### Practical File

### of

### Source Code Management

### (22CS003)

#### Submitted in

#### partial fulfillment for the award of the degreeof

## BACHELOR OF ENGINEERING

***in***

COMPUTERN SCIENCE & ENGINEERING

****

**CHITKARA UNIVERSITY INSTITUTE OF ENGINEERING &TECHNOLOGY**

**CHANDIGARH-PATIALA NATIONAL HIGHWAY**

**RAJPURA (PATIALA) PUNJAB-140401 (INDIA)**

##### **Submitted To: Submitted By:**

##### Faculty name: Dr Kalpna Guleria Student Name: Raghav Arora

##### Designation: Professor (Research) Roll No: 2310991343

##### Chitkara University, Punjab Sem: 2ndBatch:2023

##### Session: 2023-2024

**LIST OF PRACTICALS**

| **S.No.** | **Date** | **Practical** | **Page No.** | **Signature** |
| --- | --- | --- | --- | --- |
| 1. | 11-02-2024 | Introducing Version Control – Git client (CLI, GUI), Linux environment Emulation, Advantages of VCS, Installing git CLI and git GUI | 3-9 |  |
| 2. | 11-02-2024 | Setting up of the Github Account and linking Github account with gitbash | 10-12 |  |
| 3. | 11-02-2024 | Creating a repository, Adding files, checking status,  staged files/ untraced files and performing Commits | 13-16 |  |
| 4. | 11-02-2024 | Generating logs: Visualization of various git logs | 17-19 |  |
| 5. | 11-02-2024 | Git branching and Merging: Visualization of git Branch and HEAD, Git branches management, Create a new branch, Commit changes in the new branch, Explore commit in the new branch, Merging the branches | 20-24 |  |
| 6. | 11-02-2024 | Git lifecycle description Git status, add, commit, stage – Life cycle of a file in Git managed in Repos. | 24-27 |  |

**EXPERIMENT 1**

**AIM:** Introducing Version Control – Git client(CLI, GUI), Linux environment Emulation, Advantages of VCS, Installing git CLI and git GUI

**Theory:**

**Version Control System:**

Version control systems (VCS) are essential tools in software development, enabling teams to manage changes to source code efficiently. Git is one of the most popular distributed version control systems.As development environments have accelerated ,version control systems help software teams work faster and smarter. It records all the changes made to a file or a set of files, so a specific version may be called later if needed.

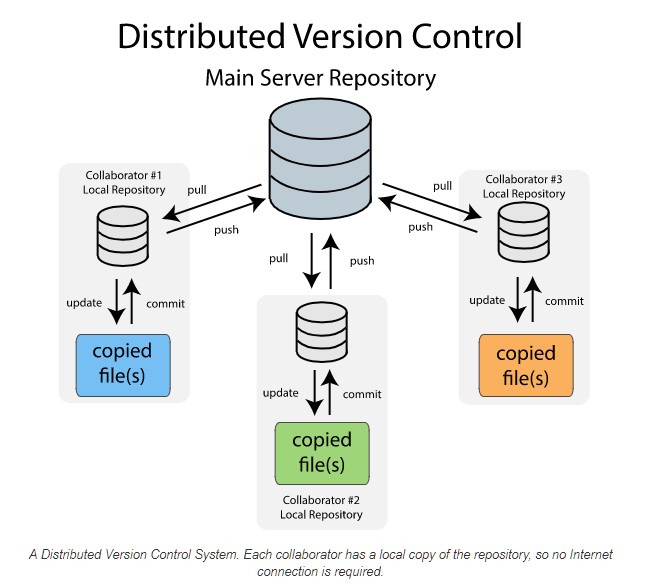
**History of VCS:**

The very first Version Control System was created in 1972 at Bell Labs where they also developed UNIX. The first one was called SCCS (Source Code Control System). It was available only for UNIX and only worked with Source Code files. Some types of Version Control Systems are:

• Local VCS: No internet is needed because it uses a database to keep and track of files.

• Centralized VCS: Centralized version control systems are based on the idea that there is a single “central” copy of your project somewhere (probably on a server), and programmers will “commit” their changes to this central copy. This simply means recording the change in the central system (OS).

• Distributed VCS: A type of version control where the complete codebase including its full version history is mirrored on every developer's computer.



Git is a distributed version control system designed to handle everything from small to very large projects with speed and efficiency. Unlike centralized version control systems (CVCS) where a single, central server stores all versions of the code and team members must communicate with this central server to access and manage files, Git allows every developer to have their own local repository, complete with a full history of commits. This decentralized approach provides several advantages, including faster access to project history, the ability to work offline, and greater resilience to server failures.

With Git, each developer maintains their own local repository, which contains the entire project history along with all branches and commits. Developers can work independently on their local repositories, making changes, creating branches, and experimenting with new features without impacting the work of others. When ready, changes can be synchronized with remote repositories, allowing for seamless collaboration with other team members.

**Explanation of Key Git Concepts:**

**Repositories:**

A Git repository, or repo, is a collection of files and directories associated with a specific project, along with the entire history of changes made to those files. Each repository can exist either locally on a developer's machine or remotely on a server (e.g., GitHub, GitLab). Repositories can be cloned, allowing developers to create local copies of remote repositories to work on.

**Commits:**

A commit in Git represents a snapshot of the project at a specific point in time. It includes the changes made to the files since the last commit, along with metadata such as the author, timestamp, and a unique identifier(SHA-1hash). Commits are the building blocks of project history and provide a way to track changes over time.

**Branches:**

A branch in Git is a lightweight movable pointer to a specific commit. Branches allow developers to work on multiple independent lines of development simultaneously. The main branch in Git is typically called "master" (or "main" in more recent conventions), and developers create new branches from this main branch to work on new features or bug fixes. Branches can be merged back into the main branch once the changes are complete and tested.

**Merges:**

Merging is the process of combining the changes from one branch into another. When a feature branch is ready to be incorporated into the main branch, developers can merge the changes using Git's merge functionality. Git automatically identifies and resolves any conflicts that arise from overlapping changes between branches, allowing for a seamless integration process.

**Git Client:**

A Git client is a software tool used to interact with Git repositories, allowing users to perform various version control tasks such as cloning repositories, making changes to files, staging and committing changes, creating branches, merging branches, and interacting with remote repositories.

Here are some common types of Git clients:

**1.Command-Line Interface (CLI):**

The Git CLI is the official command-line interface for Git and comes installed with Git itself. It provides a powerful set of commands for interacting with Git repositories directly from the terminal or command prompt. Some commonly used Git commands include git clone, git add, git commit, git push, git pull, git branch, and git merge.

**2.Graphical User Interface (GUI):**

GUIs provide a visual interface for managing Git repositories, making it easier for users who prefer graphical tools over the command line. GUI clients typically offer features like visual commit history graphs, file diff viewers, branch management tools, and integration with popular Git hosting services like GitHub, GitLab, and Bitbucket.

**Linux Environment Emulation:**

1. **Introduction to Emulation:**

Emulation allows you to run software or operating systems on a different platform than the one for which they were originally designed. Emulating a Linux environment can provide a more consistent and predictable development environment, especially for Git operations.

2. **Tools for Emulation:**

• VirtualBox: A free and open-source virtualization platform that allows you to create and manage virtual machines.

•VMware: A commercial virtualization solution with advanced features for emulating various operating systems.

3. **Benefits of Using Linux for Git:**

•Better compatibility: Many Git-related tools and tutorials are designed for Linux systems.

•Performance: Linux tends to offer better performance for Git operations compared to Windows or macOs.

**Advantages of Version Control System**

**1. History Tracking:** VCS maintains a detailed history of changes made to the code base, including who made each change, when it was made, and what was modified. This historical record enables developers to understand how the code has evolved over time and facilitates tracking down the source of bugs or regression

**2. Collaboration:** VCS enables multiple developers to work on the same codebase simultaneously, regardless of geographical location. It allows them to track each other's changes, merge modifications, and resolve conflicts. This fosters collaboration, accelerates development, and ensures that team members are always working with the latest version of the code.

**3. Branching and Merging:** VCS allows developers to create branches to work on new features or experiments without affecting the main codebase. Branches provide isolation for development work, allowing developers to iterate on changes independently. Once the changes are complete, they can be merged back into the main branch, facilitating parallel development and code review processes.

**4. Code Revert and Rollback:** VCS enables developers to revert to previous versions of files or entire codebase if needed. This is particularly useful for undoing changes that introduced bugs or undesirable behavior. By maintaining a history of changes, VCS provides a safety net that allows developers to experiment and make changes with confidence.

**5. Backup and Recovery:** VCS serves as a backup mechanism for code, ensuring that changes are not lost and can be recovered in case of accidental deletions, system failures, or data corruption. By storing the codebase in a centralized repository or distributed copies, VCS provides redundancy and safeguards against data loss.

**6. Rollback:** VCS allows developers to revert to previous versions of files or the entire project if needed, providing a safety net for experimentation

**Need of Version Control System (VCS)**:

Once upon a time, in a small startup company, there was a brilliant software development team working on a groundbreaking product. The team consisted of four talented developers, and they were all committed to building the best product they could. At first, they managed their code using a simple shared folder on their network drive. Whenever a developer wanted to make a change, they would copy the relevant files to their local machine, make the changes, and then overwrite the files in the shared folder. This process

worked reasonably well when the team was small, and they were only making minor changes to the code base. However, as the team grew and the project became more complex, managing the code base using a shared folder became increasingly difficult. Changes were being overwritten, and it was becoming impossible to keep track of who had made which changes. The developers spent more and more time trying to resolve conflicts and untangle the mess of code in the shared folder. The team realized they needed a better way to manage their code base, so they started looking for a solution. They discovered version control systems (VCS) and quickly realized how they could benefit from using one

**Installing Git CLI and Git GUI**

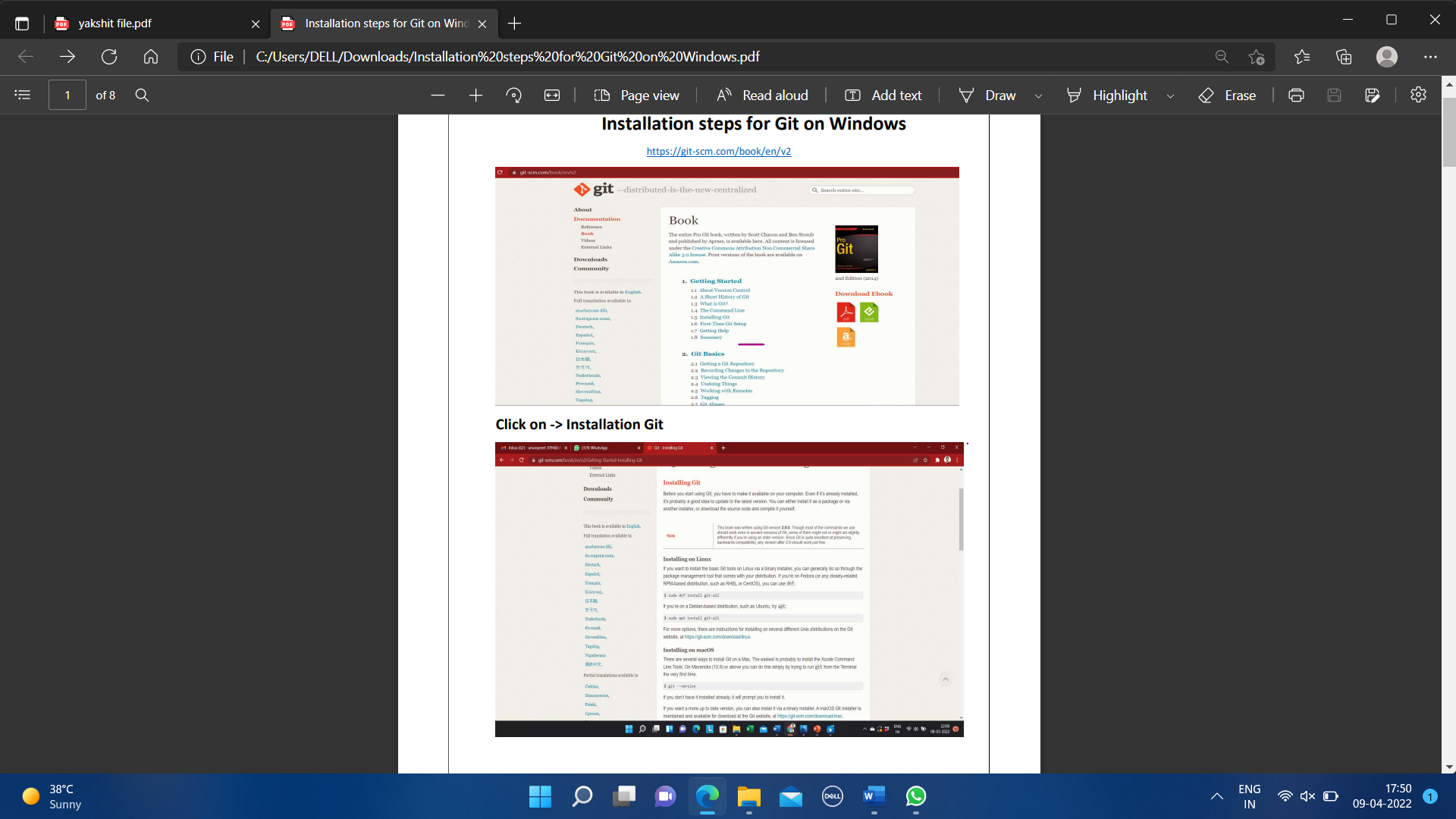
**1. Installing Git CLI:**

* Windows: Download the Git for Windows installer from the official website and follow the installation instructions. Ensure that Git is added to the system PATH during installation.
* macOS: Install Git using Homebrew by running brew install git in the terminal.
* Linux: Use the package manager of your Linux distribution (e.g., apt for Ubuntu, yum for CentOS) to install Git.
* **Benefits of Git CLI:**

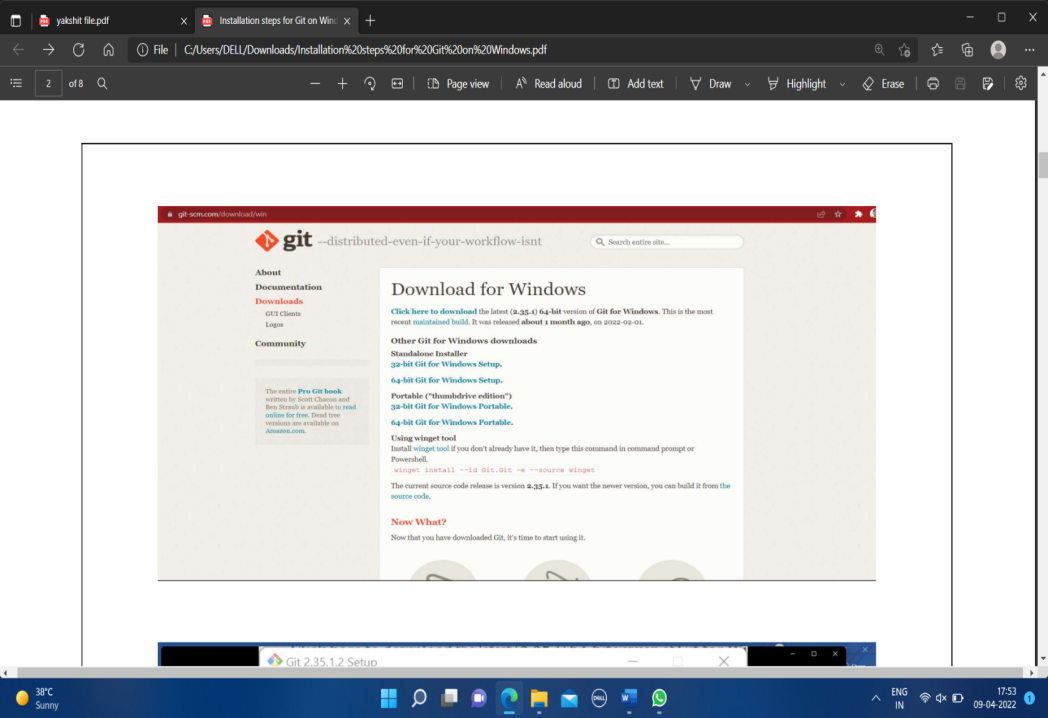
Git CLI offers efficiency and flexibility, allowing users to perform a wide range of Git operations through command-line commands.

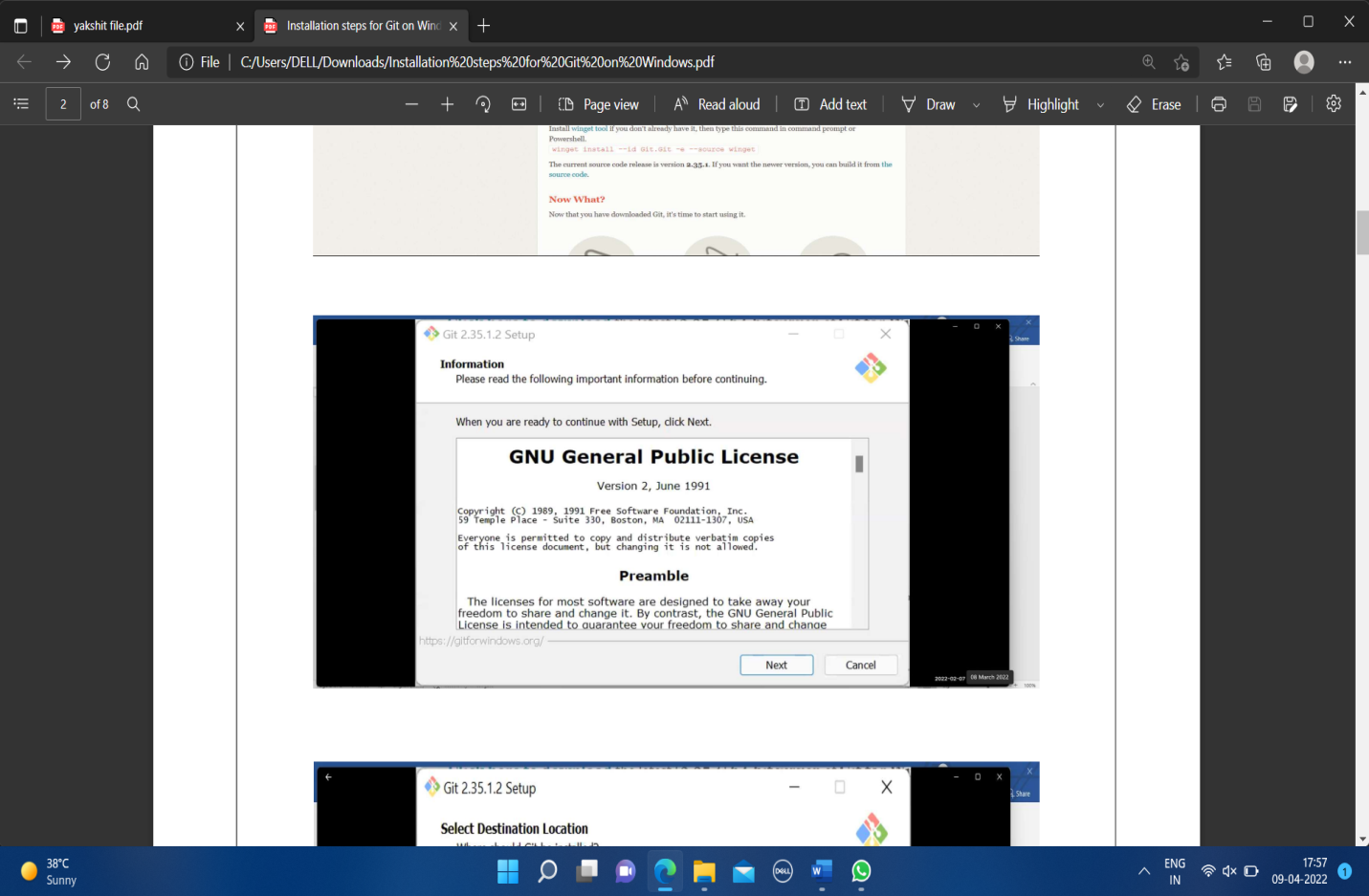
**2. Installing Git GUI:**

* GitHub Desktop: Download and install GitHub Desktop from the GitHub website. Follow the setupwizard to configure the application.
* Visitdirectly on git book page by <https://git-scm.com/book/en/v2>

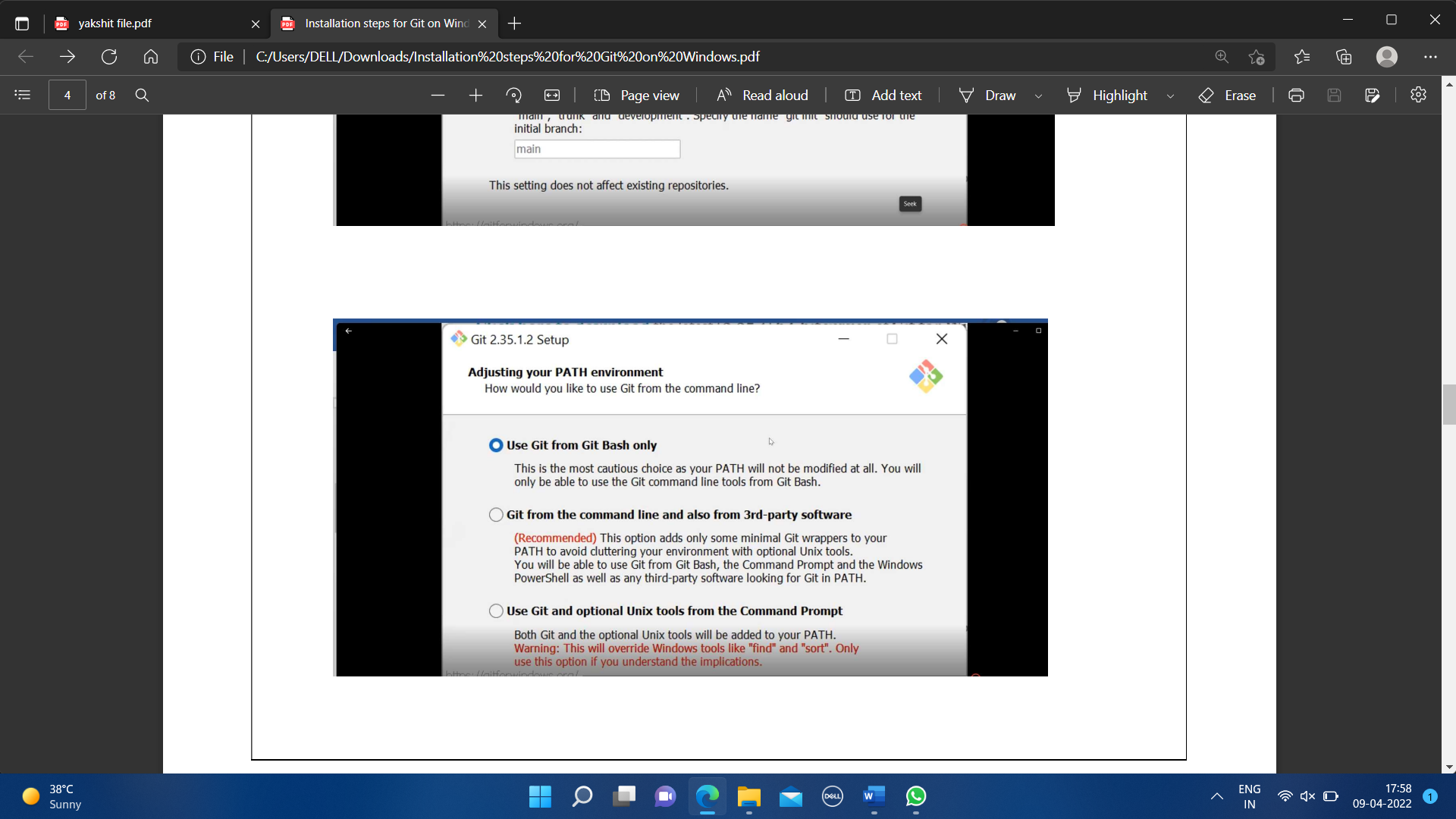


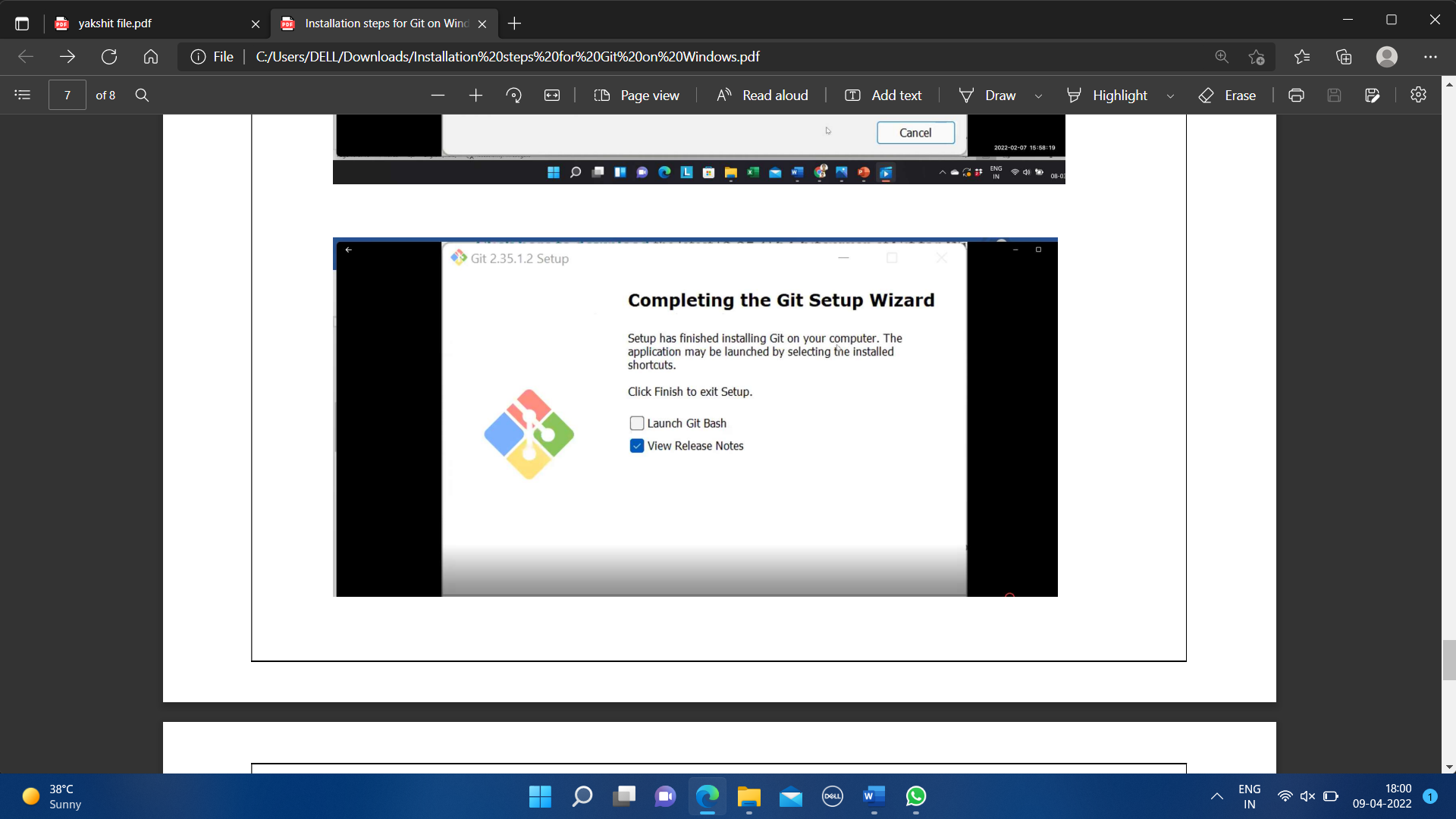
* Then click on Installation Git and click on whatever system you want, available are three- Windows, Apple and Linux.



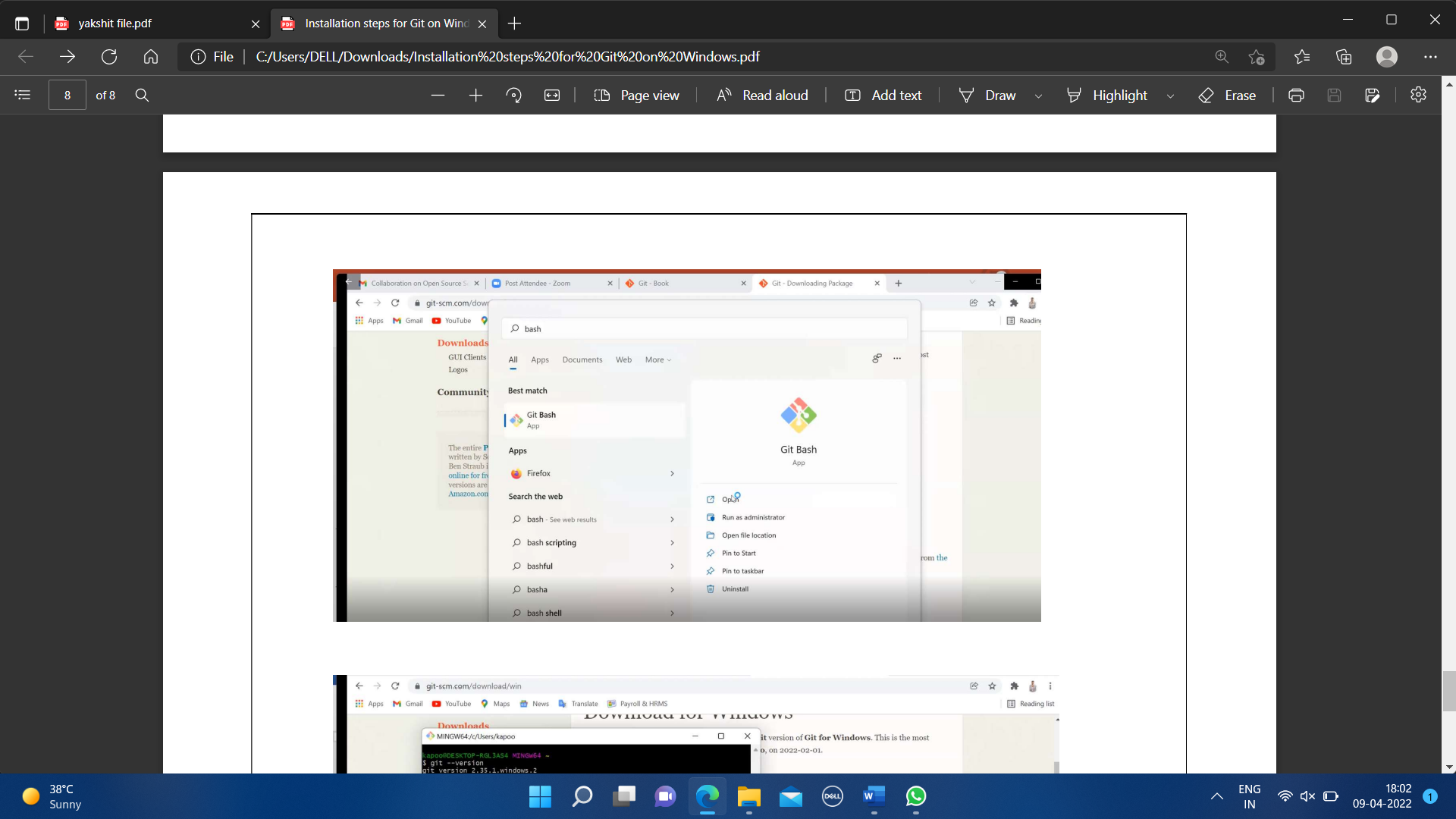


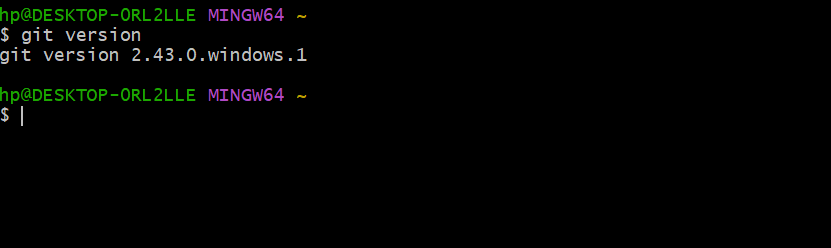
* After some more simple and easy settings and choosing your favourable environment and doing some SSH settings, it finally starts exporting the files in system and completes the Git hub wizard.





* Git bash got installed in system and seemed and opened on clicking seems of like:





* You can also check the version of installed software by checking git version.

**EXPERIMENT NO. 2**

**AIM:** Setting up of the Github Account and linking Github account with gitbash

**GITHub Account Setup:**

**What is GITHub?**

GitHub is a web-based platform built around Git, the distributed version control system. It serves as a hosting service for Git repositories and provides a wide range of collaboration tools for software development projects.

GitHub is a **cloud-based platform for software developers** to store, track, and collaborate on code projects.

It offers several key features and advantages:

**1. Version Control:**

* At the heart of GitHub lies **Git** - a version control system that tracks changes made to code files over time. This allows developers to:
  + **See the history of changes** made to the code, allowing them to revert to previous versions if needed.
  + **Collaborate** on projects by working on different parts of the code simultaneously without interfering with each other.

**2. Collaboration:**

* GitHub facilitates easy collaboration among developers:
  + **Teams can create and share repositories (repos)** where their code is stored.
  + **Features like branches** allow individuals to work on specific code sections without affecting the main project.
  + Developers can **communicate and review code changes** through features like pull requests and issue tracking.

**3. Open Source Development:**

* GitHub is a central hub for **open-source software (OSS)** projects. These are projects whose code is publicly accessible and can be freely used and modified by anyone.

This fosters:

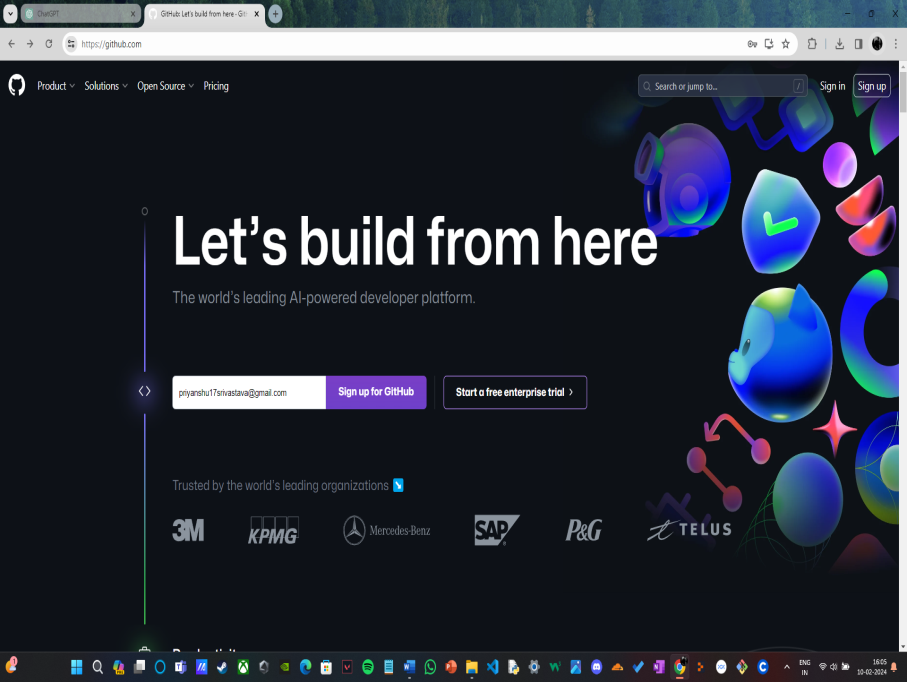
* + **Collaboration on large projects** by developers worldwide.
  + **Innovation and learning** as individuals can contribute to existing projects and build upon them.
  + **Transparency and trust** in the development process as the code is openly available for review.

**4. Additional Advantages:**

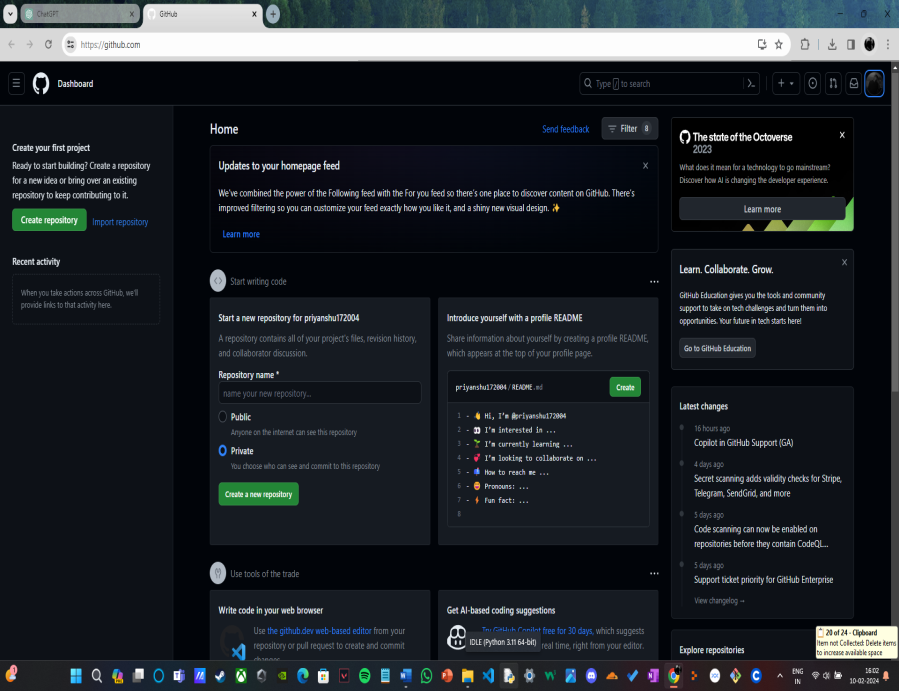
* **Social Networking:** GitHub functions as a social network for developers, allowing them to follow other users, contribute to various projects, and build their reputation.
* **Version control for other file types:** While primarily used for code, GitHub can also host other types of files, offering version control benefits for documents, configurations, or any type of project data.
* **Learning Resources:** GitHub offers resources and tutorials for beginners to learn about Git, coding, and various technical topics.

**In summary, GitHub empowers developers by providing a platform for:**

* Securely storing and tracking their code
* Collaborating effectively with others
* Contributing to and benefiting from the open-source community
* Learning and growing their technical skills
* **Installation Procedure**
* Search about GitHub: <https://github.com/signup>.



* By signing up for git you must remember your email and pass phases or password. For a new user, you must add your email and click on Sign up for GitHub. Otherwise click on Sign In at the top right corner.



**For linking Git Hub with Git Bash:**

**1. Setting your username:**

* This command sets your **identification name** for Git Bash. Replace "username in github" with your actual GitHub username.

**2. Setting your email:**

* This defines your **contact email** used by Git Bash. Replace "your email in github" with the email address associated with your GitHub account.

**3. Checking your configuration:**

* These commands **display your currently configured username and email** in Git Bash, allowing you to verify if they match your GitHub information.



**EXPERIMMENT NO. 3:**

AIM: Creating repositories and performing different fucntions on them (Init, Staging, Add, Status, Commit):

In GitHub, a **repository (often shortened to "repo")** is the **heart and soul of your project**. It's like a **secure digital storage locker** where you keep all your project's important files, code, and assets. Here's why they're crucial and where they find their use:

**Why are repositories useful?**

* **Version control:** Imagine constantly saving different versions of your project on your computer, hoping you don't accidentally overwrite the better one. Repositories offer a much more efficient way to keep track of changes. They act like a **time machine** for your project, allowing you to:
  + **See the history of changes** made to each file, allowing you to revert to previous versions if needed.
  + **Collaborate** with others without worrying about conflicting changes.
* **Collaboration:** Working on a project with multiple developers? Repositories offer a **centralized platform** for everyone to access, work on, and share the same codebase seamlessly. Imagine building a Lego castle together, each person adding their pieces without stepping on each other's builds.
* **Sharing and visibility:** Repositories can be **public or private**. Public repositories allow anyone to see your code, download it, and even contribute to it if you allow it. This fosters **open-source development** and collaboration on a global scale. Private repositories keep your code confidential but still allow you to share it with specific individuals or teams.

**Where are repositories used?**

* **Software development:** Repositories are the **backbone of software development**, used by individuals and teams to manage their code, collaborate effectively, and track progress.
* **Open-source projects:** Countless open-source projects, like Linux, utilize GitHub repositories to share their code with the world, allowing anyone to contribute, learn, and build upon their work.
* **Document management:** While primarily used for code, repositories can also be used to manage other types of files, such as documents, configurations, or any type of project data, allowing version control and collaboration for non-coding projects as well.
* **Personal projects:** Even if you're working on a personal project alone, repositories offer a safe and organized way to store your work, track your progress, and revert to previous versions if needed.

In a nutshell, repositories are the **foundational building blocks** for collaborative work and organized project management on GitHub. They offer a powerful way to **track changes, share code, and keep your projects secure and organized**.

**COMMANDS:**

cat><filename>: Create a new file with the specified filename and add content to it.

cat>><filename>: Append content to an existing file with the specified filename.

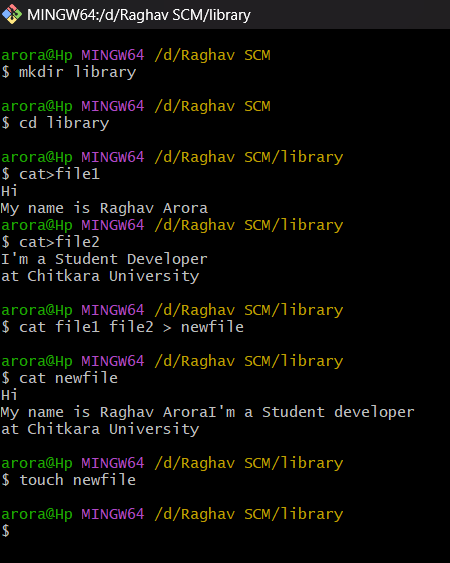
rm <filename>: Delete the file with the specified filename.

mkdir <directory name>: Create a new directory with the specified name.

rmdir <directory name>: Remove the directory with the specified name (if it's empty).

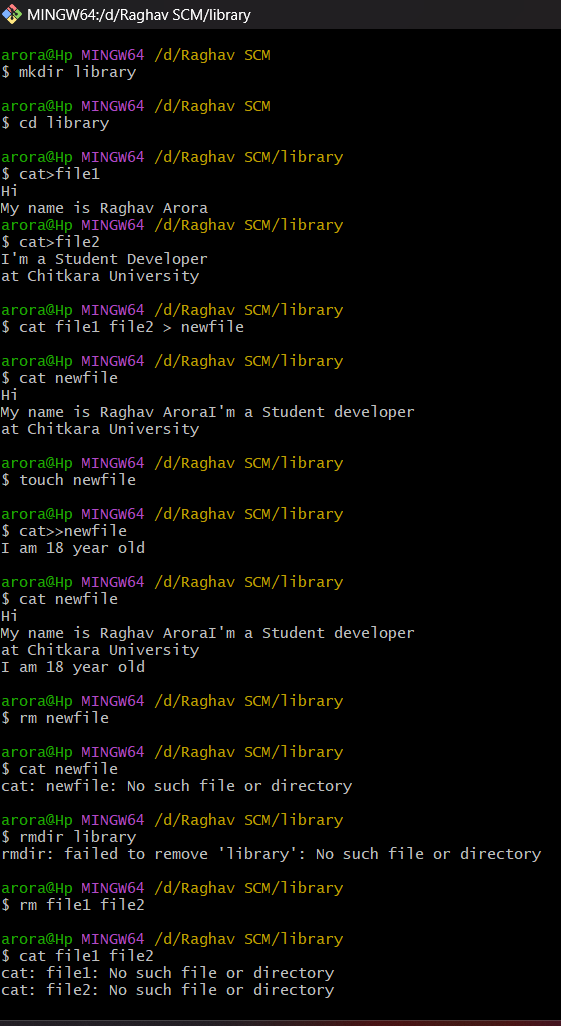
cd <directory name>: Change your current directory to the one specified.

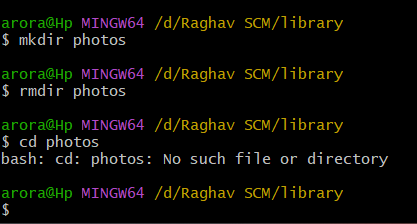
touch <filename>: Create an empty file with the specified filename, or update the file's access and modification timestamps.

****

**General commands for creating repositories:**

* **git init:** Initializes a new Git repository in the current directory.
* **git status:** Shows the current status of your files, including untracked and modified ones.
* **git add <file/directory name>:** Adds a specific file or directory to be tracked by Git.
* **git add:** Adds all modified and tracked files in the current directory to be committed.
* **git commit -m "message":** Creates a snapshot of the changes you've added, along with a message describing the changes.
* **git rm --cached <file/directory name>:** Schedules a file/directory for removal from the next commit, but keeps it in your working directory.
* **git restore --staged <filename>:** Recovers a deleted file from staging, but keeps it marked for deletion in the next commit.





**EXPERIMMENT NO. 4:**

AIM: Generate logs on GITHub

**Imagine a digital notebook** keeping track of every change made to your project. That's essentially what **git log** offers. This command acts like a time machine, revealing the **complete history** of your codebase.

**Why is this helpful?**

It's like having a **detailed record** of every edit, bug fix, and new feature added. This allows you to:

* **Go back in time** and see how your project evolved over time.
* **Understand who** made specific changes and **when**.
* **Revert to an earlier version** if needed, in case something went wrong.
* **Collaborate effectively** with others by tracking everyone's contributions.

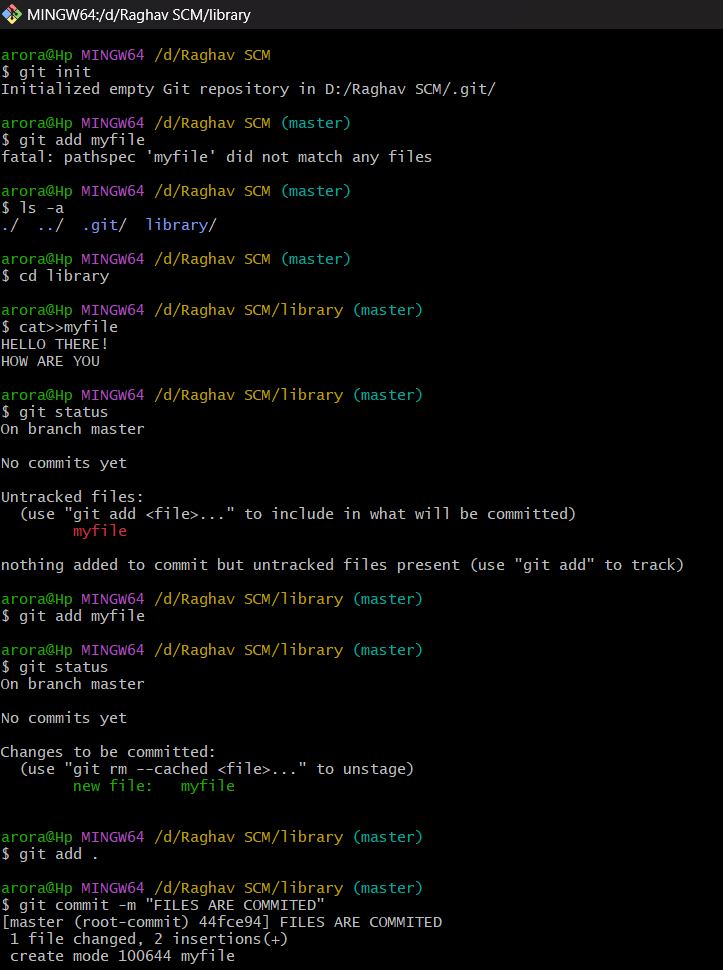
**Think of it as a story of your project's journey, told through commits.** With git log, you can always keep track of where you've been and where you're headed.

* **THE EXACT DEFINITION:**
* **Git Logs:**

The git log command shows a list of all the commits made to a repository. You can see the hash of each Git commit, the message associated with each commit, and more metadata. This command is basically used for displaying the history of a repository.

* **Why do we need logs?**

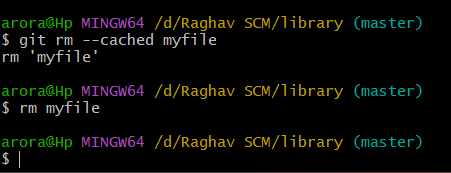
Git log is a utility tool to review and read a history of everything that happens to a repository. Anything we change at what time, by which log, everything is getting recorded in git logs.

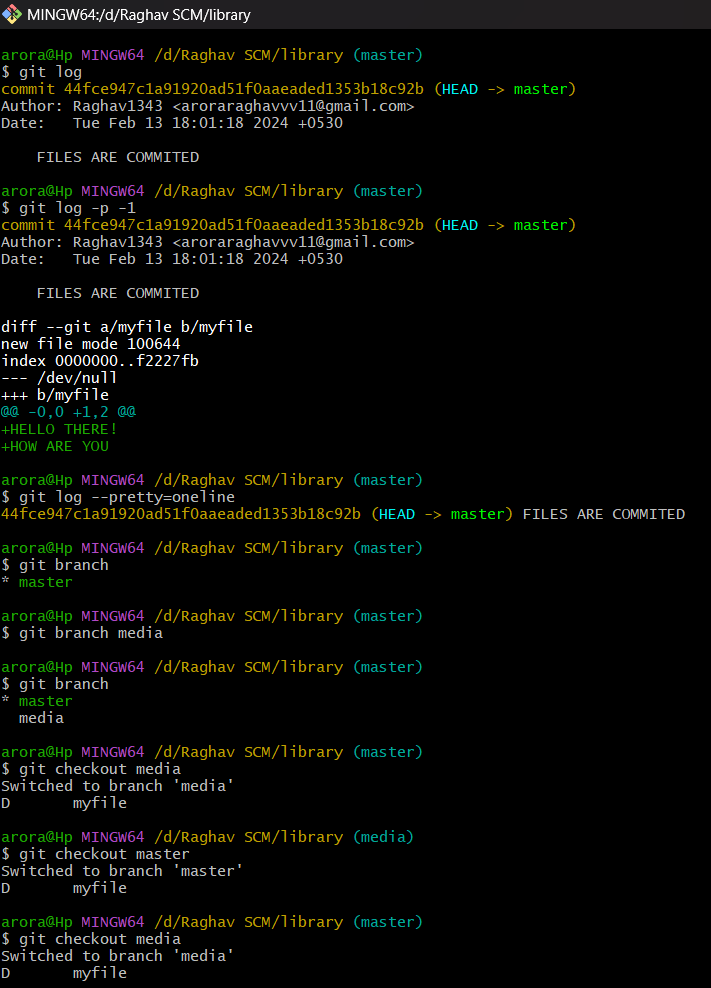


**Unveil your project's history with** git log**!**

This powerful command acts like a **time machine**, granting you access to a record of every change made, complete with timestamps.

* git log -p -1**:** Dive deeper! This command not only shows the latest commit message and details, but also displays the specific changes made to files within that commit, giving you a clear picture of what was modified.
* git log --pretty=oneline**:** Keep it concise! This command provides a brief overview of all commits, presenting each one-line summaries along with unique identifiers (not exactly encrypted codes, but rather unique codes used for tracking purposes).





**EXPERIMENT NO. 05:**

**AIM:** Git branching and Merging: Visualization of git Branch and HEAD, Git branches management, create a new branch, Commit changes in the new branch, explore commit in the new branch, Merging the branches

1.Creating and Visualizing the Branches On Git Client:

**Branching:**

* Imagine creating **copies of your project** to work on different features or bug fixes **independently**. This is what branching allows you to do.
* Think of it like creating **different paths on a map**, each representing a specific line of development.

**Merging:**

* Once you're happy with the changes in a branch, you can **"merge" them back into the main project**, integrating your work with the main codebase.
* This is like **combining different paths on the map** into a single, updated route.

**Why are they useful?**

* Branching allows you to **experiment and make changes safely** without affecting the main project.
* Merging helps you **incorporate the progress from different branches** into a central location.

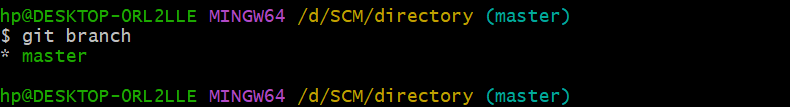
**Benefits:**

* **Improved collaboration:** Teams can work on different features simultaneously without interfering with each other.
* **Reduced risk:** Branching allows experimentation without affecting the main project.
* **Easier code management:** Complex projects can be broken down into smaller, more manageable tasks.

**How to create branches?**

The main purpose of branching is to isolate the workspace from branch-master on different branch. The main branch in which we are working is master branch. you can use the “git branch” command with the branch name and the commit SHA for the new branch.

* git branch: command shows the current branch on which we are working.



* git branch <branch name>: Command to create new branch.
* git checkout <branch name>: Command to switch to the entered branch name from previously working branch.

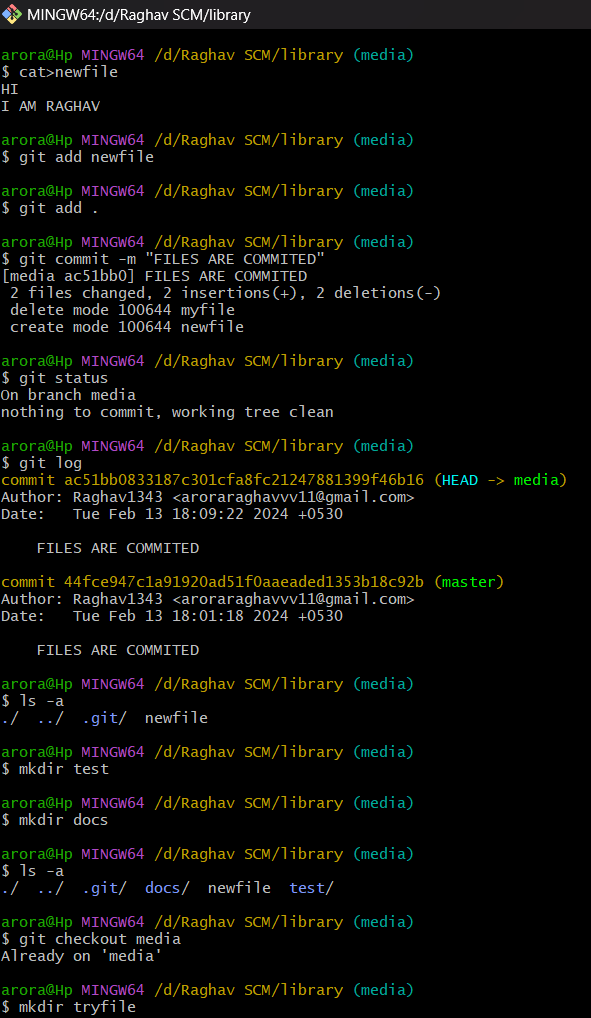


**Visualizing branches:**

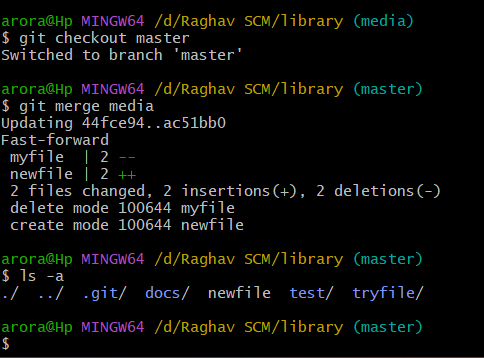
**Imagine your project as a garden:**

* **Creating a new branch:**
  + Think of it as planting a **seed** for a new section in the garden (a feature or bug fix). You create a new branch using git branch <branch\_name>.
* **Checking status:**
  + It's like **inspecting the soil** in your new section. You use git status to see if there are any unstaged or untracked changes.
* **Creating a new file:**
  + This is like **planting a flower** in your new section. You create a new file in the branch's directory using your preferred text editor.
* **Adding and committing:**
  + **Staging the flower** involves preparing it for planting by using git add <filename>. Then, **committing the flower** means burying it securely in the soil with git commit -m "message" (message describes the change).
* **Checking commit logs:**
  + This is like looking at a **gardening journal** to see the history of your actions. Use git log to show a list of all commits made in the current branch (your new section), including those in the main branch (the original garden) as a reference point.

This way, visualizing branches helps you understand how they isolate your work while keeping track of the project's overall history. It's like having multiple plots in your garden, each with its own changes, but all contributing to the entire landscape.



* git merge<branch name>: To merge the repository of another branch into current working branch (branch master).



**EXPERIMENT NO. 6:**

**AIM:** Git lifecycle description Git status, add, commit, stage – Life cycle of a file in Git managed in Repository.

## The Stages of a Git Workflow: A Deeper Dive

Imagine your project as a living, evolving document. Git helps you manage this evolution by providing a clear workflow with distinct stages:

**1. Working Directory:**

This is your project's **home base**, where you actively **create, edit, and delete files**. Think of it as your workbench where you modify and experiment with your project. Changes made here are not yet tracked by Git.

**2. Staging Area:**

Before permanently "saving" changes, you can **select specific modifications** to be included in the next commit. You do this by "staging" them. Think of it like **gathering the tools and materials** you want to keep from your workbench for future reference. Only staged changes are considered for the next "snapshot" of your project.

**3. Local Repository:**

When you're ready to capture the current state of your project, you **commit** the staged changes. This creates **a permanent record** in your **local Git repository**, which resides on your own computer. Each commit is like a **snapshot** of your project at a specific point in time, with a unique identifier (hash) and a descriptive message explaining the changes.

**4. Remote Repository (Optional):**

If you're collaborating with others or want a backup, you can **push** your committed changes from your local repository to a **remote repository.** This is typically hosted online on platforms like GitHub. Think of it like **uploading your project files and history** to a secure online storage location accessible to your team.

**5. Branches (Optional):**

Git empowers you to work on **multiple lines of development** simultaneously through **branches.** Each branch acts as a separate **working directory with its own history.** You can create branches using git branch <branch\_name>, switch between them with git checkout

<branch\_name>, and merge completed branches back into the main branch using git merge <branch\_name>. This allows you to explore new features, fix bugs, or experiment without impacting the main project's stability.

**6. Git Directory:**

Hidden within your project directory, the .git folder serves as the **central hub** for all Git-related information. It stores the **commit history**, including the unique identifiers, commit messages, and details of all changes made. It also holds data about branches, remote repositories, and configurations. This folder acts as the **backbone** for Git's operations and allows you to track the entire project's evolution.

**In summary, the Git lifecycle involves a well-defined workflow that allows you to manage your project's evolution effectively. Understanding each stage empowers you to collaborate seamlessly, experiment safely, and maintain a clear history of your project's development.**