompt the text editor. It will run as follows:

$ git commit -m "Commit message."

The above command will make a commit with the given commit message. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git commit -m "Introduced newfile4"
[master 64d1891] Introduced newfile4
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newfile4.txt
```

In the above output, a **newfile4.txt** is committed to our repository with a commit message.

We can also use the **-am** option for already staged files. This command will immediately make a commit for already staged files with a commit message. It will run as follows:

$ git commit -am "Commit message."

4. Edit commit message

Git Commit Amend (Change commit message)

The **git commit --amend** command is commonly used to make changes to the most recent commit in Git. It allows you to modify the commit message, add more changes, or even remove some changes that were accidentally included. Here are some examples of how **git commit --amend** can be applied:

1. **Changing the commit message:**

Suppose you made a commit with a message containing a typo, and you want to correct it. You can use **git commit --amend** to change the commit message:

Make some changes to the code

git add

git commit -m "Fixe a bug in the login process"

Oops! Realized the typo in the commit message

git commit --amend -m "Fixed a bug in the login process"

Adding forgotten changes to the previous commit:

If you forgot to include some changes in the last commit, you can use **git commit --amend** to add those changes to the previous commit:

Make some changes to the code

git add.

git commit -m "Added new feature A"

Realized that you forgot to add changes to feature B

Add the changes to the staging area

git add path/to/featureB.py

Amend the previous commit to include the changes to feature B

git commit --amend --no-edit

Splitting a commit into multiple commits:

Suppose you made several unrelated changes in a single commit and want to split it into multiple smaller commits. You can use **git commit --amend** to do that interactively:

Make several changes

git add change1.py

git add change2.py

git add change3.py

Commit all changes together

git commit -m "Added changes 1, 2, and 3"

Realized that the changes should be separate commits

Amend the commit and interactively split the changes

git add change1.py

git commit --amend

Stage and commit changes for change2.py

git add change2.py

git commit -m "Added change 2"

Stage and commit changes for change3.py

git add change3.py

git commit -m "Added change 3"

Please note that using **git commit --amend** to modify commits that have already been

pushed to a remote repository can cause issues with the history and should be used with caution. It's best to avoid amending commits that have already been shared with others.

5. Operation on git log command

We can use git log command in order to list, filter, view commit history in different ways. we will examine git log command usage in detail with examples.

To see simplified list of commit

The git log command is used to view the commit history in a Git repository. By default, it displays a detailed list of commits, including information such as commit hash, author, date, and commit message. Display a simplified list of commits with essential information:

1. Show a simplified list of commits with the author and date
2. View a simplified list of the last five commits
3. Display a simplified list of commits for a specific branch

To see a list of commits with more detail

To see a list of Git commits with more detailed information, you can use the git log command without any additional options. By default, git log provides a detailed view of the commit history.

List Commit History

We will start with git log command without any parameter. This will list all commit history in an interactive terminal where we can see and navigate.

$ git log

```
commit 434de730f5abe43c8f6f8e32247f2e04d31635f6 (HEAD -> newversion, origin/master, ori
Author: fyodor <fyodor@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Sun Dec 9 02:00:55 2018 +0000

    Update copyright year for Ncat and Ncat Guide

commit 6d420e82b2c55b6c7723c07e33771c52ad193b5e
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Sun Dec 2 05:54:58 2018 +0000

    Changelog for #1227

commit 1ba01193725f4c83bf9e4b4cd589dbc9fc626152
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Sun Dec 2 05:48:27 2018 +0000

    Add a length check for certificate parsing. Fixes #1399

commit b1efd742499b00eef970feeef84dc64f301db61f
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 20:27:05 2018 +0000

    Warn for raw scan options without needed privileges

commit b642dc129c4d349a849fb0eb055cde263d9d3eb6
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 17:42:09 2018 +0000

    Fix a bug in the fix. https://github.com/nmap/nmap/commit/ebf083cb0bfc239a000aea776

commit 350bbe0597d37ad67abe5fef8fba984707b4e9ad
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 17:42:09 2018 +0000

    Avoid a crash (double-free) when SSH connection fails
```

5.1 List Commit History

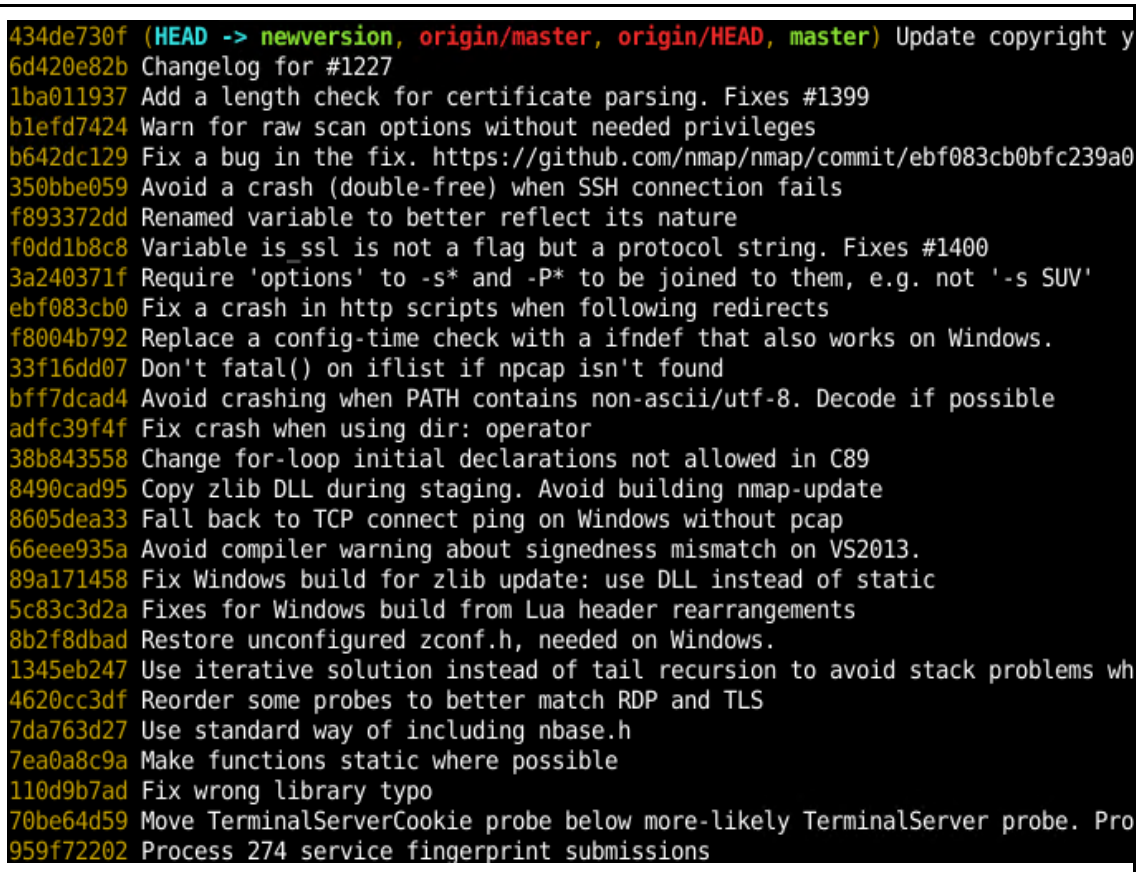
We can see from output that following information about the commit provided.

- **`Commit`** number which is a unique hash identifies the commit
- **`Author`** the developer who commit. Also email information is provided
- **`Date`** specifies when the commit occurred
- The last line provides note and information about the commit.

5.2 List One Commit Per Line

If we need to only list unique part of the commit id with the note provided by author, we can use --oneline option which will just print single line about each commit.

$ **git log --oneline**

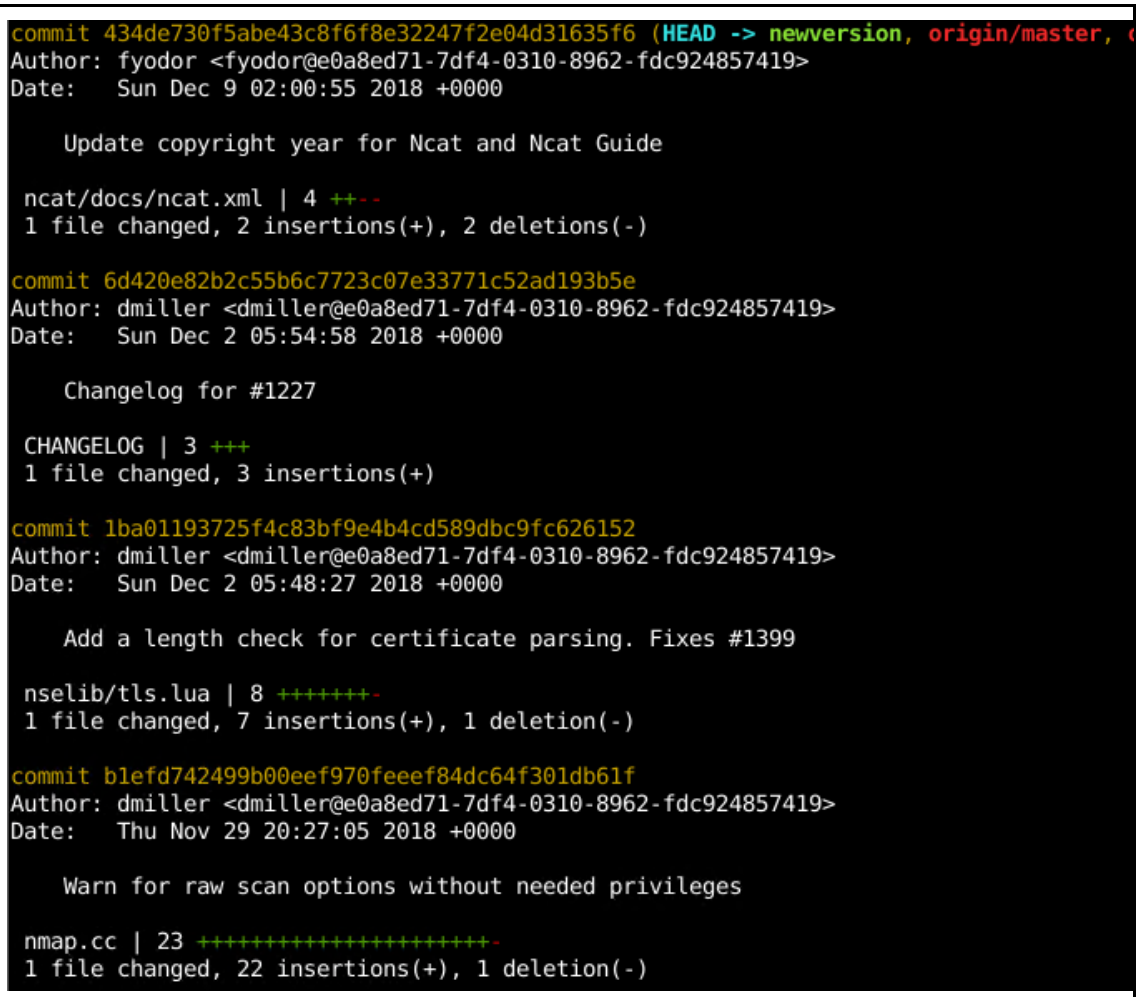


List One Commit Per Line

5.3 Print Statistics

We may need to print information about the commit in details. We will use --stat option.

$ git log --stat



5.4 Print Statistics

We can see from output that extra information like changed file, changed file count, number of lines added, number of lines deleted.

Print Patch or Diff Information

If we are interested with the code diff information we need to use -p option. -p option can be used to print path or diff of the files for the commits.

$ git log -p

```
Date:   Sun Dec 2 05:48:27 2018 +0000

    Add a length check for certificate parsing. Fixes #1399

diff --git a/nselib/tls.lua b/nselib/tls.lua
index e57a87f1e..e7e7b180b 100644
--- a/nselib/tls.lua
+++ b/nselib/tls.lua
@@ -1212,7 +1212,13 @@ handshake_parse = {
         end
         local b = {certificates = {}}
         while j < cert_end do
-          local cert_len, cert
+          local cert_len = unpack(">I3", buffer, j)
+          if cert_len + 3 + j > cert_end then
+            stdnse.debug1("server_certificate parsing error!")
+            j = cert_end
+            break
+          end
+          local cert
           cert, j = unpack(">s3", buffer, j)
           -- parse these with sslcert.parse_ssl_certificate
           table.insert(b["certificates"], cert)

commit b1efd742499b00eef970feeef84dc64f301db61f
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 20:27:05 2018 +0000

    Warn for raw scan options without needed privileges

diff --git a/nmap.cc b/nmap.cc
index fd4574701..2b351511a 100644
--- a/nmap.cc
+++ b/nmap.cc
@@ -521,6 +521,7 @@ public:
     this->advanced              = false;
     this->af                    = AF_UNSPEC;
     this->decoys                = false;
+    this->raw_scan_options      = false;
   }
```

5.5 Print Patch or Diff Information

We see from screenshot that added and removed code is shown clearly. Added code color is green and removed code is red. Also added code lines start with +plus and removed code lines starts with - minus.

Show/Print Specific Commit In Detail

If we need to look specific commit we need to use git show command. We will also provide the commit id or number we can to print.

$ git show b1efd742499b00eef970feeef84dc64f301db61f

```
commit b1efd742499b00eef970feeef84dc64f301db61f
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 20:27:05 2018 +0000

    Warn for raw scan options without needed privileges

diff --git a/nmap.cc b/nmap.cc
index fd4574701..2b351511a 100644
--- a/nmap.cc
+++ b/nmap.cc
@@ -521,6 +521,7 @@ public:
     this->advanced              = false;
     this->af                    = AF_UNSPEC;
     this->decoys                = false;
+    this->raw_scan_options      = false;
   }

   // Pre-specified timing parameters.
@@ -535,7 +536,7 @@ public:
   double pre_scripttimeout;
 #endif
   char  *machinefilename, *kiddiefilename, *normalfilename, *xmlfilename;
-  bool  iflist, decoys, advanced;
+  bool  iflist, decoys, advanced, raw_scan_options;
   char  *exclude_spec, *exclude_file;
   char  *spoofSource, *decoy_arguments;
   const char *spoofmac;
@@ -751,6 +752,7 @@ void parse_options(int argc, char **argv) {
           // If they only want open, don't spend extra time (potentially) distinguish
           o.defeat_rst_ratelimit = true;
         } else if (strcmp(long_options[option_index].name, "scanflags") == 0) {
+          delayed_options.raw_scan_options = true;
           o.scanflags = parse_scanflags(optarg);
           if (o.scanflags < 0) {
             fatal("--scanflags option must be a number between 0 and 255 (inclusive)
@@ -776,6 +778,7 @@ void parse_options(int argc, char **argv) {
```

5.6 Print Specific Commit In Detail

We can see that specific commit provides diff information in detail.

Show/Print Specific Commit Stats

If we can't to just print specific commit stat and information we can provide --stat option to the git show command.

$ **git show --stat b1efd742499b00eef970feeef84dc64f301db61f**

```
$ git show --stat b1efd742499b00eef970feeef84dc64f301db61f
commit b1efd742499b00eef970feeef84dc64f301db61f
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 20:27:05 2018 +0000

    Warn for raw scan options without needed privileges

 nmap.cc | 23 ++++++++++++++++++++-
 1 file changed, 22 insertions(+), 1 deletion(-)
```

Show/Print Specific Commit Stats

5.7 Group Commits By Author

If we want to inspect the commits according to the author name we need to group commits by author. We can use shortlog command in order to list commits notes grouped by author name.

$ **git shortlog**

```
abhishek (14):
      Closes #366
      Removes the bug so as to compare cmd fixes #381
      Adds Pong response, closes #383
      Updates the Nsock examples, closes #395
      implements map data structure to speed up search in process_result() of nmap
      Falls back to "getnameinfo" for truncated replies in reverse DNS resolver. C
      Adds zero-byte option(-z) for Ncat. Fixes #22 and #225
      Adds test for -z option in ncat, closes #444
      Add little documentation for DNS resolution of truncated packets. Closes #46
      Add support for decoys in IPv6 closes #433 and fixes #98
      Fix timeout problem for http-slowloris
      Add --script-timeout option to limit the script's runtime. Closes #330 and F
      Documentation for script-timeout. Closes #385
      Add CHANGELOG entry for --script-timeout.

aca (37):
      Small patch to rdp-vuln-ms12-020.nse
      Rewrite of ftp-brute.nse script
      Fixed a typo in the description.
      Rewrote mysql-brute to use brute library
      Commited rmi-vuln-classloader script
      Commited http-frontpage-login to main branch
      Merged dns-nsec3-enum to trunk
      removed .exe, added info to Changelog
      added Daniel's patch
      Wrong @usage descriptions fix
      Added missing library requirements to dns-nsec3-enum
      Few variables were not declared as locals. This fixes it.
      Added pcanywhere-brute script
      Typo...
      Added few aditional credentials to http-default-accounts fingerprints
      Added metasploit-msgrpc-brute to trunk
      Merged metasploit-info from my dev branch
      Brute and unpwdb lib improvements that allow more flexible iterator specific
      Added a patch by Patrick. A cleaner way to deal with varargs.
      Ooops. Forgot the returns
```

Group Commits By Author

5.8 Show Author Commit Numbers

If we are interested with the authors commit numbers we need to provide -s options to the shortlog command.This will provides commit numbers for each authors.

$ git shortlog -s

```
$ git shortlog -s
    14  abhishek
    37  aca
     1  alex
     1  andrew
   336  batrick
    42  bmenrigh
     1  claude
     7  claudiu
    22  colin
    88  d33tah
    17  daniel
  3703  david
     8  devin
    14  diman
   130  djalal
  2247  dmiller
    86  doug
     9  drazen
    16  ejlbell
     1  evangel
  2114  fyodor
    38  gio
    56  gorjan
    46  gyani
   321  henri
     5  ithilgore
    98  jah
    51  jay
     3  jiayi
    33  joao
    17  josh
     3  jurand
     3  kirubakaran
   340  kris
    78  kroosec
   125  luis
```

5.9 Show Author Commit Numbers

Sort Authors By Commit Numbers

We can improve previous example and sort authors by their commit numbers. We will add -n too the previous example where final command will be like below.

$ git shortlog -n -s

```
$ git shortlog -n -s
  3703  david
  2247  dmiller
  2114  fyodor
   475  patrik
   340  kris
   336  batrick
   321  henri
   227  nnposter
   193  paulino
   192  ron
   148  tomsellers
   130  djalal
   125  luis
    98  jah
    88  d33tah
    86  doug
    78  kroosec
    58  sophron
    56  gorjan
    52  robert
    51  jay
    46  gyani
    45  sean
    44  shinnok
    43  vincent
    42  bmenrigh
    38  gio
    37  aca
    33  joao
    31  michael
    29  perdo
    28  pgpickering
    25  sven
```

Sort Authors By Commit Numbers

5.10 Pretty Print

We can also customize the log output according to our needs. We can use --pretty option and some parameters to print different log output. In this example we will use %cn for author name %h hash value of commit and %cd for commit time.

$ git log --pretty="%cn committed %h on %cd"

```
fyodor committed 434de730f on Sun Dec 9 02:00:55 2018 +0000
dmiller committed 6d420e82b on Sun Dec 2 05:54:58 2018 +0000
dmiller committed 1ba011937 on Sun Dec 2 05:48:27 2018 +0000
dmiller committed b1efd7424 on Thu Nov 29 20:27:05 2018 +0000
dmiller committed b642dc129 on Thu Nov 29 17:42:09 2018 +0000
dmiller committed 350bbe059 on Thu Nov 29 17:42:09 2018 +0000
nnposter committed f893372dd on Tue Nov 27 20:14:55 2018 +0000
nnposter committed f0dd1b8c8 on Tue Nov 27 19:28:24 2018 +0000
dmiller committed 3a240371f on Tue Nov 27 18:12:43 2018 +0000
dmiller committed ebf083cb0 on Tue Nov 27 04:43:16 2018 +0000
dmiller committed f8004b792 on Wed Nov 21 06:23:08 2018 +0000
dmiller committed 33f16dd07 on Wed Nov 21 03:43:10 2018 +0000
dmiller committed bff7dcad4 on Thu Nov 15 16:23:32 2018 +0000
dmiller committed adfc39f4f on Thu Nov 15 05:03:46 2018 +0000
dmiller committed 38b843558 on Tue Nov 13 17:32:32 2018 +0000
dmiller committed 8490cad95 on Thu Nov 8 15:30:15 2018 +0000
dmiller committed 8605dea33 on Thu Nov 8 15:28:13 2018 +0000
dmiller committed 66eee935a on Thu Nov 8 14:52:32 2018 +0000
dmiller committed 89a171458 on Thu Nov 8 14:51:33 2018 +0000
dmiller committed 5c83c3d2a on Thu Nov 8 04:55:29 2018 +0000
dmiller committed 8b2f8dbad on Thu Nov 8 04:35:52 2018 +0000
dmiller committed 1345eb247 on Thu Nov 8 04:25:12 2018 +0000
dmiller committed 4620cc3df on Tue Nov 6 15:07:04 2018 +0000
dmiller committed 7da763d27 on Tue Nov 6 15:07:03 2018 +0000
dmiller committed 7ea0a8c9a on Tue Nov 6 15:07:02 2018 +0000
dmiller committed 110d9b7ad on Tue Nov 6 15:07:01 2018 +0000
dmiller committed 70be64d59 on Mon Nov 5 18:12:12 2018 +0000
dmiller committed 959f72202 on Mon Nov 5 18:08:58 2018 +0000
dmiller committed 5a34fd3d8 on Mon Nov 5 18:07:42 2018 +0000
dmiller committed 824f9dcb2 on Thu Nov 1 04:35:00 2018 +0000
dmiller committed 27807aadb on Thu Nov 1 04:34:59 2018 +0000
dmiller committed c223ec5c3 on Thu Nov 1 04:34:58 2018 +0000
dmiller committed 0f916ec3b on Wed Oct 31 23:44:52 2018 +0000
dmiller committed 625884e7d on Wed Oct 31 20:32:16 2018 +0000
```

Pretty Print

5.11 Filter By Author

In some cases we may need to filter commits according to the author name. We will use --author and provide the author name to filter and show only given author. In this example we will filter author named dmiller.

$ git log --author="dmiller"

```
commit 6d420e82b2c55b6c7723c07e33771c52ad193b5e
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Sun Dec 2 05:54:58 2018 +0000

    Changelog for #1227

commit 1ba01193725f4c83bf9e4b4cd589dbc9fc626152
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Sun Dec 2 05:48:27 2018 +0000

    Add a length check for certificate parsing. Fixes #1399

commit b1efd742499b00eef970feeef84dc64f301db61f
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 20:27:05 2018 +0000

    Warn for raw scan options without needed privileges

commit b642dc129c4d349a849fb0eb055cde263d9d3eb6
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 17:42:09 2018 +0000

    Fix a bug in the fix. https://github.com/nmap/nmap/commit/ebf083cb0bfc239a000aea7

commit 350bbe0597d37ad67abe5fef8fba984707b4e9ad
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 17:42:09 2018 +0000

    Avoid a crash (double-free) when SSH connection fails

commit 3a240371fcb3108bc73760e0684e63c85223d913
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Tue Nov 27 18:12:43 2018 +0000

    Require 'options' to -s* and -P* to be joined to them, e.g. not '-s SUV'

commit ebf083cb0bfc239a000aea7764cdca42f016affe
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Tue Nov 27 04:43:16 2018 +0000

    Fix a crash in http scripts when following redirects
```

Filter By Author

5.12 Filter By Number

If we want to list and print specified number of commit we need to use - with the number we want to print. In this example we will print last 5 commit.

$ git log -5 --oneline

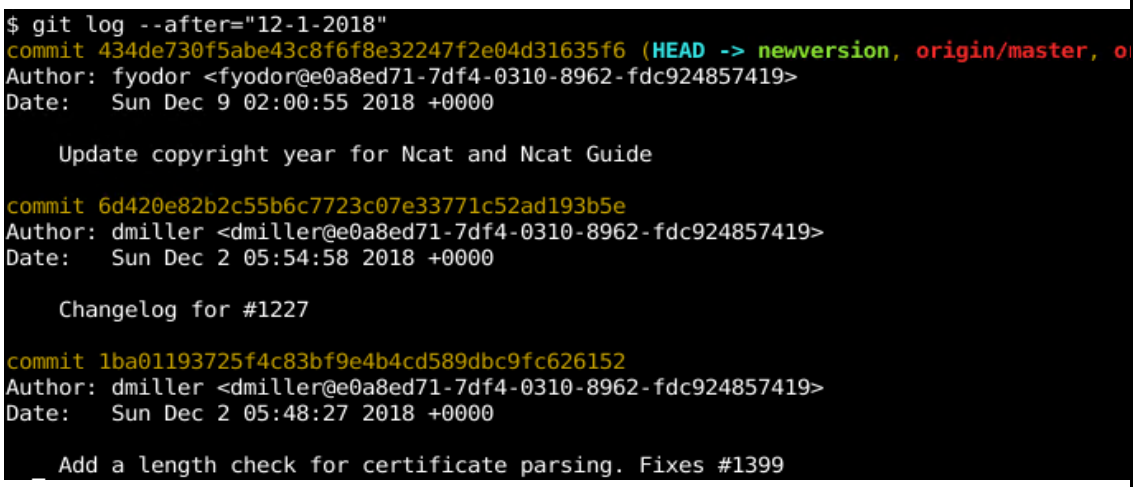
```
$ git log -5 --oneline
434de730f (HEAD -> newversion, origin/master, origin/HEAD, master) Update copyright year
6d420e82b Changelog for #1227
1ba011937 Add a length check for certificate parsing. Fixes #1399
b1efd7424 Warn for raw scan options without needed privileges
b642dc129 Fix a bug in the fix. https://github.com/nmap/nmap/commit/ebf083cb0bfc239a000a
```

Filter By Number

5.13 Filter By Date

We can also filter according to date. We will provide the date we want to start listing. We will use --after option and provide the date. Date will be MM-DD-YYYY format. In this example we will list commits those created after 1 December 2018.

$ **git log --after="12-1-2018"**



Filter By Date

We can also use --before where commits created before specified date will be printed.

5.14 Filter By Message

We can print or list logs by filtering according to the message. We will use --grep option and provide the filter term. We will filter for message http in this example.

$ **git log --grep="http"**

```
commit b642dc129c4d349a849fb0eb055cde263d9d3eb6
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 17:42:09 2018 +0000

    Fix a bug in the fix. https://github.com/nmap/nmap/commit/ebf083cb0bfc239a000aea776

commit ebf083cb0bfc239a000aea7764cdca42f016affe
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Tue Nov 27 04:43:16 2018 +0000

    Fix a crash in http scripts when following redirects

commit 324965d1d221c72ebc962bc2871732bc15b3ba22
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Wed Oct 31 14:01:34 2018 +0000

    Use a radix tree (trie) to match exclude addresses

    Current exclusions list from --excludefile takes linear time to match
    against. Using a trie structure, we can do matching in O(log n) time,
    with a hard maximum of 32 comparisons for IPv4 and 128 comparisons for
    IPv6. Each node of the trie represents an address prefix that all
    subsequent nodes share; matching stops when one is matched exactly or
    when the candidate address does not match any prefix of the addresses in
    the trie.

    For now, only numeric addresses without netmask are supported. We plan
    to extend this to addresses with netmasks, including resolved names.
    Storing IPv4 ranges and wildcards in this structure would be
    prohibitively complex, so the existing linear match method will be used
    for those. It is unlikely that any users are using large exclusion lists
    of these types of specifications, so performance impact is small.

    Potential future features could use the trie structure to implement
    custom routing or scope-limiting.

    This was a todo list item based on this report:
    https://seclists.org/nmap-dev/2012/q4/420

commit de2b08e27a997517c00ad0bcb5039660b22aaacd
Author: paulino <paulino@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Fri Oct 19 05:00:46 2018 +0000

    Adds http-sap-netweaver-leak to detect SAP instances with the Knowledge Management

commit 93edeefa3c7bf8d55928ecf64c5abef82e28a904
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Wed Oct 17 20:21:05 2018 +0000
```

Filter By Message

5.15 Filter By File

If we are looking for specific file change during commit we can filter for file. We will use -- and provide file names which is expected to be in commit change. In this example we will look file ip.c which is expected to be committed.

$ **git log -- ip.c**

Filter By Content

Also we can filter commits according to the commit content. This will be very useful if we want to search and filter for specific change. We will use -S option and provide filter term. In this example we will filter for raw_scan. Keep in mind that this may take some time because it will search in all commits which is not indexed for fast search.

$ **git log -S"raw_scan"**

```
$ git log -S"raw_scan"
commit b1efd742499b00eef970feeef84dc64f301db61f
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 20:27:05 2018 +0000

    Warn for raw scan options without needed privileges
```

Filter By Content

5.16 Filter By Commit Id/Hash Range

Commits have their own hash ids. If we want to list range of Commits we can provide the start and end commit id where commits between them will be listed.

$ git log b642dc129c4d349a849fb0e..1ba01193725f4c

```
$ git log b642dc129c4d349a849fb0e..1ba01193725f4c
commit 1ba01193725f4c83bf9e4b4cd589dbc9fc626152
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Sun Dec 2 05:48:27 2018 +0000

    Add a length check for certificate parsing. Fixes #1399

commit b1efd742499b00eef970feeef84dc64f301db61f
Author: dmiller <dmiller@e0a8ed71-7df4-0310-8962-fdc924857419>
Date:   Thu Nov 29 20:27:05 2018 +0000

    Warn for raw scan options without needed privileges
```

Filter By Commit Id/Hash Range

5.17 List Only Merges

By default merge commits are printed and listed. But if the default behavior is change with config and we want to list and print merge commits we can use --merge option to list merge commits too.

$ git log –merge

List No Merges

By default merges commits are printed and listed with git log command. If we do not want to list or print then for all operations we can use --no-merges option which will

do not show merge commits.

$ **git log --no-merge**

6. Manage branches

6.1 Git Branch

A branch is a version of the repository that diverges from the main working project. It is a feature available in most modern version control systems. A Git project can have more than one branch. These branches are a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug, you spawn a new branch to summarize your changes. So, it is complex to merge the unstable code with the main code base and also facilitates you to clean up your future history before merging with the main branch.

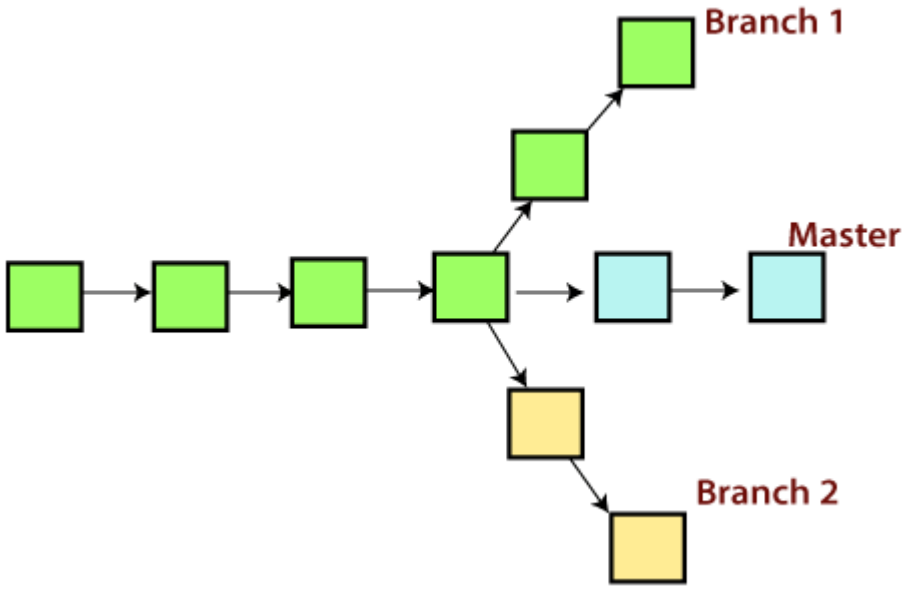


Figure 11: Git Master Branch

The master branch is a default branch in Git. It is instantiated when first commit made on the project. When you make the first commit, you're given a master branch to the starting commit point. When you start making a commit, then master branch pointer automatically moves forward. A repository can have only one master branch.

Master branch is the branch in which all the changes eventually get merged back. It can be called as an official working version of your project.

6.2 Operations on Branches

We can perform various operations on Git branches. The **git branch command** allows you to **create**, **list**, **rename** and **delete** branches. Many operations on branches are

applied by git checkout and git merge command. So, the git branch is tightly integrated with the **git checkout** and **git merge commands**.

The Operations that can be performed on a branch:

1.Create Branch

You can create a new branch with the help of the **git branch** command. This command will be used as:

Syntax:

1. $ **git branch <branch name>**

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch B1
```

This command will create the **branch B1** locally in Git directory.

2.List Branch

You can List all of the available branches in your repository by using the following command.

Either we can use **git branch - list** or **git branch** command to list the available branches in the repository.

Syntax:

1. **$ git branch --list or $ git branch**

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch
  B1
  branch3
* master

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch --list
  B1
  branch3
* master
```

Here, both commands are listing the available branches in the repository. The symbol * is representing currently active branch.

3.Delete Branch

You can delete the specified branch. It is a safe operation. In this command, Git

prevents you from deleting the branch if it has unmerged changes. Below is the command to do this.

Syntax:

1. $ git branch -d**<branch** name**>**

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch -d B1
Deleted branch B1 (was 554a122).
```

This command will delete the existing branch B1 from the repository.

The **git branch d** command can be used in two formats. Another format of this command is **git branch D**. The '**git branch D**' command is used to delete the specified branch.

1. $ **git branch -D <branch name>**

Delete a Remote Branch

You can delete a remote branch from Git desktop application. Below command is used to delete a remote branch:

Syntax:

1. **$ git push origin -delete <branch name>**

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git push origin --delete branch2
To https://github.com/ImDwivedi1/GitExample2
 - [deleted]         branch2

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

As you can see in the above output, the remote branch named **branch2** from my GitHub account is deleted.

4.Switch Branch

Git allows you to switch between the branches without making a commit. You can switch between two branches with the **git checkout** command. To switch between the branches, below command is used:

1. **$ git checkout<branch name>**

Switch from master Branch

You can switch from master to any other branch available on your repository without making any commit.

Syntax:

1. **$ git checkout <branch name>**

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git checkout branch4
Switched to branch 'branch4'
```

As you can see in the output, branches are switched from **master** to **branch4** without making any commit.

Switch to master branch

You can switch to the master branch from any other branch with the help of below command.

Syntax:

1. $ git branch -m master

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (branch4)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

As you can see in the above output, branches are switched from **branch1 to master** without making any commit.

5.Rename Branch

We can rename the branch with the help of the **git branch** command. To rename a branch, use the below command:

Syntax:

1. $ git branch -m **<old** branch name**><new** branch name**>**

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch -m branch4 renamedB1

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch
* master
  renamedB1

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

As you can see in the above output, **branch4** renamed as **renamedB1**.

6.Merge Branches

Git allows you to merge the other branch with the currently active branch. You can merge two branches with the help of **git merge** command. Below command is used to merge the branches:

Syntax:

1. $ git merge **<branch** name**>**

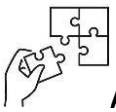
Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git merge renamedB1
Already up to date.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

From the above output, you can see that **the master** branch **merged** with **renamedB1**. Since I have made no-commit before merging, the output is showing as already up to date.

Points to Remember

- When using the `git commit` command in Git, several operations are applied to creat e a new commit in the repository. There are operations involved in the `git commit` c ommand: Staging, Creating the commit, recording the commit message, committing to the local repository, advancing the branch pointer.
- **Here are the steps to commit file changes to a Git local repository:**
  1. Check Status
  2. Stage Change
  3. Verify Staging
  4. Commit Changes
  5. Verify Commit



Application of learning 2.2.

In a team project, Alice takes charge of creating a new branch called **feature-xyz** to develop a new feature. She wants to manage her changes and review the commit history to ensure that her work is well-documented. Meanwhile, Bob, another team member, is working on a separate feature on the **feature-abc** branch. Both they want to keep track of available branches and switch between them. Alice periodically merges the latest changes from the main branch, supervised by Sarah, the project lead, to ensure her branch remains up to date. Together, Alice, Bob, and Sarah effectively collaborate using Git's powerful features to manage their workflow, track progress, and maintain code integrity.

Learning outcome 2 end assessment

Theoretical assessment

1. Match the Git branch operations with their corresponding commands:

Operation	**Command**
Create branch	a. git branch <branch-name>
List branch	b. git branch
Delete local branch	c. git branch -d <branch-name>
Delete remote branch	d. git push origin --delete <branch-name>
Switch branch	e. git checkout <branch-name>
Rename branch	f. git branch -m <old-branch-name> <new-branch-name>

2. Complete the sentence:

I. Before staging a file, you must check all.....files and.....files.
II. To create a new branch in Git, you use the command git _____ <branch-name>.
III. To list all branches in a Git repository, you use the command git _____.
IV. To delete a local branch, you use the command git _____ <branch-name>.
V. To delete a remote branch, you use the command git _____ origin --delete <branch-name>.
VI. To switch to a different branch, you use the command git _____ <branch-name>.
VII. To rename a branch, you use the command git _____ <old-branch-name> <new-branch-name>.

3. How do I add files to a commit?

- ✓ $ git stage
- ✓ $ git commit
- ✓ $ git add
- ✓ $ git reset

4. How to save the current state of your code in git?

- ✓ By validating the modifications staged with $ git commit
- ✓ By adding all the changes and staging them with $ git stage
- ✓ By adding all the changes and organizing them with $ git add
- ✓ By creating a new commit with $ git init

5. Read the Following statement and answer by true if correct or false otherwise

The git add command is crucial for staging changes, whether they involve new files or modifications to existing ones, preparing them for the next commit. Once staged, the git commit command is used to permanently save these changes to the local repository, typically requiring a commit message for clarity. Additionally, the git commit command

can be employed in the process of merging branches within Git, integrating changes from different lines of development.

a. The git add command is used to stage changes for the next commit.
b. The git add command can be used to stage both new files and modifications to existing files
c. The git commit command is used to permanently save changes to the local repository.
d. The git commit command requires a commit message to be provided.
e. The git commit command can be used to merge branches in Git.

Practical assessment

As the project's owner, Sarah wants to tweak a few features and has assigned her coworkers assignments to do so. She begins by creating a new Git repository and a branch called "feature_branch." By adding a new file with the name new_file.txt, Alex makes modifications to the project files. Emily verifies that the repository is functioning properly, adds new_file.txt to the staging area, and then commits the modifications with the message "Add new_file.txt." David makes a switch to the main branch, merges the updates from "feature_branch," and clears up any conflicts. Finally, Sarah, the repository's owner, removes the "feature_branch" following a successful merge. They handle the project's branches, use the staging area properly, and commit file changes to the local Git repository.

References

LinuxHint. (n.d.). Git: List of new, modified, and deleted files. Retrieved from https://linuxhint.com/git-list-of-new-modified-deleted-files/

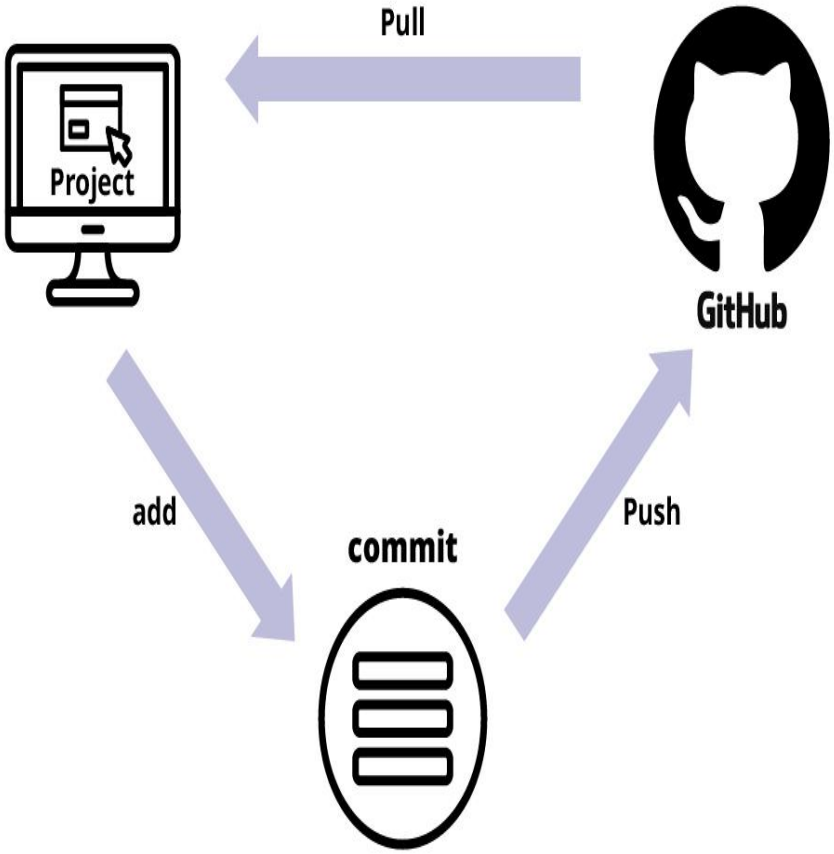
Noble Desktop. (n.d.). How to stage and commit files in Git. Retrieved from https://www.nobledesktop.com/learn/git/stage-commit-files

JavaTpoint. (n.d.). Git Reset. Retrieved from https://www.javatpoint.com/git-reset

JavaTpoint. (n.d.). Git rm. Retrieved from https://www.javatpoint.com/git-rm

CareerFoundry. (n.d.). Git commit command explained. Retrieved from https://careerfoundry.com/en/blog/web-development/git-commit-command/

TutorialsPoint. (n.d.). Git Managing Branches. Retrieved from https://www.tutorialspoint.com/git/git_managing_branches.htm


Pull
Project
GitHub
add
commit
Push

Indicative contents
3.1 Fetch file from GitHub repository **3.2 Push files to remote branch** **3.3 Merge branches on remote repository**

Key Competencies for Learning Outcome 3: Ship codes.

Knowledge	Skills	Attitudes
• Description of pull and fetch commands operations • Description of pull request • Explanation of Tags used on git push command and operations • Description of operation on git rebase command • Description of operation on git merge.	• Fetching file from GitHub repository • Pulling files to GitHub repository • Pushing files to remote branch • Creating pull request • Merging branches on remote repository	• Being Practical oriented • Have Communication Skills • Have critical thinking • Have Team work spirit • Being Problem solver

Duration: 20hrs

Learning outcome 2 objectives:

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

1. Describe correctly pull, fetch, push and git rebase commands based on git project
2. Fetch correctly files in different operations based on git instructions
3. Push properly files to remote branch based on committed files
4. Merge effectively branches on remote repository based on pull request created.

Resources

Equipment	Tools	Materials
• Computer	• Git • GitHub • Text editor (vs code) • Terminal (CMD, Gitbash).	• Internet • Electricity

Indicative content 3.1: Fetch file from GitHub repository

Duration: 7 hrs

Theoretical Activity 3.1.1: Introduction of Fetching file from GitHub

Tasks:

1: In small groups, you are requested to answer the following questions related to the file from GitHub:

i. i.What do you understand about the following term:
   a. Fetch
   b. Pull
ii. Can you explain the operations involved in the `git fetch` command in Git?
iii. Can you explain the operations involved in the `git pull` command in Git?

2: Participate in group formulation

3: Present your findings to your classmates and trainer

4: For more clarification, read the key readings 3.1.1. In addition, ask questions where necessary.

Key readings 3.1.1 : Description of Fetching file from GitHub

1. Git Fetch

Git "fetch" Downloads commits, objects and refs from another repository. It fetches branches and tags from one or more repositories. It holds repositories along with the objects that are necessary to complete their histories to keep updated remote-tracking branches.

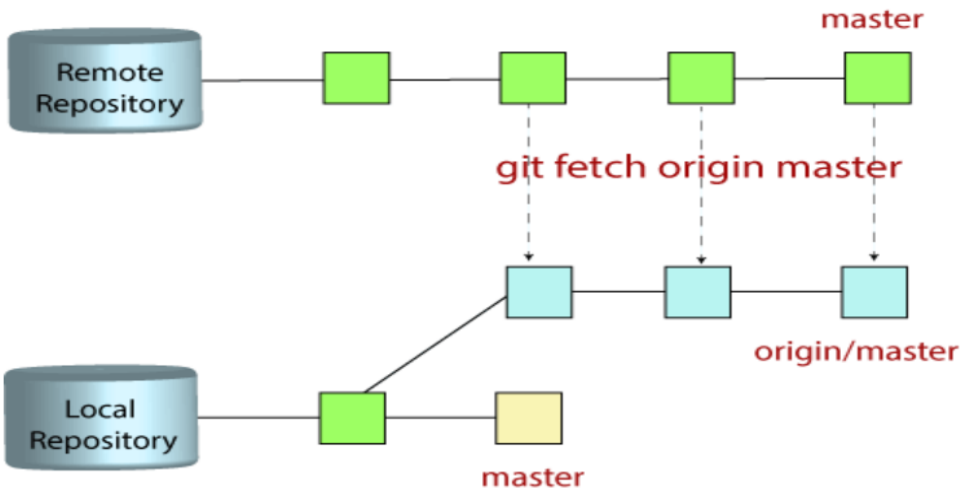


Figure 12: Git fetch branches

How git fetch works with remote branches

To better understand how git fetch works let us discuss how Git organizes and stores commits. Behind the scenes, in the repository's ./.git/objects directory, Git stores all commits, local and remote. Git keeps remote and local branch commits distinctly separate through the use of branch refs. The refs for local branches are stored in the ./.git/refs/heads/. Executing the git branch command will output a list of the local branch refs. The following is an example of git branch output with some demo branch names.

git branch main feature1 debug2

Examining the contents of the /.git/refs/heads/ directory would reveal similar output.

ls ./.git/refs/heads/ main feature1 debug2

Remote branches are just like local branches, except they map to commits from somebody else's repository. Remote branches are prefixed by the remote they belong to so that you don't mix them up with local branches. Like local branches, Git also has refs for remote branches. Remote branch refs live in the ./.git/refs/remotes/ directory. The next example code snippet shows the branches you might see after fetching a remote repo conveniently named remote-repo:

```
git branch -r
# origin/main
# origin/feature1
# origin/debug2
# remote-repo/main
# remote-repo/other-feature
```

This output displays the local branches we had previously examined but now displays them prefixed with origin/. Additionally, we now see the remote branches prefixed with remote-repo. You can check out a remote branch just like a local one, but this puts you in a detached HEAD state (just like checking out an old commit). You can think of them as read-only branches. To view your remote branches, simply pass the -r flag to the git branch command.

You can inspect remote branches with the usual git checkout and git log commands. If you approve the changes a remote branch contains, you can merge it into a local branch with a normal git merge. So, unlike SVN, synchronizing your local repository with a remote repository is actually a two-step process: fetch, then merge. The git pull command is a convenient shortcut for this process.

The "git fetch"command

The "**git fetch**" **command** is used to pull the updates from remote-tracking branches. Additionally, we can get the updates that have been pushed to our remote branches to our local machines. As we know, a branch is a variation of our repositories main code, so the remote-tracking branches are branches that have been set up to pull and push from remote repository.

1.1 Git fetch commands and options

git fetch <remote>

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

git fetch <remote> <branch>

Same as the above command, but only fetch the specified branch.

git fetch --all

A power move which fetches all registered remotes and their branches:

git fetch --dry-run

The --dry-run option will perform a demo run of the command. It will output examples of actions it will take during the fetch but not apply them.

1.2 Git Pull

The term pull is used to receive data from GitHub. It fetches and merges changes from the remote server to your working directory.

The **git pull command** is used to pull a repository.

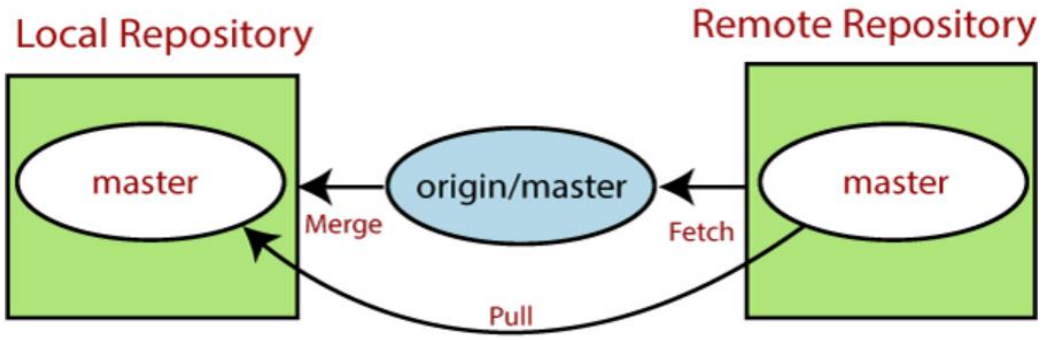


Figure13: Git pull

The **git pull** command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration work flows. The **git pull** command is actually a combination of two other commands, git fetch followed by git merge. In the first stage of operation git pull will execute a git fetch scoped to the local branch that HEAD is pointed at. Once the content is downloaded, git pull will enter a merge workflow. A new merge commit will be-created and HEAD updated to point at the new commit.

The git pull command first runs git fetch which downloads content from the specified remote repository. Then a git merge is executed to merge the remote content refs and heads into a new local merge commit. To better demonstrate the pull and merging process let us consider the following example. Assume we have a repository with a main branch and a remote origin.

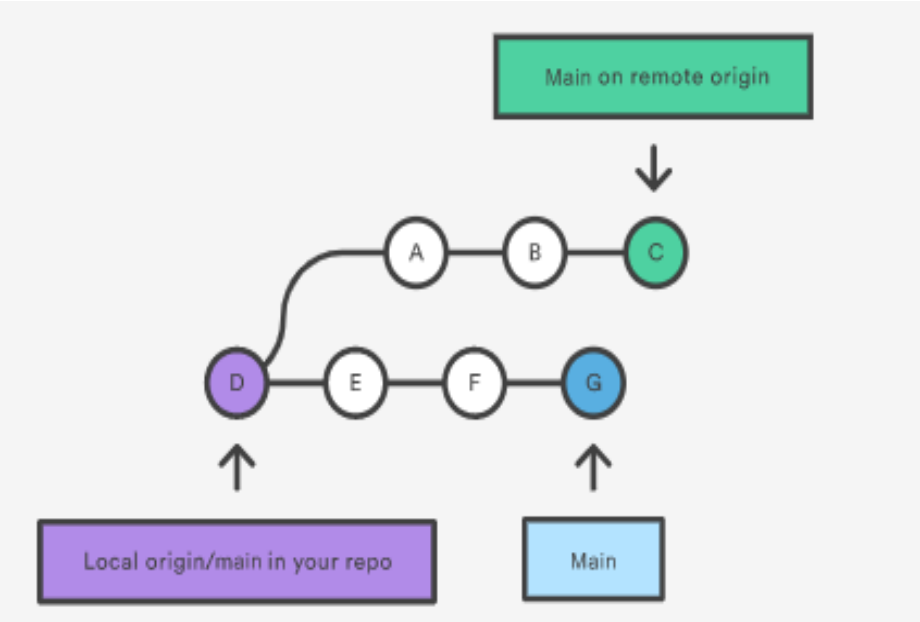


Figure 14: Local main branch and remote origin branch

In this scenario, git pull will download all the changes from the point where the local and main diverged. In this example, that point is E. git pull will fetch the diverged remote commits which are A-B-C. The pull process will then create a new local merge commit containing the content of the new diverged remote commits.

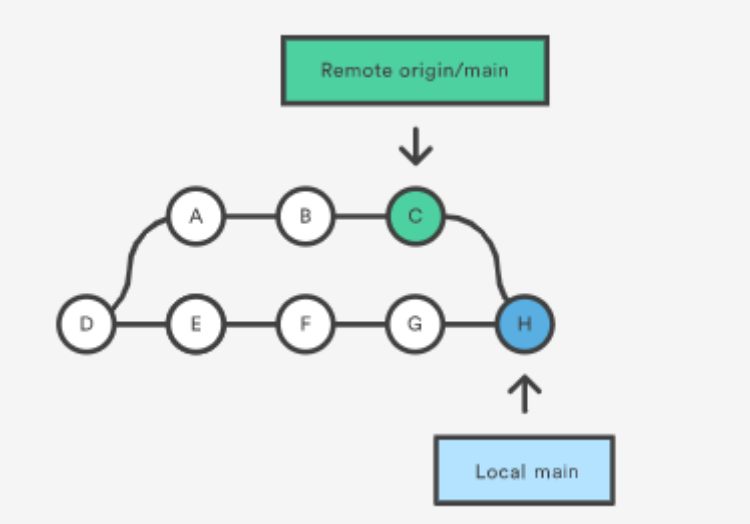


Figure 15: New local merge after pull

In the above diagram, we can see the new commit H. This commit is a new merge commit that contains the contents of remote A-B-C commits and has a combined log message. This example is one of a few git pull merging strategies. A --rebase option can be passed to git pull to use a rebase merging strategy instead of a merge commit. The next example will demonstrate how a rebase pull works. Assume that we are at a starting point of our first diagram, and we have executed git pull --rebase.

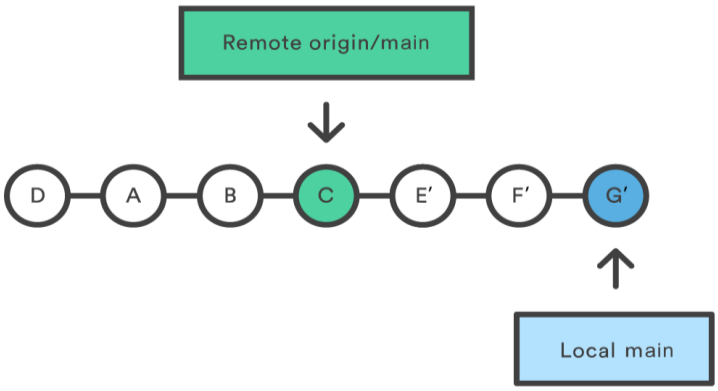


Figure16: Git pull --rebase

In this diagram, we can now see that a rebase pull does not create the new H commit. Instead, the rebase has copied the remote commits A--B--C and rewritten the local commits E--F--G to appear after them in the local origin/main commit history.

Common Options

git pull <remote>

Fetch the specified remote's copy of the current branch and immediately merge it into the local copy. This is the same as git fetch < remote > followed by git merge origin/ < current-branch > .

git pull --no-commit <remote>

Similar to the default invocation, fetches the remote content but does not create a new merge commit.

git pull --rebase <remote>

Same as the previous pull Instead of using git merge to integrate the remote branch with the local one, use git rebase.

git pull --verbose

Gives verbose output during a pull which displays the content being downloaded.

1.2 Operations on git fetch command

The git fetch command is a powerful tool in Git that allows you to retrieve changes from a remote repository and update your local repository accordingly. Here are some applications of the various operations you mentioned:

1. **Fetch the remote repository:** When you run **git fetch**, Git connects to the remote repository specified in the configuration and retrieves all the new commits, branches, and tags from that remote repository. It downloads the latest changes to your local repository, but it does not integrate them with your current working branch.
2. **Fetch the specific branch:** By providing the branch name as an argument to **git fetch**, you can fetch the latest changes only for that specific branch from the remote repository. For example, if you want to fetch the latest changes for the branch named "development," you would run **git fetch origin development**. This operation updates your local repository with the latest commits and updates related to the specified branch.
3. **Fetch all branches simultaneously**: By default, when you run **git fetch** without specifying a specific branch, it fetches changes for all branches from the remote repository. This operation retrieves the latest commits and updates for all branches, updating your local repository accordingly.
4. **Synchronize the local repository: git fetch** is often used to synchronize your local repository with the latest changes from the remote repository. After fetching the latest changes, you can compare them with your local branches to see the differences and decide how to integrate them. It allows you to review the changes, merge them manually, or switch to a different branch to work on the updated version.

git fetch is a useful command to keep your local repository up-to-date with the latest changes from the remote repository. It fetches new commits, branches, and tags and allows you to review and integrate them at your convenience without automatically merging them into your current working branch.

2. Operations on git pull

The git pull command is used to fetch and merge changes from a remote repository into your local branch. Here are the applications of the different operations you mentioned:

1. **Default git pull:** When you run git pull without any additional arguments, it performs the default behavior of fetching the latest changes from the remote repository and merging them into your current branch. It connects to the remote repository specified in the configuration (usually the "origin" remote) and downloads any new commits, branches, and tags. Then, it automatically merges those changes into your current branch using the default merge strategy. If there are any conflicts, you will need to resolve them manually.

2. **git pull remote branch:** If you want to pull changes from a specific branch in the remote repository, you can specify the branch name as an argument to git pull. For example, running git pull origin feature-branch fetches the latest changes from the "feature-branch" in the remote repository specified by the "origin" remote. It then merges those changes into your current branch.
3. **git force pull:** The git force pull command is not a built-in Git command. However, you can achieve a similar effect by combining the git fetch and git reset commands. Running git fetch retrieves the latest changes from the remote repository, and then git reset --hard origin/master resets your current branch to match the "master" branch of the remote repository, discarding any local changes. Using this combination effectively forces a pull-like behavior, overwriting your local branch with the remote branch. Be cautious when using git force pull as it can cause irreversible data loss.
4. **git pull origin master:** This git pull command explicitly specifies the remote repository ("origin") and the branch to pull from ("master"). Running git pull origin master fetches the latest changes from the "master" branch of the remote repository and merges them into your current branch. It's a specific way to update your current branch with the latest changes from the "master" branch of the remote repository.

git pull is a versatile command that allows you to fetch and merge changes from a remote repository. By using different arguments, such as specifying the remote branch or force pulling, you can tailor the behavior of git pull to suit your needs.

2.1 Differences between git fetch and git pull

To understand the differences between fetch and pull, let's know the similarities between both of these commands. Both commands are used to download the data from a remote repository. But both of these commands work differently. Like when you do a git pull, it gets all the changes from the remote or central repository and makes it available to your corresponding branch in your local repository. When you do a git fetch, it fetches all the changes from the remote repository and stores it in a separate branch in your local repository. You can reflect those changes in your corresponding branches by merging.
So basically.

1. **git pull = git fetch + git merge**

Git Fetch vs. Pull

Some of the key differences between both of these commands are as follows:

git fetch	**git pull**

Fetch downloads only new data from a remote repository.	Pull is used to update your current HEAD branch with the latest changes from the remote server.
Fetch is used to get a new view of all the things that happened in a remote repository.	Pull downloads new data and directly integrates it into your current working copy files.
Fetch never manipulates or spoils data.	Pull downloads the data and integrates it with the current working file.
It protects your code from merge conflict.	In git pull, there are more chances to create the **merge conflict**.
It is better to use git fetch command with git merge command on a pulled repository.	It is not an excellent choice to use git pull if you already pulled any repository.

Practical Activity 3.1.2: Fetching file from GitHub repository

Task:

1: Read key reading 3.1.2 and ask clarification where necessary
2: Referring to the previous theoretical activities (3.1.1) you are requested to go to the computer lab to fetch file from GitHub repository. This task should be done individually.
3: Apply safety precautions
4: Present out the steps to fetch file from GitHub repository.
5: Referring to the steps provided on task 3, fetch file from GitHub repository.
6: Present your work to the trainer and whole class

Key readings 3.1.2: Fetching file from GitHub repository

1. Fetch a file from a GitHub repository

To fetch a file from a GitHub repository, you typically need to clone the repository to your local machine. Here are the steps:

1. Find the Repository: Go to the GitHub website and navigate to the repository containing the file you want to fetch.

2. Get the Repository URL: Click on the "Code" button, then copy the URL provided. It should look something like https://github.com/username/repository.git.

3. Open Terminal/Command Prompt: Open your terminal or command prompt on your local machine.

4. Navigate to the Desired Directory: Use the cd command to navigate to the directory where you want to clone the repository. For example:

```
cd path/to/directory
```

5.Clone the Repository: Use the git clone command followed by the repository URL to clone the repository to your local machine:

```
git clone https://github.com/username/repository.git
```

6.Navigate into the Cloned Repository: Move into the cloned repository directory using the cd command:

```
cd repository
```

7. Fetch the File: If you know the path to the file you want to fetch, you can navigate to it using a file explorer or use commands like ls (Unix-based systems) or dir (Windows) to list the contents of the directory and locate the file. If the file is located within a subdirectory of the repository, you can navigate to that subdirectory using the cd command.

Copy or Open the File: Once you have located the file, you can copy it to another location on your local machine or open it using a text editor or IDE to view its contents.

1.1 To fetch the remote repository:

We can fetch the complete repository with the help of fetch command from a repository URL like a pull command does. See the below output:

Syntax:

1. **$ git fetch< repository Url>**

example

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch https://github.com/ImDwivedi1/Git-Example.git
warning: no common commits
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/ImDwivedi1/Git-Example
 * branch            HEAD       -> FETCH_HEAD

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$
```

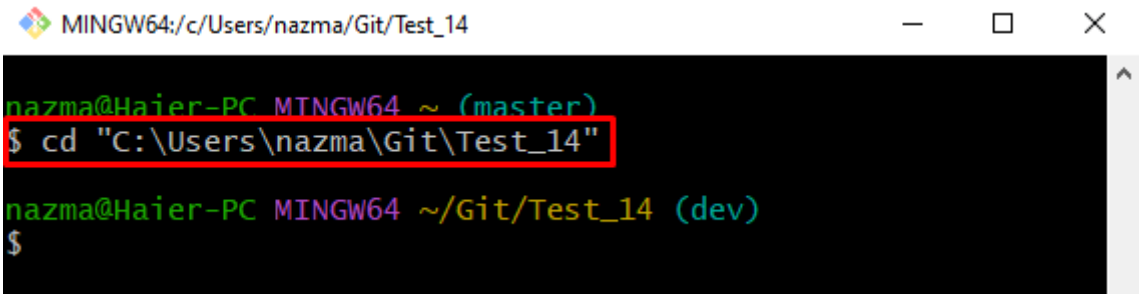
In the above output, the complete repository has fetched from a remote URL.

1.2 To fetch a specific branch:

Step1: **Navigate to Git Repository**

Go to the desired local repository by executing the “cd” command:

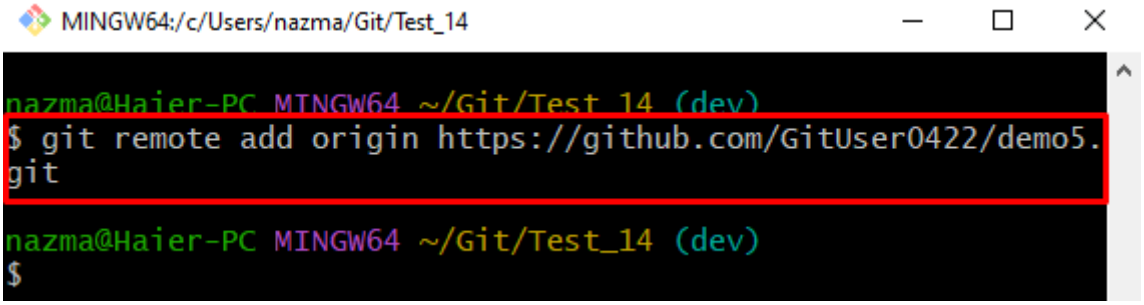
$ cd "C:\Users\nazma\Git\Test_14"



Step2:**Add Remote URLs**

Next, use the “git remote add” command along with the remote name and remote repository URL for tracking changes:

$ git remote add origin https://github.com/GitUser0422/demo5.git

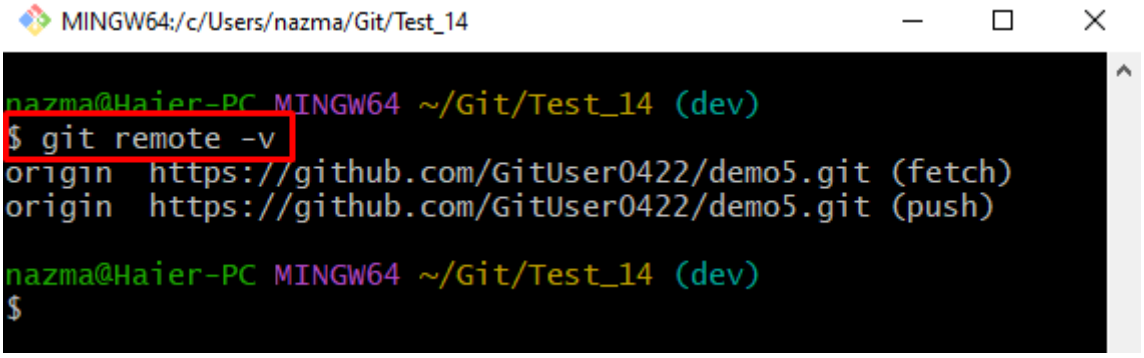


Step 3: Verify Remote URLs List

Now, check the newly added remote URL in Git by running the following command:

$ git remote -v

It can be seen that the remote URL has been added successfully:

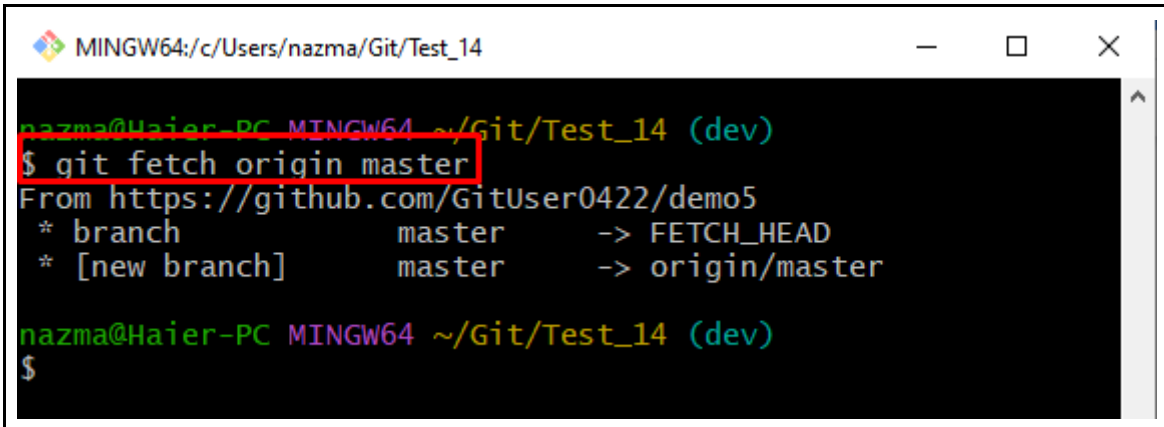


Step 4: **Fetch Particular Remote Branch**

Finally, execute the “git fetch” command with the remote name and the desired remote branch name:

$ git fetch origin master

Here, we have specified the remote branch name as “**master**”:

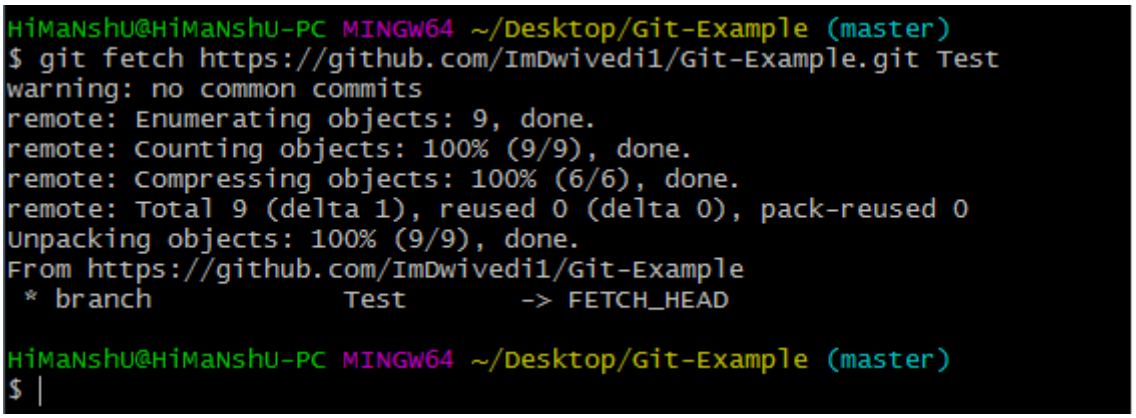


We can also fetch a specific branch from a repository which is not master. It will only access the element from a specific branch.

Syntax:

1. $ git fetch <branch URL><branch name>

Example



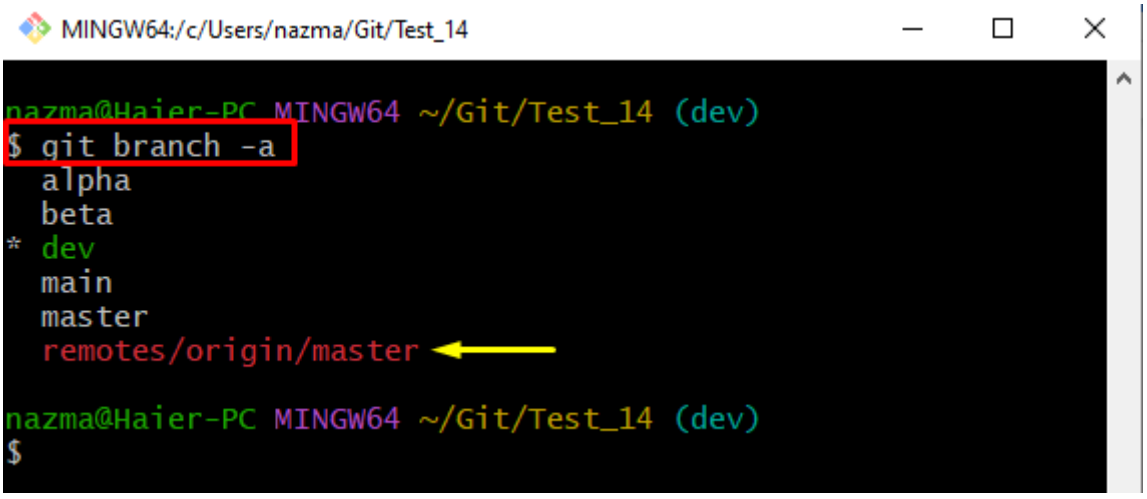
In the given output, the specific branch **test** has fetched from a remote URL.

Step 5: **Verify Fetch Remote Branch**

Lastly, run the “git branch” command along with the “-a” flag to list all branches including the local and remote:

$ git branch -a

As you can see the particular remote branch has been fetched successfully:



That was all about fetching the particular remote Git repository branch.

To fetch all the branches simultaneously:

The git fetch command allows to fetch all branches simultaneously from a remote repository. See the below example:

Syntax:

$ git fetch -all

Example:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch --all
Fetching origin
From https://github.com/ImDwivedi1/Git-Example
 * [new branch]      master     -> origin/master
 * [new branch]      Test       -> origin/Test

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$
```

In the above output, all the branches have fetched from the repository Git-Example.

To synchronize the local repository:

Suppose, your team member has added some new features to your remote repository. So, to add these updates to your local repository, use the git fetch command. It is used as follows.

Syntax:

$ git fetch origin

Example:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch origin

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch origin
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ImDwivedi1/Git-Example
 * [new branch]      test2      -> origin/test2

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$
```

In the above output, new features of the remote repository have updated to my local system. In this output, the branch **test2** and its objects are added to the local repository.

The git fetch can fetch from either a single named repository or URL or from several repositories at once. It can be considered as the safe version of the git pull commands.

The git fetch downloads the remote content but not update your local repo's working state.

When no remote server is specified, by default, it will fetch the origin remote.

1.3 Differences between git fetch and git pull

To understand the differences between fetch and pull, let's know the similarities between both commands. Both commands are used to download the data from a remote repository. But both commands work differently. Like when you do a git pull, it gets all the changes from the remote or central repository and makes it available to your corresponding branch in your local repository. When you do a git fetch, it fetches all the changes from the remote repository and stores it in a separate branch in your local repository. You can reflect those changes in your corresponding branches by merging.

So basically,

git pull = git fetch + git merge

Pulling changes from a remote repository in Git involves several steps, depending on the specific operation you're performing. Here are the steps for various scenarios:

1.Pulling Changes from the Default Branch:

- Navigate to your local repository directory using the terminal or command prompt.
- Use the following command to pull changes from the default branch (usually "main" or "master") of the remote repository:

```
git pull
```

2.Pulling Changes from a Specific Branch:

- If you want to pull changes from a specific branch of the remote repository, specify both the remote and branch names in the pull command:

```
git pull <remote-name> <branch-name>
```

Replace <remote-name> with the name of the remote repository (often "origin") and <branch-name> with the name of the branch you want to pull changes from.

Pulling Changes Without Merging:

- If you want to fetch changes from the remote repository without merging them into your local branch immediately, you can use the fetch command followed by the checkout command to inspect the changes before merging:

```
git fetch
git checkout <branch-name>
```

This will fetch changes from the remote repository but keep your local branch unchanged until you explicitly merge the changes.

3.Pulling Changes with Rebase:

- To pull changes from the remote repository and rebase your local commits on top of the remote commits, use the rebase option with the pull command:

```
git pull --rebase
```

4.Pulling Changes with Force:

- In some cases, you may need to forcefully overwrite your local changes with the changes from the remote repository. Use the force option with the pull command:

```
git pull --force
```

Be cautious when using this option, as it can result in the loss of local changes.

There is another way to pull the repository. We can pull the repository by using the git pull command. The syntax is given below:

1. $ git pull **<options><remote>/<branchname>**
2. **$ git pull origin master**

Resolving merge conflicts in Git involves several steps. Here's a guide on how to handle merge conflicts:

1. **Identify Conflicts:** After attempting to merge changes from a remote branch or another local branch, Git may encounter conflicts if the changes overlap or conflict with each other. Git will notify you of these conflicts and mark the conflicted files in your working directory.
2. **View Conflicts:** Open the conflicted files in a text editor. Git will insert conflict markers (<<<<<<<, =======, and >>>>>>>) to indicate the conflicting sections. These markers delineate the conflicting changes from both branches.
3. **Analyze Conflicts:** Review the conflicting sections in the file. Identify the changes from both branches and decide how to resolve the conflicts. You may choose to keep one version of the changes, combine them, or make entirely new changes.
4. **Resolve Conflicts:** Edit the conflicted file to resolve the conflicts manually. Remove the conflict markers and make the necessary adjustments to reconcile the conflicting changes. Ensure that the final version of the file integrates the desired changes from both branches.
5. **Save Changes:** After resolving the conflicts, save the changes to the file in your text editor.
6. **Mark as Resolved:** Once you have resolved the conflicts in all conflicted files, stage the changes by marking the conflicts as resolved:

```
git add <conflicted-file>
```

7. **Commit Changes:** After staging the resolved changes, commit the changes to complete the merge process:

```
git commit
```

This will open a commit message editor where you can provide a description of the merge and the resolutions you made to the conflicts.

8. **Continue with Merge:** If you were in the middle of a merge operation when conflicts occurred, you can continue the merge process after resolving conflicts by running:

```
git merge --continue
```

This command will finalize the merge commit with the resolved conflicts.

9.Verify Resolution: After committing the changes, verify that the conflicts have been resolved successfully by reviewing the merged files and testing the functionality of the merged code.

10.Push Changes: If you were merging changes from a remote branch, push the merged changes to the remote repository:

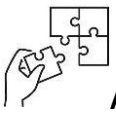
```
git push
```

This will update the remote repository with the resolved merge

Points to Remember

To fetch a file from a GitHub repository, you typically need to clone the repository to your local machine. Here are the steps:

1.Find the Repository
2.Get the Repository URL
3.Open Terminal/Command Prompt
4.Clone the Repository
5.Navigate into the Cloned Repository
6. Fetch the File



Application of learning 3.1.

The owner of the GBY project, which uses Git for version control, needs to improve it by adding new features and content to make management, collaboration, and maintenance easier. Different tasks are given to a team of developers to complete. As with getting the most recent updates from a cloud-based document collaboration platform, Alice is responsible with obtaining the remote repository. Like subscribing to channels or subjects on a social media platform, Bob oversees fetching a particular branch. Claire's job is to synchronize the local repository across all branches at once, which is like syncing a music streaming service across several devices. David executes the standard git pull action, which is comparable to an email client automatically downloading new emails. Emma "pulls" a particular remote branch, which is comparable to downloading specific files or directories from cloud storage. In a manner like overwriting local files on a file synchronization service, Frank manages the git pull --force procedure. Last but not least, Grace updates a piece of software to the most recent stable version by pulling changes from the "master" branch.

Indicative content 3.2: Push files to remote branch

Duration: 6 hrs

Theoretical Activity 3.2.1: Introduction to files pushing to remote

Tasks:

1: In small groups, you are requested to answer the following questions related to the files pushing to remote branch:

1) What do you understand about the term push?
2) Can you provide a description of the tags used in the `git push` command?

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 3.2.1. In addition, ask questions where necessary.

Key readings 3.2.1: Description of Git push operations

1. Definition of Git Push

Using the git push command, you can upload your files available on your local machine to the remote repository. After git pushes the changes to the remote repository other developers can access the changes and they can contribute their changes by git pulling. Before pushing it to the remote repository you need to do a git commit to your local machine.

1.1 Git Push Command

Git push allows us to transfer files from the local repository to remote repository hosting services like GitHub, GitLab, etc. Other developers who want to work on the files can access them after being uploaded to a remote repository.

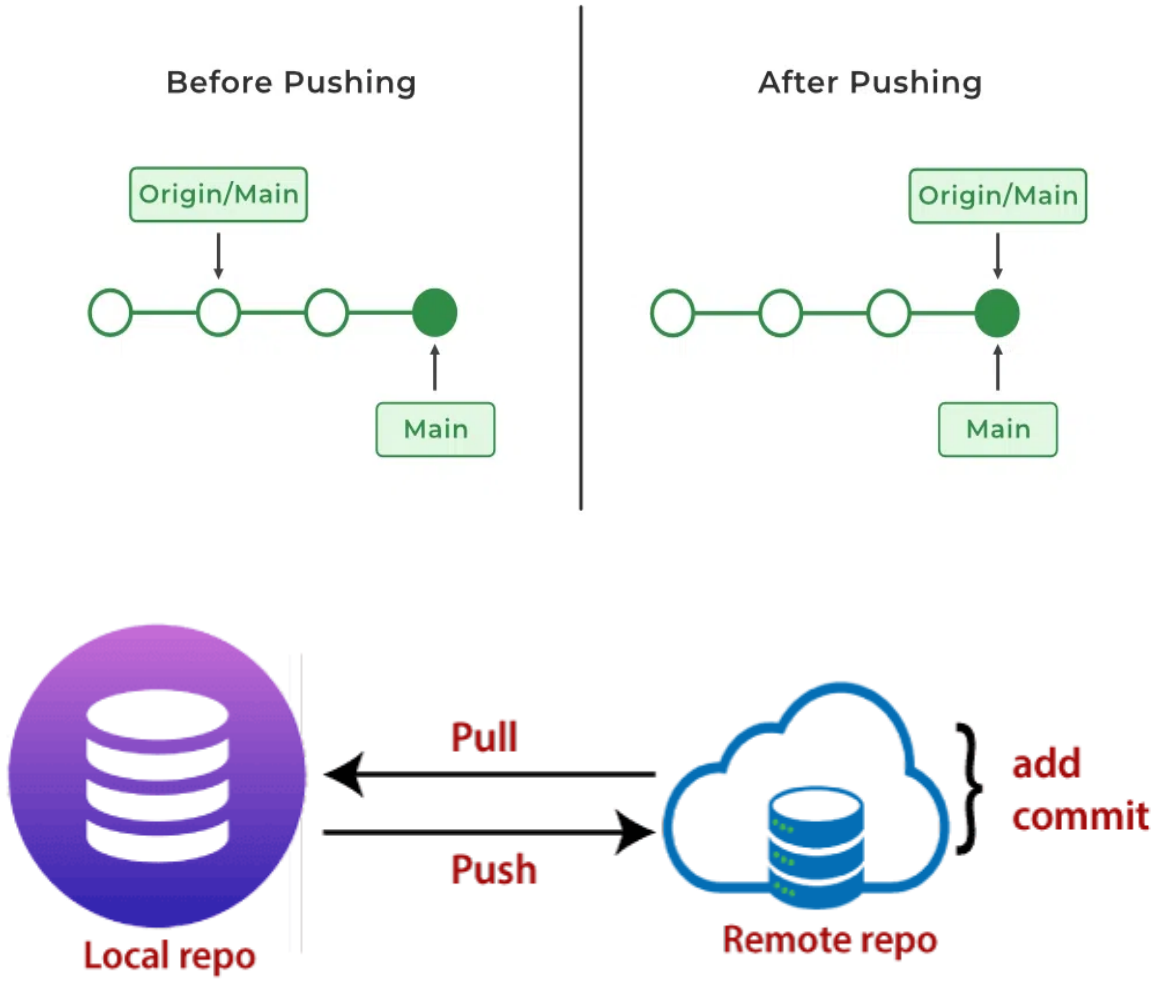


Figure 17: Push diagram

In the above diagram, we can observe If our local main branch is way back when compared to the central main repository after that git push origin main will publish the changes. git push is essential as same as the git merge.

git push <remote> <branch>

1. The <remote> option refers to the remote repository to which you want to push your files it will refer to its alias name where the name is mapped with the remote repository URL
2. The <branch> option represents the branch of the GitHub repository which you want to push

The push term refers to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repository. Pushing is capable of overwriting changes; caution should be taken when pushing.

Moreover, we can say the push updates the remote refs with local refs. Every time you push into the repository, it is updated with some interesting changes that you made. If we do not specify the location of a repository, then it will push to default location at **origin master**.
The "git push" command is used to push into the repository. The push command can be considered as a tool to transfer commits between local and remote repositories. The basic syntax is given below:

1. $ git push <option> [<Remote URL><branch name><refspec>...]

1.2 Description of Git Push Tags

<repository>: The repository is the destination of a push operation. It can be either a URL or the name of a remote repository.
<refspec>: It specifies the destination ref to update source object.
--all: The word "all" stands for all branches. It pushes all branches.
--prune: It removes the remote branches that do not have a local counterpart. Means, if you have a remote branch say demo, if this branch does not exist locally, then it will be removed.
--mirror: It is used to mirror the repository to the remote. Updated or Newly created local refs will be pushed to the remote end. It can be force updated on the remote end. The deleted refs will be removed from the remote end.
--dry-run: Dry run tests the commands. It does all this except originally update the repository.
--tags: It pushes all local tags.
--delete: It deletes the specified branch.
-u: It creates an upstream tracking connection. It is very useful if you are going to push the branch for the first time.

1.3 Explanation of operations on git push

push on origin master

In Git, the command git push origin master is used to push your local branch named "master" to the remote repository named "origin." Here's a breakdown of what each part of the command means:
git push: This is the command used to upload your local commits to a remote repository.
origin: This refers to the name of the remote repository where you want to push your changes. By convention, "origin" is the default name given to the main remote repository when you clone it.
master: This is the name of the local branch that you want to push. In this case, "master" is the branch you are pushing.
Git push origin master is a special command-line utility that specifies the remote branch and directory. When you have multiple branches and directory, then this command assists you in determining your main branch and repository.

Generally, the term **origin stands** for the remote repository, and master is considered as the main branch. So, the entire statement "**git push origin master**" pushed the local content on the master branch of the remote location.

Syntax:

1. $ git push origin master

Git push force

The **git push --force** command is used to forcefully push your local branch and overwrite the corresponding branch on the remote repository, even if it results in losing commits or overwriting someone else's work. It allows you to make significant changes to the history of the remote branch.

The git force push allows you to push local repository to remote without dealing with conflicts. It is used as follows:

1. **$ git push <remote><branch> -f**

Or

1. **$ git push <remote><branch> -force**

The -f version is used as an abbreviation of force. The remote can be any remote location like GitHub, Subversion, or any other git service, and the branch is a particular branch name. For example, we can use git push origin master -f.

We can also omit the branch in this command. The command will be executed as:

1. **$git push <remote> -f**

We can omit both the remote and branch. When the remote and the branch both are omitted, the default behavior is determined by **push.default** setting of git config. The command will be executed as:

1. **$ git push -f**

How to Safe Force Push Repository:

There are several consequences of force pushing a repository like it may replace the work you want to keep. Force pushing with a lease option is capable of making fail to push if there are new commits on the remote that you didn't expect. If we say in terms of git, then we can say it will make it fail if remote contains untracked commit. It can be executed as:

1. **$git push <remote><branch> --force-with-lease**

git push verbose or Git push -v/--verbose

The **git push --verbose** command is used to display detailed information about the push operation. It provides additional output, including progress updates, error messages, and other relevant details during the push process.

The -v stands for verbosely. It runs command verbosely. It pushed the repository and gave a detailed explanation about objects. Suppose we have added a **newfile2.txt** in our local

repository and commit it. Now, when we push it on remote, it will give more description than the default git push. Syntax of push verbosely is given below:

Syntax:

1. **$ git push -v**

Or

1. **$ git push --verbose**

delete a remote branch

To delete a remote branch in Git, you can use the git push command with the --delete or -d option, followed by the name of the remote branch you want to delete.

We can delete a remote branch using git push. It allows removing a remote branch from the command line. To delete a remote branch, perform below command:

Syntax:

1. $ git push origin -delete edited

Practical Activity 3.2.2: Pushing files to remote branch.

Task:

1: Read key reading 3.2.2 and ask clarification where necessary

2: Referring to three (3) previous theoretical activities (3.2.1) you are requested to go to the computer lab to push files to remote branch. This task should be done individually.

3: Apply safety precautions.

4: Present the steps of pushing files to the remote branch.

5: Referring to the steps provided on task 3, push files to the remote branch.

6: Present your work to the trainer and whole class

Key readings 3.2.1: Applying Git push operations

1. Pushing files to a remote branch

Pushing files to a remote branch in Git involves several steps. Here's a detailed guide**:**

1. **Add and Commit Changes:** First, ensure that you have made the necessary changes to your files. Use the following commands to stage your changes and commit them:

```
git add <file1> <file2> ...    # Stage the changes
git commit -m "Your commit message"    # Commit the changes
```

2. **Check Remote Repository Status:** Before pushing changes, it's a good practice to verify the status of your local repository compared to the remote repository. Use the following command:

```
git remote -v    # Check remote repository URLs
git branch -vv   # Check local and remote branches
```

3. **Fetch and Pull Changes:** Fetch the latest changes from the remote repository and ensure your local branch is up to date. This step helps in avoiding conflicts during the push.

```
git fetch origin    # Fetch latest changes from remote repository
git pull origin <branch_name>    # Pull changes from the remote branch
```

4.**Push Changes to Remote Branch:** After committing your changes and ensuring your local branch is up to date, push your changes to the remote branch:

```
git push origin <local_branch_name>:<remote_branch_name>
```

Replace <local_branch_name> with the name of your local branch and <remote_branch_name> with the name of the remote branch you want to push to. If the branch does not exist on the remote repository, it will be created. If it does exist, the changes will be pushed to that branch.

Alternatively, if you're working on the same branch locally and remotely, you can use the simpler form:

```
git push origin <branch_name>
```

5.**Verify Changes on Remote Repository:** Once the push is successful, verify that your changes are reflected on the remote repository by visiting the repository's web interface or using Git commands to check the remote branch:

```
git ls-remote origin    # List references in a remote repository
git branch -r    # List remote branches
```

1.1 Git Push Origin Master

Git push origin master is a special command-line utility that specifies the remote branch and directory. When you have multiple branches and directory, then this command assists you in determining your main branch and repository.

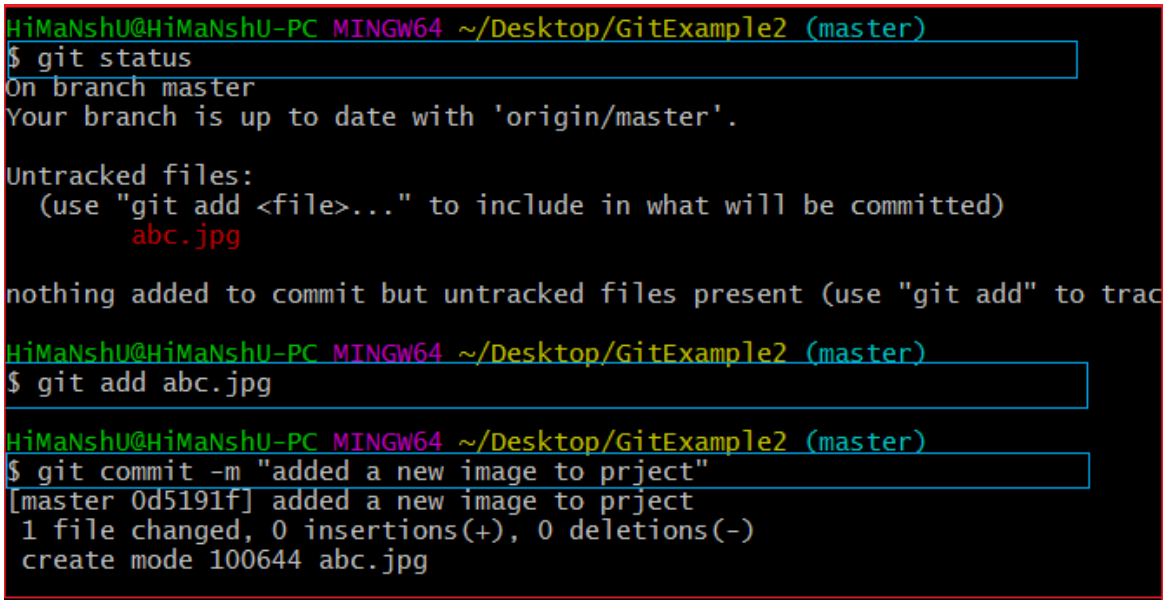
Generally, the term origin stands for the remote repository, and master is considered as the main branch. So, the entire statement "git push origin master" pushed the local content on the master branch of the remote location.

Syntax:

1. **$ git push origin master**

Let's understand this statement with an example.

Let's make a new commit to my existing repository, say **GitExample2**. I have added an image to my local repository named **abc.jpg** and committed the changes. Consider the below image:



In the above output, I have attached a picture to my local repository. The git status command is used to check the status of the repository. The git status command will be performed as follows:

1. **$ git status**

It shows the status of the untracked image **abc.jpg**. Now, add the image and commit the changes as:

1. **$ git add abc.jpg**

2. $git commit -m "added a new image to project."

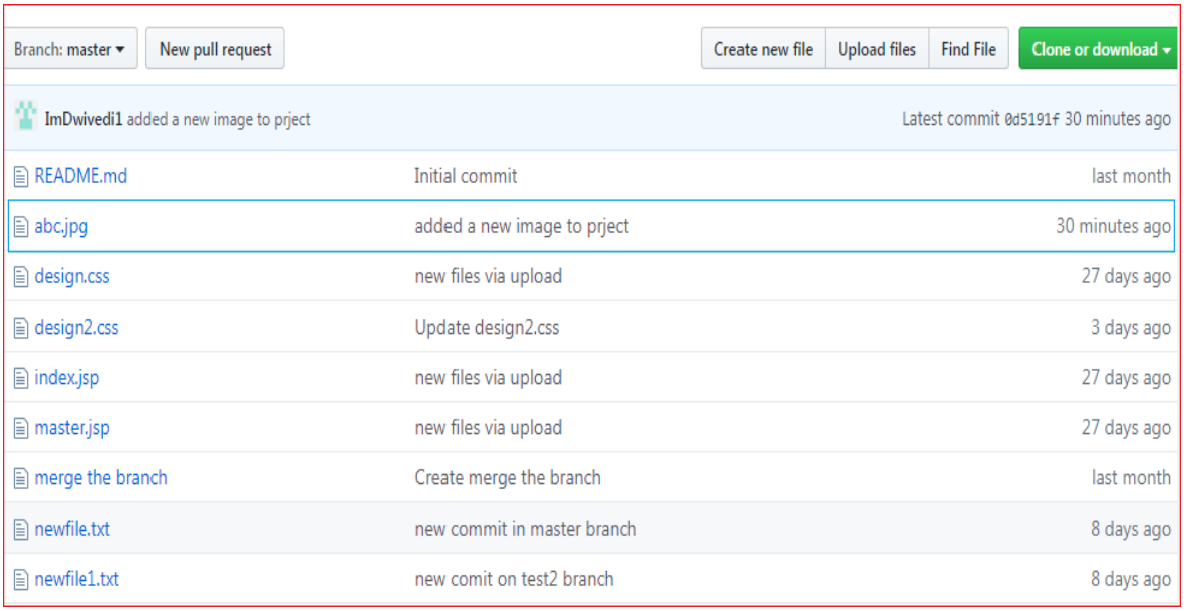
The image is wholly tracked in the local repository. Now, we can push it to origin master as:

1. $ git push origin master

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git push origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 757.29 KiB | 6.95 MiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/ImDwivedi1/GitExample2.git
   828b962..0d5191f  master -> master
```

The file **abc.jpg** is successfully pushed to the origin master. We can track it on the remote location. I have pushed these changes to my GitHub account. I can track it there in my repository. Consider the below image:



In the above output, the pushed file abc.jpg is uploaded on my GitHub account's master branch repository.

1.2 Git Force Push

The git force push allows you to push local repository to remote without dealing with conflicts. It is used as follows:

1. $ git push <remote><branch> -f

Or

1. **$ git push <remote><branch> -force**

The -f version is used as an abbreviation of force. The remote can be any remote location like GitHub, Subversion, or any other git service, and the branch is a particular branch name. For example, we can use git push origin master -f.

We can also omit the branch in this command. The command will be executed as:

1. **$git push <remote> -f**

We can omit both the remote and branch. When the remote and the branch both are omitted, the default behavior is determined by **push.default** setting of git config. The command will be executed as:

1. **$ git push -f**

How to Safe Force Push Repository:

There are several consequences of force pushing a repository like it may replace the work you want to keep. Force pushing with a lease option is capable of making fail to push if there are new commits on the remote that you didn't expect. If we say in terms of git, then we can say it will make it fail if remote contains untracked commit. It can be executed as:

1. **$git push <remote><branch> --force-with-lease**

Git push -v/--verbose

The -v stands for verbosely. It runs command verbosely. It pushed the repository and gave a detailed explanation about objects. Suppose we have added a **newfile2.txt** in our local repository and commit it. Now, when we push it on remote, it will give more description than the default git push. Syntax of push verbosely is given below:

Syntax:

1. **$ git push -v** Or $ **git push --verbose**

Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git push -v
Pushing to https://github.com/ImDwivedi1/GitExample2.git
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 289 bytes | 48.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
POST git-receive-pack (452 bytes)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/ImDwivedi1/GitExample2.git
   0d5191f..56afce0  master -> master
updating local tracking ref 'refs/remotes/origin/master'
```

If we compare the above output with the default git option, we can see that git verbose gives descriptive output.

2. Delete a Remote Branch

We can delete a remote branch using git push. It allows removing a remote branch from the command line. To delete a remote branch, perform below command:

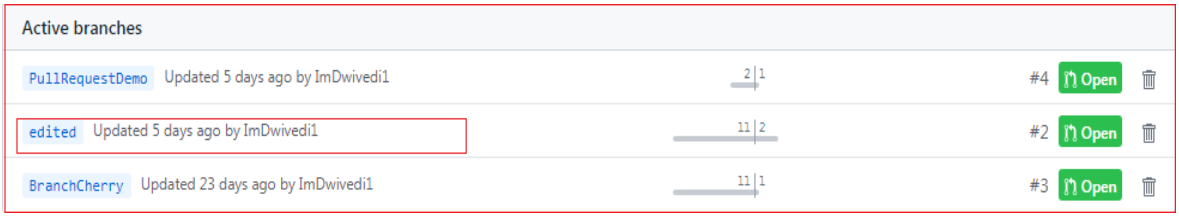
Syntax:

1. **$ git push origin -delete edited**

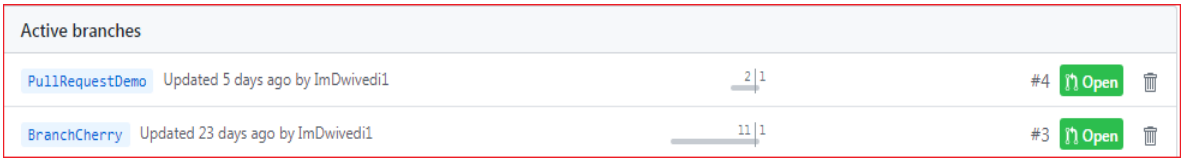
Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git push origin --delete edited
To https://github.com/ImDwivedi1/GitExample2.git
 - [deleted]         edited

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

In the above output, the git push origin command is used with -delete option to delete a remote branch. I have deleted my remote branch **edited** from the repository. Consider the below image:



It is a list of active branches of my remote repository before the operating command.



The above image displays the list of active branches after deleting command. Here, you can see that the branch **edited** has removed from the repository.

Points to Remember

- Tags in Git are used to mark specific points in history, such as releases or important c ommits.
- There are the tags used with the git push command like :<repository>,<refspec>,--all, --prune,--mirror,--dry-run,--tags.
- **To push files to a remote branch in Git, you typically follow these steps:**
    1. Add and Commit Changes
    2. Check Remote Repository Status
    3. Fetch and Pull Changes
    4. Push Changes to Remote Branch
    5. Verify Changes on Remote Repository

Application of learning 3.2.

As a member of a team project called "TechApp" using Git for version control. You are asked to implement a new feature and merge it into the main codebase. However, you face challenges along the way. You start by pushing your local changes to the "origin master" branch to make them accessible to your team. Later, you encounter conflicts with recent updates from others, requiring you to forcefully update the remote repository. To monitor the push process, you use the verbose option, which provides detailed information about the operation. Finally, after successfully merging your changes, you delete the remote branch that is no longer needed. Through effective communication and proper use of Git commands, you ensure smooth collaboration and maintain a clean repository.

Indicative content 3.3: Merge branches on remote repository

Duration: 7hrs

Theoretical Activity 3.3.1: Description of merge branches on remote repository

Tasks:

1: In small groups, you are requested to answer the following questions related to the merge branches on remote:

I. Can you discuss the operation on git rebase command, pull requests, operation on git merge.

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 3.3.1. In addition, ask questions where necessary.

Key readings 3.3.1.: Description of merge branches on remote repository

1.Git Rebase

Rebasing is a process to reapply commits on top of another base trip. It is used to apply a sequence of commits from distinct branches into a final commit. It is an alternative of git merge command. It is a linear process of merging.

In Git, the term rebase is referred to as the process of moving or combining a sequence of commits to a new base commit. Rebasing is very beneficial and it visualized the process in the environment of a feature branching workflow.

git rebase is a Git command used to reapply a series of commits from one branch onto another branch. It allows you to rewrite the commit history of the current branch by incorporating changes from another branch, typically the branch specified as the new base.

Here is the basic syntax for the git rebase command:

```
git rebase <base>
```

<base>: The branch or commit onto which you want to reapply the commits from the current branch. This can be a branch name or a commit hash.

When you run git rebase <base>, Git performs the following actions:

Identifies the common ancestor commit between the current branch and the specified <base> branch.

Retrieves the commits unique to the current branch since the common ancestor.

"Replays" each of these commits onto the <base> branch one by one, effectively incorporating the changes into the target branch.

Moves the current branch pointer to the tip of the rebased commits, effectively rewriting the branch history.

In cases where conflicts occur during the rebase process, Git will pause the rebase operation and prompt you to resolve the conflicts manually. After resolving conflicts, you can continue the rebase operation using git rebase --continue. If you encounter issues or decide to abort the rebase, you can use git rebase --abort to revert to the state before the rebase began.

git rebase is a Git command used to reapply a series of commits from one branch onto another branch. It allows you to rewrite the commit history of the current branch by incorporating changes from another branch, typically the branch specified as the new base. Here is the basic syntax for the git rebase command:

git rebase <base>

- <base>: The branch or commit onto which you want to reapply the commits from the current branch. This can be a branch name or a commit hash.

When you run git rebase <base>, Git performs the following actions:

Identifies the common ancestor commit between the current branch and the specified <base> branch.

Retrieves the commits unique to the current branch since the common ancestor.

"Replays" each of these commits onto the <base> branch one by one, effectively incorporating the changes into the target branch.

Moves the current branch pointer to the tip of the rebased commits, effectively rewriting the branch history.

In cases where conflicts occur during the rebase process, Git will pause the rebase operation and prompt you to resolve the conflicts manually. After resolving conflicts, you can continue the rebase operation using git rebase -**continue**. If you encounter issues or decide to abort the rebase, you can use git rebase **-abort** to revert to the state before the rebase began.

It's essential to use git rebase carefully, especially when working with shared branches, as it alters the commit history. Rewriting history can cause confusion and conflicts for collaborators who have based their work on the original branch. Therefore, it's crucial to communicate with team members before performing rebases on shared branches.

It is good to rebase your branch before merging it.

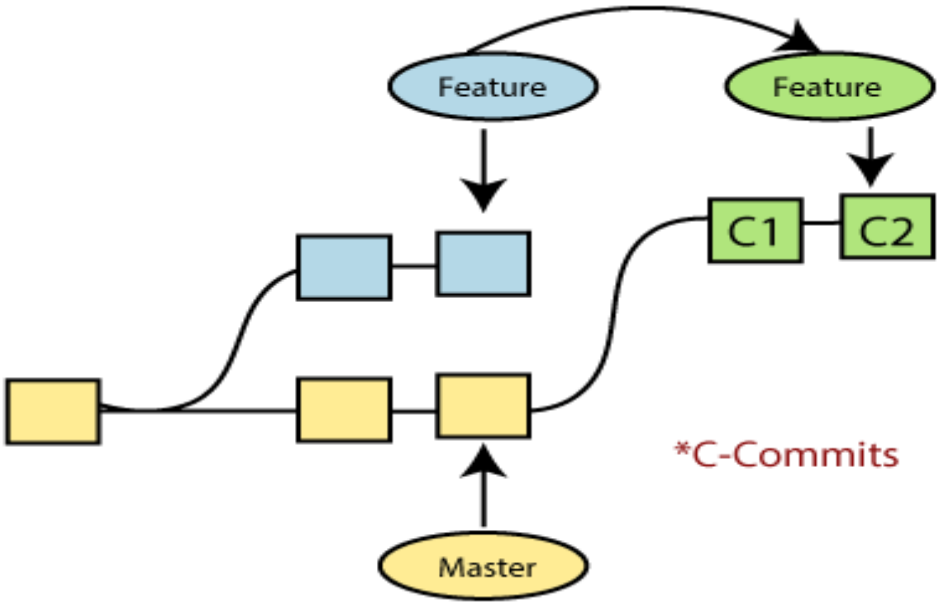


Figure18: Git rebase

Generally, it is an alternative of git merge command. Merge is always a forward changing record. Comparatively, rebase is a compelling history rewriting tool in git. It merges the different commits one by one.

Suppose you have made three commits in your master branch and three in your other branch named test. If you merge this, then it will merge all commits in a time. But if you rebase it, then it will be merged in a linear manner. Consider the below image:

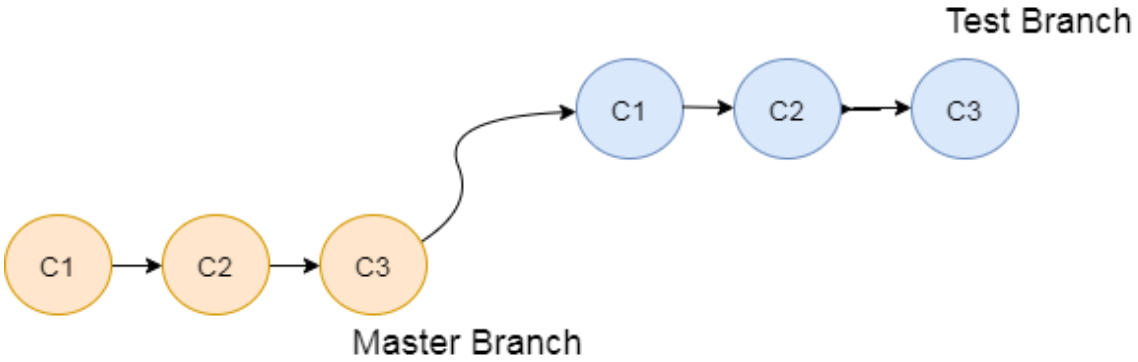


Figure 19 : Merging after git rebase

The above image describes how git rebase works. The three commits of the master branch are merged linearly with the commits of the test branch.
Merging is the most straightforward way to integrate the branches. It performs a three-way merge between the two latest branch commits.
When using the `git rebase` command in Git, there are several key operations and steps involved in the process:

1. Start a Rebase: Begin the rebase operation by running `git rebase <base-branch>`, where `<base-branch>` is the branch you want to rebase onto. This command sets the base for the rebase operation.
2. Resolve Conflicts: During the rebase process, Git may encounter conflicts if changes in the current branch conflict with changes in the base branch. You need to resolve these conflicts by editing the conflicting files, marking them as resolved, and continuing the rebase.
3. Continue Rebase: After resolving conflicts, continue the rebase process by using `git rebase --continue`. Git will apply the remaining commits on top of the base branch.
4. Abort Rebase: If you encounter issues during the rebase process and want to abort it, you can use `git rebase --abort` to return to the state before the rebase started.
5. Skip Commits: In some cases, you may want to skip applying certain commits during the rebase. You can do this by using `git rebase --skip` for the specific commit you want to skip.
6. Edit Commits: You can edit, squash, or reorder commits during the rebase process using interactive rebase by running `git rebase -i`.
7. Push Changes: Once the rebase is complete and the branch is updated with the new commit history, you can push the changes to the remote repository using `git push`.

2. Git Pull Request

Pull request allows you to announce a change made by you in the branch. Once a pull request is opened, you are allowed to converse and review the changes made by others. It allows reviewing commits before merging into the main branch.

Pull request is created when you committed a change in the GitHub project, and you want it to be reviewed by other members. You can commit the changes into a new branch or an existing branch.

Once you've created a pull request, you can push commits from your branch to add them to your existing pull request.

3. Git Merge and Merge Conflict

In Git, the merging is a procedure to connect the forked history. It joins two or more development history together. The git merge command facilitates you to take the data created by git branch and integrate them into a single branch. Git merge will associate a series of commits into one unified history. Generally, git merge is used to combine two branches.

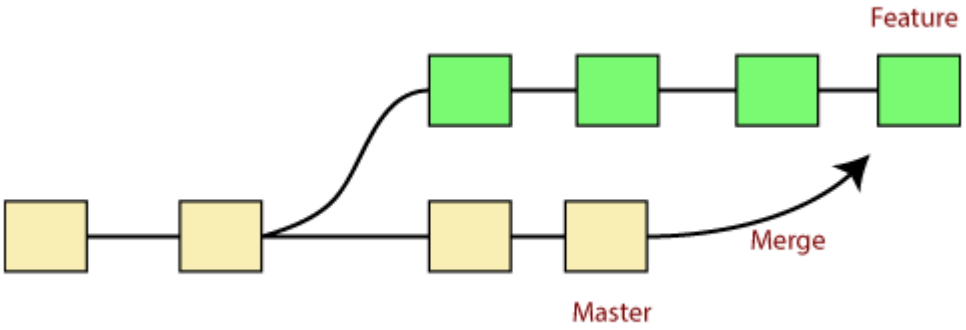


Figure 20: Merge branches

It is used to maintain distinct lines of development; at some stage, you want to merge the changes in one branch. It is essential to understand how merging works in Git.
In the above figure, there are two branches **master** and **feature**. We can see that we made some commits in both functionality and master branch, and merge them. It works as a pointer. It will find a common base commit between branches. Once Git finds a shared base commit, it will create a new "merge commit." It combines the changes of each queued merge commit sequence.

4. Git Merge Conflict

When two branches are trying to merge, and both are edited at the same time and in the same file, Git won't be able to identify which version is to take for changes. Such a situation is called merge conflict. If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually.

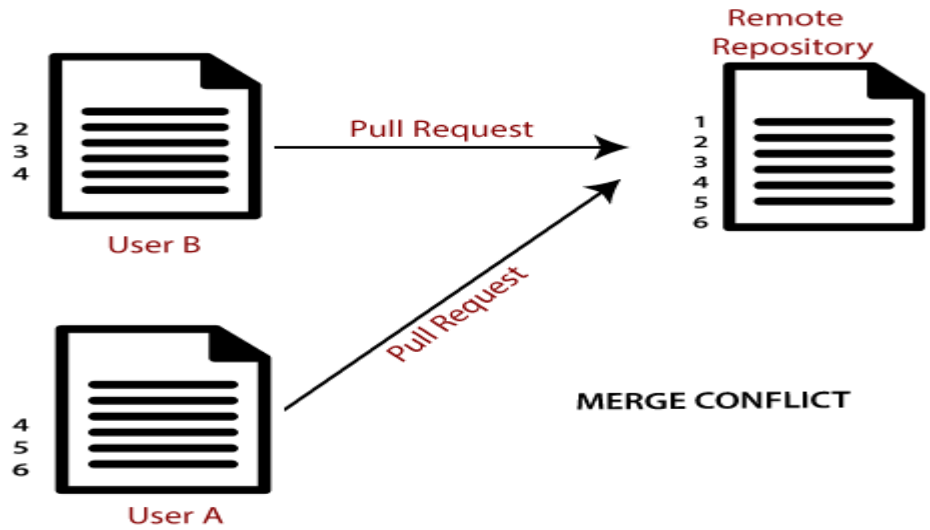


Figure 21: Merge conflict

Practical Activity 3.3.2: Merging branches on remote repository

Task:

1: Read key reading 3.3.2 and ask clarification where necessary

2: Referring to the previous theoretical activities (3.3.1) you are requested to go to the computer lab to merge branches on remote repository. This task should be done individually.

3: Apply safety precautions.

4: Present out the steps of merging branches on remote repository.

5: Referring to the steps provided on task 3, merge branches on remote repository.

6: Present your work to the trainer and whole class

Key readings 3.3.2: Applying Git merge commands

1. Merging branches

Merging branches on a remote repository involves several steps. Below are the general steps to merge branches on a remote repository, assuming you're using Git:

1. **Fetch Remote Changes:** Before merging branches, ensure you have the latest changes from the remote repository. Use the following command to fetch the latest changes:

```
git fetch origin
```

Replace origin with the name of your remote repository if it's different.

2.Checkout the Target Branch: Switch to the branch into which you want to merge the changes. For example, to merge changes into the master branch:

```
git checkout master
```

3. **Merge the Branch:** Merge the changes from the source branch into the target branch using the git merge command. For instance, if you want to merge changes from a branch named feature-branch into master:

```
git merge feature-branch
```

This command will merge the changes from feature-branch into the currently checked out branch (in this case, master).

4. **Resolve Conflicts (if any):** If there are conflicts between the changes in the branches being merged, Git will prompt you to resolve them. Open the conflicted files, resolve the conflicts, and then stage the changes.

5.**Commit the Merge:** After resolving conflicts, commit the merge changes:

```
git commit -m "Merge branch 'feature-branch' into master"
```

This commits the merge, incorporating the changes from the source branch into the target branch.

6.**Push Changes to Remote Repository**: Finally, push the merged changes to the remote repository:

```
git push origin master
```

Replace master with the name of your target branch if it's different. This command will update the remote repository with the merged changes.

7.**Delete Source Branch (Optional):** If you've finished with the source branch and no longer need it, you can delete it using:

```
git branch -d feature-branch
```

Or, to delete it remotely as well:

```
git push origin --delete feature-branch
```

This step is optional and should be performed only if you're certain the branch is no longer needed.

2. Git rebase

When you made some commits on a feature branch (test branch) and some in the master branch. You can rebase any of these branches. Use the git log command to track the changes (commit history). Checkout to the desired branch you want to rebase. Now perform the rebase command as follows:

Syntax:

$git rebase **<branch** name**>**

If there are some conflicts in the branch, resolve them, and perform below commands to

continue changes:

$ git status

It is used to check the status,

$git rebase --continue

The above command is used to continue with the changes you made. If you want to skip the change, you can skip as follows:

1. $ git rebase --skip

When the rebasing is completed. Push the repository to the origin. Consider the below example to understand the git merge command.

Suppose that you have a branch say **test2** on which you are working. You are now on the test2 branch and made some changes in the project's file **newfile1.txt**.

Add this file to repository:

1. $ git add newfile1.txt

Now, commit the changes. Use the below command:

1. $ git commit -m "new commit for test2 branch."

The output will look like:

[test2 a835504] new commit for test2 branch

1 file changed, 1 insertion(+)

Switch the branch to master:

1. **$ git checkout master**

Output:

Switched to branch 'master.'

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

Now you are on the master branch. I have added the changes to my file, says **newfile.txt**. The below command is used to add the file in the repository.

1. $ **git add newfile.txt**

Now commit the file for changes:

1. **$ git commit -m "new commit made on the master branch."**

Output:

[master 7fe5e7a] new commit made on master

1 file changed, 1 insertion (+)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)

2.1 Rebase Branch

If we have many commits from distinct branches and want to merge it in one. To do so, we have two choices either we can merge it or rebase it. It is good to rebase your branch.

From the above example, we have committed to the master branch and want to rebase it on the test2 branch. Let's see the below commands:

1. **$ git checkout test2**

This command will switch you on the test2 branch from the master.

Output:

Switched to branch 'test2.'

Now you are on the test2 branch. Hence, you can rebase the test2 branch with the master branch. See the below command:

1. **$ git rebase master**

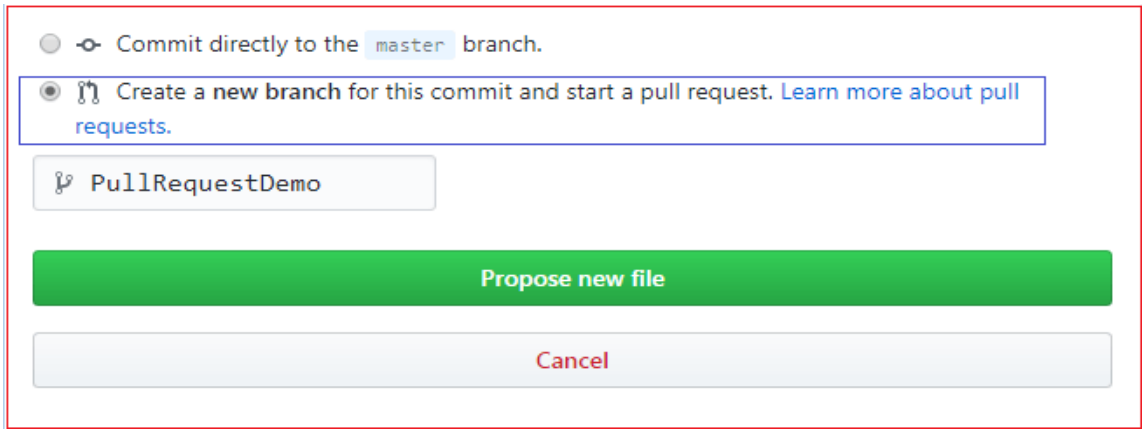
This command will rebase the test2 branch and will show as **Applying: new commit on test2 branch**. Consider the below output:

Output:

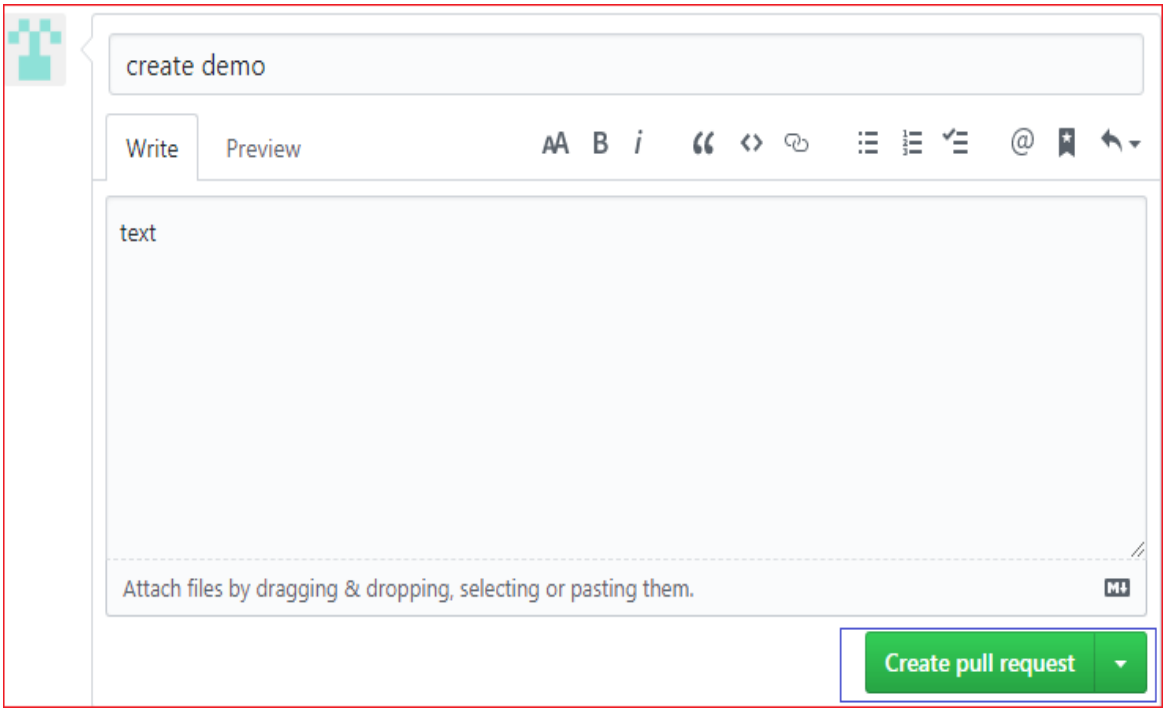
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git rebase master
First, rewinding head to replay your work on top of it...
Fast-forwarded test to master.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$
```

3. Pull Request

To create a pull request, you need to create a file and commit it as a new branch. As we mentioned earlier in this topic, how to commit a file to use git pull. Select the option "**create a new branch for this commit and start a pull request**" from the bottom of the page. Give the name of the new branch. Select the option to **propose a new file** at the bottom of the page. Consider the below image.



In the above image, I have selected the required option and named the file as **PullRequestDemo**. Select the option to propose a new file. It will open a new page. Select the option **create pull request**. Consider the below image:



Now, the pull request is created by you. People can see this request. They can merge this request with the other branches by selecting a merged pull request.

Use of "git merge" command

4. apply git merge operations

The git merge command is used to merge the branches.

The syntax for the git merge command is as:

1. $ git merge **<query>**

It can be used in various context. Some are as follows:

Scenario1: To merge the specified commit to currently active branch:

Use the below command to merge the specified commit to currently active branch.

1. $ git merge **<commit>**

The above command will merge the specified commit to the currently active branch. You can also merge the specified commit to a specified branch by passing in the branch name in <commit>. Let's see how to commit to a currently active branch.

See the below example. I have made some changes in my project's file **newfile1.txt** and committed it in my **test** branch.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git add newfile1.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git commit -m "edited newfile1.txt"
[test d2bb07d] edited newfile1.txt
 1 file changed, 1 insertion(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git log
commit d2bb07dc9352e194b13075dcfd28e4de802c070b (HEAD -> test)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 11:27:44 2019 +0530

    edited newfile1.txt

commit 2852e020909dfe705707695fd6d715cd723f9540 (test2, master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 10:29:07 2019 +0530

    newfile1 added
```

Copy the particular commit you want to merge on an active branch and perform the merge operation. See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git checkout test2
Switched to branch 'test2'

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git merge  d2bb07dc9352e194b13075dcfd28e4de802c070b
Updating 2852e02..d2bb07d
Fast-forward
 newfile1.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$
```

In the above output, we have merged the previous commit in the active branch test2.

Scenario2: To merge commits into the master branch:

To merge a specified commit into master, first discover its commit id. Use the log command to find the particular commit id.

1. $git log

See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git log
commit 2852e020909dfe705707695fd6d715cd723f9540 (HEAD -> test)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 10:29:07 2019 +0530

    newfile1 added

commit 4a6693a71151323623c04dd75cb0d44c1c4dbadf (origin/master, origin/HEAD, mas
ter)
Merge: 30193f3 78c5fbd
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Mon Sep 9 15:24:13 2019 +0530

    Merge pull request #1 from ImDwivedi1/branch2

    Create merge the branch
```

To merge the commits into the master branch, switch over to the master branch.

1. $ git checkout master

Now, Switch to branch 'master' to perform merging operation on a commit. Use the git merge command along with master branch name. The syntax for this is as follows:

1. $ git merge master

See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git merge 2852e020909dfe705707695fd6d715cd723f9540
Updating 4a6693a..2852e02
Fast-forward
 newfile.txt  | 1 +
 newfile1.txt | 1 +
 2 files changed, 2 insertions(+)
 create mode 100644 newfile.txt
 create mode 100644 newfile1.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

As shown in the above output, the commit for the commit

id ***2852e020909dfe705707695fd6d715cd723f9540*** has merged into the master branch. Two files have changed in master branch. However, we have made this commit in the **test** branch. So, it is possible to merge any commit in any of the branches.

Open new files, and you will notice that the new line that we have committed to the test branch is now copied on the master branch.

Scenario 3: Git merge branch.

Git allows merging the whole branch in another branch. Suppose you have made many changes on a branch and want to merge all of that at a time. Git allows you to do so. See the below example:

In the given output, I have made changes in newfile1 on the test branch. Now, I have committed this change in the test branch.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git commit -m "edit newfile1"
[test2 a3644e1] edit newfile1
 1 file changed, 1 insertion(+)
```

Now, switch to the desired branch you want to merge. In the given example, I have switched to the master branch. Perform the below command to merge the whole branch in the active branch.

1. $ git merge **<branchname>**

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git merge test2
Updating 2852e02..a3644e1
Fast-forward
 newfile1.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

As you can see from the given output, the whole commits of branch test2 have merged to branch master.

5. Git Merge Conflict

Let's understand it by an example.

Suppose my remote repository has cloned by two of my team member **user1** and **user2**. The user1 made changes as below in my projects index file.

```
.bash_profile | index.html | index.html
1  <head>
2  <body>
3    <title> This is a Git example</Title>
4    <h1> Git is a version control</h1>
5    </head>
6    </body>
```

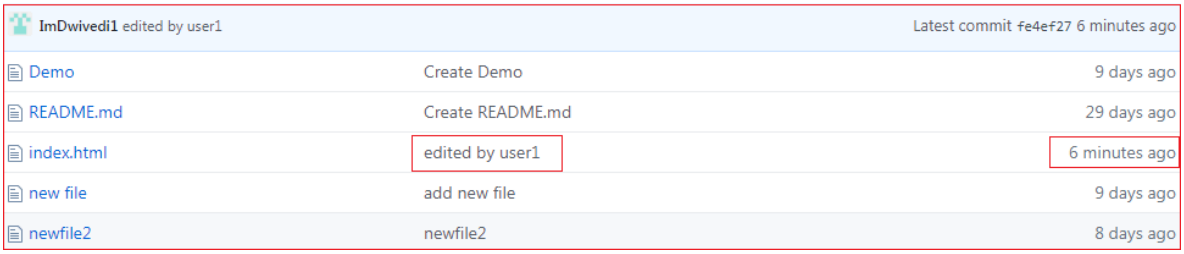
Update it in the local repository with the help of git add command.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
$ git add index.html
```

Now commit the changes and update it with the remote repository. See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
$ git commit -m "edited by user1"
[master fe4ef27] edited by user1
 1 file changed, 1 insertion(+)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 345 bytes | 345.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/ImDwivedi1/Git-Example
   039c01b..fe4ef27  master -> master

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
```

Now, my remote repository will look like this:



It will show the status of the file like edited by whom and when.

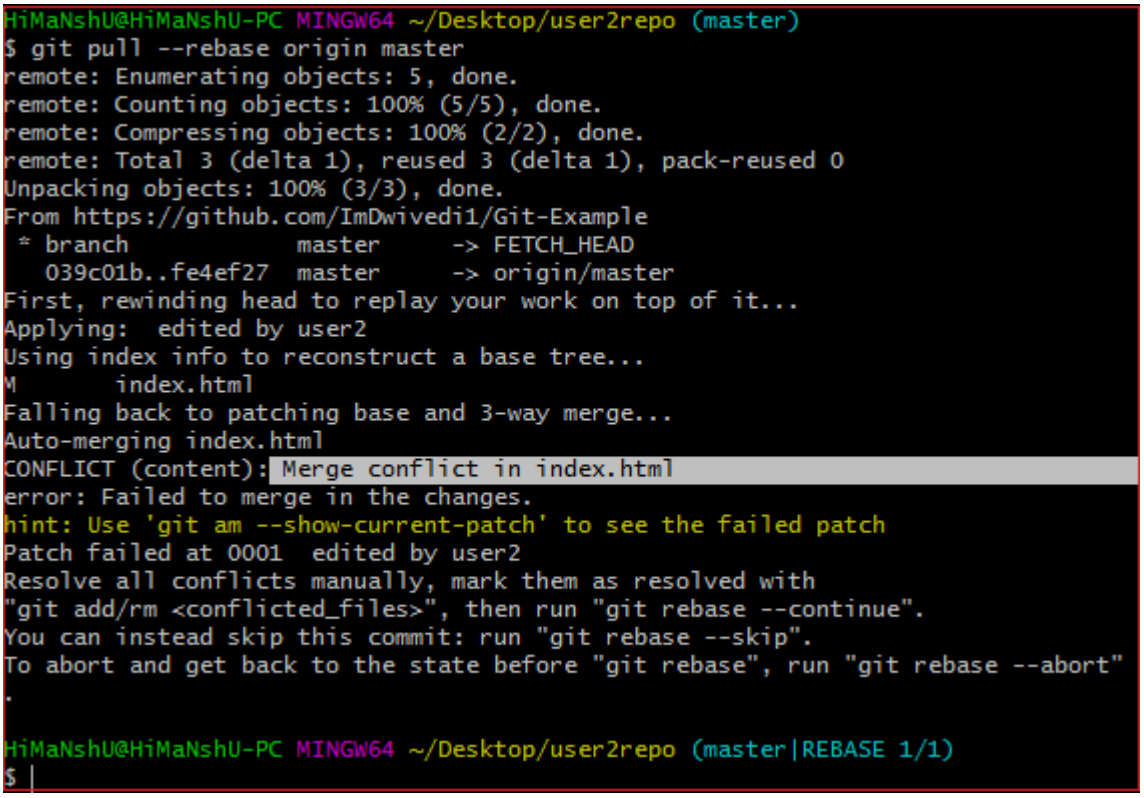
Now, at the same time, **user2** also update the index file as follows.

```
.bash_profile | index.html | index.html
1  <head>
2  <body>
3   <title> This is a Git example</Title>
4   <h2> Git is a version control system</h2>
5  </head>
6  </body>
```

User2 has added and committed the changes in the local repository. But when he tries to push it to remote server, it will throw errors. See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$ git add index.html

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$ git commit -m " edited by user2"
[master 3ee71e0]  edited by user2
 1 file changed, 1 insertion(+)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$ git push origin master
To https://github.com/ImDwivedi1/Git-Example
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/ImDwivedi1/Git-Example'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$
```

In the above output, the server knows that the file is already updated and not merged with other branches. So, the push request was rejected by the remote server. It will throw an error message like **[rejected] failed to push some refs to <remote URL>**. It will suggest you to pull the repository first before the push. See the below command:



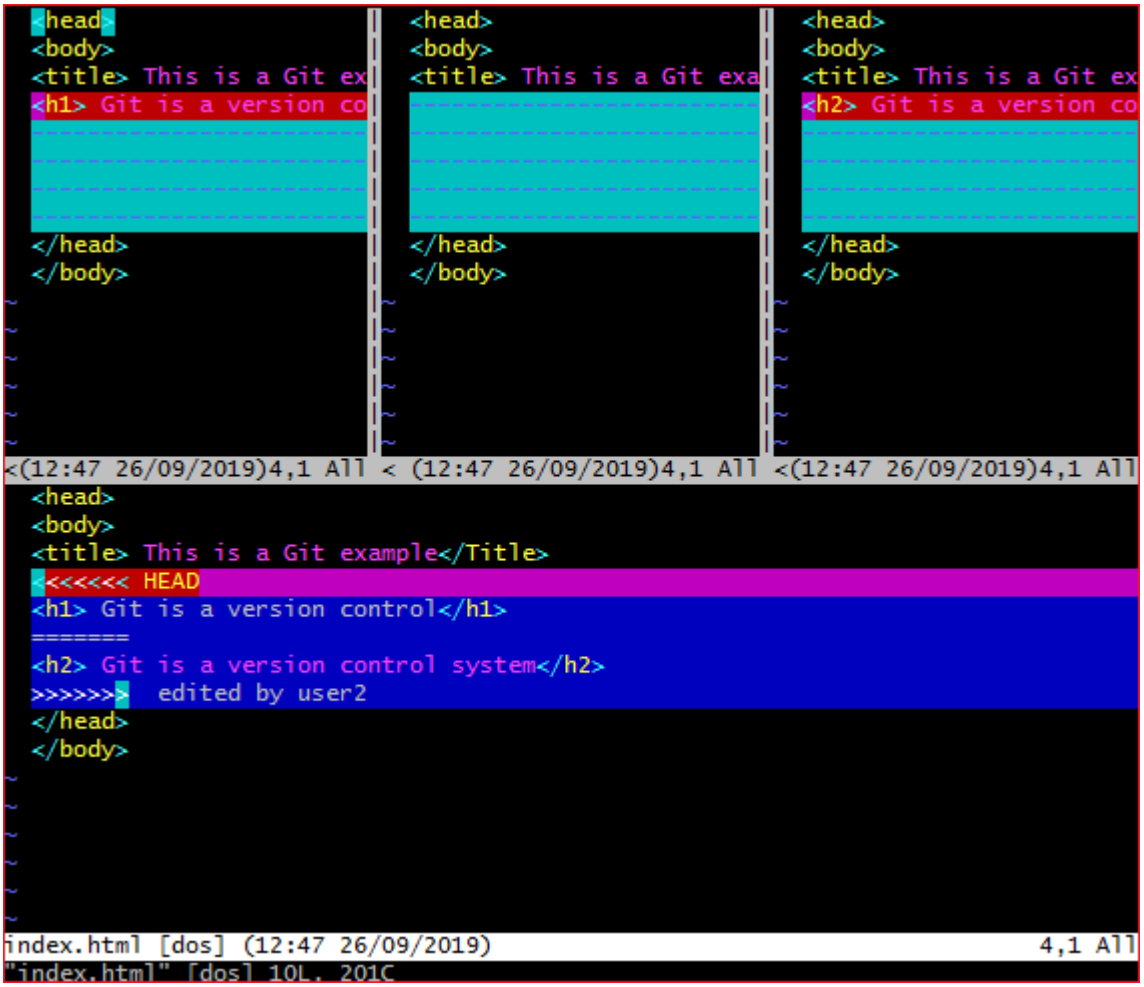
In the given output, git rebase command is used to pull the repository from the remote URL. Here, it will show the error message like **merge conflict in <filename>**.

5.1 Resolve Conflict:

To resolve the conflict, it is necessary to know whether the conflict occurs and why it occurs. Git merge tool command is used to resolve the conflict. The merge command is used as follows:

1. $ git mergetool

In my repository, it will result in:



The above output shows the status of the conflicted file. To resolve the conflict, enter in the insert mode by merely pressing **I key** and make changes as you want. Press the **Esc key**, to come out from insert mode. Type the: **w!** at the bottom of the editor to save and exit the changes. To accept the changes, use the rebase command. It will be used as follows:

1. $ git rebase --continue

Hence, the conflict has resolved. See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master|REBASE 1/1)
$ git rebase --continue
Applying:  edited by user2

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 373 bytes | 124.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/ImDwivedi1/Git-Example
   fe4ef27..b3db7dc  master -> master

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
```

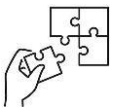
In the above output, the conflict has resolved, and the local repository is synchronized with a remote repository.

To see that which is the first edited text of the merge conflict in your file, search the file attached with conflict marker **<<<<<<<**. You can see the changes from the **HEAD** or base branch after the line **<<<<<<< HEAD** in your text editor. Next, you can see the divider like **=======**. It divides your changes from the changes in the other branch, **followed by >>>>>>> BRANCH-NAME**. In the above example, user1 wrote "<h1> Git is a version control</h1>" in the base or HEAD branch and user2 wrote "<h2> Git is a version control</h2>".

Decide whether you want to keep only your branch's changes or the other branch's changes, or create a new change. Delete the conflict markers **<<<<<<<, =======, >>>>>>>** and create final changes you want to merge.

Points to Remember

- The `git rebase` command in Git allows you to reapply a series of commits on top of a nother base commit, effectively moving the entire branch to begin from the tip of an other branch. Key operations involved in using `git rebase` include starting a rebase,R esolve Conflicts,Continue Rebase,Abort Rebase,Skip Commits,Skip Commits.
- **To merge branches on a remote repository in Git, you typically follow these steps:**
  1.Fetch Remote Changes
  2. Checkout the Target Branch
  3.Merge the Branch
  4.Push Changes to Remote Repository



Application of learning 3.3.

MyApp is a web application project targeted at enhancing the user authentication procedure that is being worked on by a team of developers. The following actions were carried out by the appropriate team members. The developer, John, used the "git rebase" command to merge the most recent modifications from the main branch into his feature branch, ensuring that his improvements were based on the most recent codebase. Sarah, the team leader, issued a pull request to merge John's branch into the main branch, allowing for easier collaboration and review. This enabled the team to handle the issue as a whole and confirm that the proposed solution was in line with the project's goals and criteria. Finally, after reading and approving “this is merging exercise” pull request, David as project manager use the merge procedure, thereby Implementing the upgraded user authentication feature properly and contributing to the overall solution of improving the application's security and user experience.

Learning outcome 3 end assessment

Theoretical assessment

1. Which of these Git client commands creates a copy of the repository and a working directory in the client's workspace. (Choose one.)
    a) update
    b) checkout
    c) clone
    d) import
    e) None of the above
2. True or False? In Git, if you want to make your local repository reflect changes that have been made in a remote (tracked) repository, you should run the pull command.
    a) True
    b) False
3. Now, imagine that you have a local repository, but other team members have pushed changes into the remote repository. What Git operation would you use to download those changes into your working copy?
    a) checkout
    b) commit
    c) export
    d) pull
    e) update
    f) a, b, and c

4. fill-in-the-blank space related to fetching, pulling, and pushing files in Git
    a) To fetch changes from a remote repository without merging them into your local branch, you use the command `git _____`.
    b) To fetch changes from a remote repository and merge them into your local branch, you use the command `git _____`.
    c) To upload your local repository changes to a remote repository, you use the command `git _____ origin <branch-name>`.
    d) To see the status of your local repository, including which branch you are on and any changes that are staged for commit, you use the command `git _____`.
    e) To add all changes in your working directory to the staging area, you use the command `git _____ `.
    f) To commit your staged changes with a message, you use the command `git _____ -m "Your commit message"`.
    g) To create a connection to a new remote repository, you use the command `git remote _____ <name> <URL>`.
    h) To view the details of your remote repository connections, you use the command `git remote _____`.

5. Match the operations with corresponding commands

Operation	Command
Fetch changes from remote repository	a. git fetch
Fetch and merge changes from remote	b. git pull
Push changes to remote repository	c. git push origin <branch-name>
See the status of the local repository	d. git status
Add all changes to the staging area	e. git add .
Commit staged changes with a message	f. git commit -m "Your commit message"
Create a connection to a remote repo	g. git remote add <name> <URL>

Practical assessment

ABC Software Solutions is a leading software development company, Alex and Sarah are two experienced developers who need to collaborate on a project aimed at resolving a critical bug in a mission-critical software system for a major client. The bug has caused disruptions in the client's operations, and immediate attention is required to rectify the issue. Alex takes on the responsibility of simulating updates from the client and making changes to a separate branch to identify the root cause of the bug. Additionally, Alex merges the client's changes into their branch to test potential fixes and improve the overall stability of the system. Throughout the process, Alex actively contributes by making modifications, committing them, and pushing their branch to the remote repository, ensuring that the bug fixes are well-documented and can be easily shared with the client. Sarah, an integral team member, communicates her additional changes to Alex, addressing specific edge cases and proposing enhancements to optimize the software system's performance. After pushing her branch to the remote repository, Alex ensures synchronization by fetching Sarah's updated branch and merging it into their own, incorporating the valuable contributions made by Sarah. ABC Software Solutions demonstrates their commitment to providing high-quality software solutions and meeting their client's critical needs by leveraging efficient collaboration, version control, and code management practices to swiftly address and resolve complex problems in their client's software systems.

References

JavaTpoint. (n.d.). Git Pull. Retrieved from https://www.javatpoint.com/git-pull

JavaTpoint. (n.d.). Git Push. Retrieved from https://www.javatpoint.com/git-push

Varonis. (n.d.). Git Branching: A guide to working with branches. Retrieved from https://www.varonis.com/blog/git-branching

TutorialsPoint. (n.d.). Git Managing Branches. Retrieved from https://www.tutorialspoint.com/git/git_managing_branches.htm

Atlassian. (n.d.). Git Merge: Everything you need to know about merge in Git. Retrieved from https://www.atlassian.com/git/tutorials/using-branches/git-merge

RTB | RWANDA TVET BOARD

October , 2024