Course Code **BCS358C** CIE Marks 50

Teaching Hours/Week (L:T:P: S) 0: 0 : 2: 0 SEE Marks 50

Credits 01 Exam Marks 100

Examination type (SEE) Practical

**Course objectives:**

● .To familiar with basic command of Git

● To create and manage branches

● To understand how to collaborate and work with Remote Repositories

● To familiar with version-controlling commands

**Sl.NO Experiments**

1 **Setting Up and Basic Commands**

Initialize a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.

2 **Creating and Managing Branches**

Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."

3 **Creating and Managing Branches**

Write the commands to stash your changes, switch branches, and then apply the stashed

changes.

4 **Collaboration and Remote Repositories**

Clone a remote Git repository to your local machine.

5 **Collaboration and Remote Repositories**

Fetch the latest changes from a remote repository and rebase your local branch onto the

updated remote branch.

6 **Collaboration and Remote Repositories**

Write the command to merge "feature-branch" into "master" while providing a custom

commit message for the merge.

7 **Git Tags and Releases**

Write the command to create a lightweight Git tag named "v1.0" for a commit in your local

repository.

8 **Advanced Git Operations**

Write the command to cherry-pick a range of commits from "source-branch" to the current

branch.

9 **Analysing and Changing Git History**

Given a commit ID, how would you use Git to view the details of that specific commit,

including the author, date, and commit message?

10 **Analysing and Changing Git History**

Write the command to list all commits made by the author "JohnDoe" between "2023-01-01"

and "2023-12-31."

11 **Analysing and Changing Git History**

Write the command to display the last five commits in the repository's history.

12 **Analysing and Changing Git History**

Write the command to undo the changes introduced by the commit with the ID "abc123".

**Course outcomes (Course Skill Set):**

At the end of the course the student will be able to:

● Use the basics commands related to git repository

● Create and manage the branches

● Apply commands related to Collaboration and Remote Repositories

● Use the commands related to Git Tags, Releases and advanced git operations

● Analyse and change the git history

**Overview of GITHUB Account:**

GitHub is a web-based platform that provides a hosting service for software development projects. It utilizes Git, a distributed version control system, to help developers collaborate on projects and track changes made to the source code. Here are some key aspects of GitHub:

* **Version Control:** Git is a version control system that allows developers to track changes in their codebase, collaborate with others, and manage different versions of their projects. GitHub provides a platform for hosting Git repositories.
* **Repository (Repo):** A repository is a container for a project, containing all the files, documentation, and version history. Each project on GitHub is stored in its own repository.
* **Collaboration:** GitHub facilitates collaboration among developers by providing tools for branching, merging, and resolving conflicts in the code. Multiple developers can work on different branches of a project and then merge their changes back into the main codebase.
* **Issues:** GitHub has an issue tracking system that allows users to report bugs, request new features, or discuss ideas. Issues can be assigned, labeled, and commented on, making it easier for teams to manage and prioritize their work.
* **Pull Requests:** When a developer wants to contribute changes to a project, they create a pull request (PR). A PR is a proposal to merge changes from one branch into another. It allows for code review and discussion before the changes are merged.
* **GitHub Actions:** GitHub Actions is a feature that enables automation of workflows, such as running tests, deploying applications, or performing other tasks, directly from the GitHub repository.
* **Gists:** Gists are a way to share snippets or small pieces of code with others. They are like mini-repositories and can be used for sharing code snippets, configuration files, or any other text content.
* **Wikis:** Repositories on GitHub can have associated wikis for documentation. This allows developers to create and maintain project documentation directly on the platform.

GitHub is widely used in the software development community, serving as a central hub for open source projects, collaborative coding, and community-driven development. Many organizations and individual developers use GitHub to host, manage, and collaborate on their projects.



**1. Code:**

Menu Items:

* Files: Displays the files in the repository.
* Commits: Shows a list of commits made to the repository.
* Branches: Lists all branches in the repository.
* Pull Requests: View and create pull requests.
* Compare: Compare different branches or commits.

Sample Use Cases:

* Viewing Files: Navigate through the project's codebase to understand its structure.
* Reviewing Commits: Check the history of changes made to the codebase.
* Creating Pull Requests: Propose changes to the code and initiate code reviews.

**2. Issues:**

Menu Items:

* Overview: Displays a summary of open and closed issues.
* Assigned: Shows issues assigned to you.
* Mentioned: Lists issues where you are mentioned.
* Filters: Custom filters for sorting and organizing issues.

Sample Use Cases:

* Reporting Issues: Open new issues to report bugs or suggest enhancements.
* Assigning Issues: Assign issues to team members for resolution.
* Tracking Mentions: Keep track of discussions where you are mentioned.

**3. Pull Requests:**

Menu Items:

* Open: Displays open pull requests.
* Closed: Lists closed pull requests.
* Drafts: Shows pull requests in draft status.
* Merged: Lists merged pull requests.

Sample Use Cases:

* Creating Pull Requests: Propose changes to merge into the main branch.
* Reviewing Code: Collaborate on code changes before merging.
* Merging Changes: Incorporate approved changes into the main branch.

**4. Actions:**

Menu Items:

* Workflows: View and manage GitHub Actions workflows.
* Workflow runs: Monitor the runs of GitHub Actions workflows.
* Artifacts: Access artifacts produced by workflows.
* Settings: Configure GitHub Actions settings.

Sample Use Cases:

* Continuous Integration (CI): Automate testing and build processes.
* Continuous Deployment (CD): Automate deployment workflows.
* Artifact Storage: Store and access build artifacts.

**5. Projects:**

Menu Items:

* Your projects: Lists projects you've created or are involved in.
* Explore: Discover and explore projects.
* New project: Create a new project board for task management.

Sample Use Cases:

* Task Management: Organize and manage tasks on a project board.
* Kanban Boards: Visualize and track the progress of work items.
* Collaboration: Coordinate work across team members.

**6. Wiki:**

Menu Items:

* Pages: View and edit wiki pages associated with the repository.
* History: See the revision history of wiki pages.

Sample Use Cases:

* Documentation: Create and maintain project documentation.
* Knowledge Sharing: Share information about project architecture or processes.

**7. Security:**

Menu Items:

* Code scanning: Perform and review code scanning results.
* Dependency graph: Explore and manage dependencies in your repository.
* Secrets: Manage secrets used in GitHub Actions workflows.
* Advanced security: Access advanced security features.

Sample Use Cases:

* Code Scanning: Identify and remediate security vulnerabilities in the code.
* Dependency Management: Monitor and update dependencies for security.
* Secrets Management: Securely store and manage sensitive information.

**8. Insights:**

Menu Items:

* Traffic: View repository traffic and page views.
* Commits: See statistics about commits.
* Code frequency: Analyze code changes over time.
* Contributors: List contributors to the repository.

Sample Use Cases:

* Analytics: Understand repository usage and activity.
* Contributor Recognition: Acknowledge and appreciate contributors.

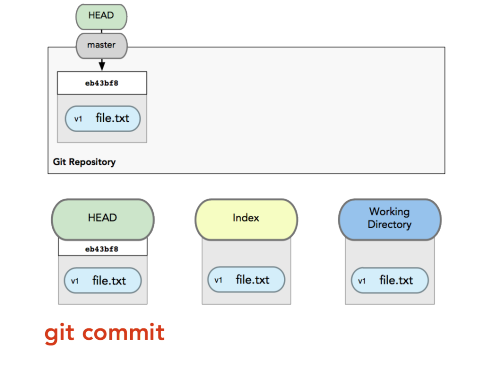
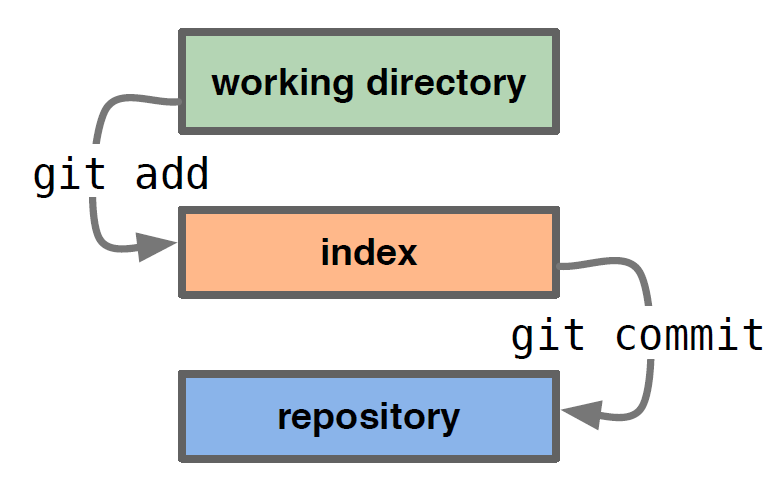
9. Settings:

Menu Items:

* Options to configure settings: Configure repository settings, collaborators, and other options.

Sample Use Cases:

* Repository Configuration: Set repository-specific options.
* Collaborator Management: Manage access and permissions for collaborators.
* Branch Protection: Configure rules to protect specific branches.



**LABSET 1 : SIGNUP GITHUB ACCOUNT**

**Step 1: Visit GitHub Website**

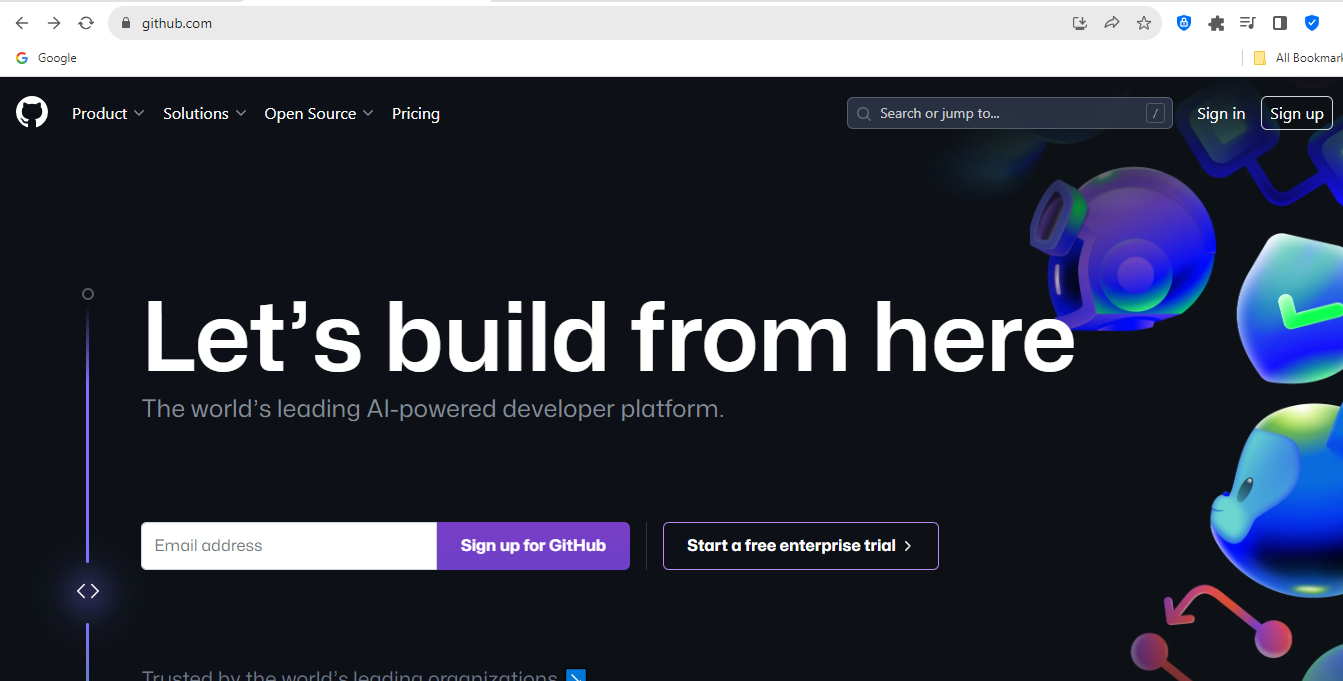
Go to the GitHub website: <https://github.com/>

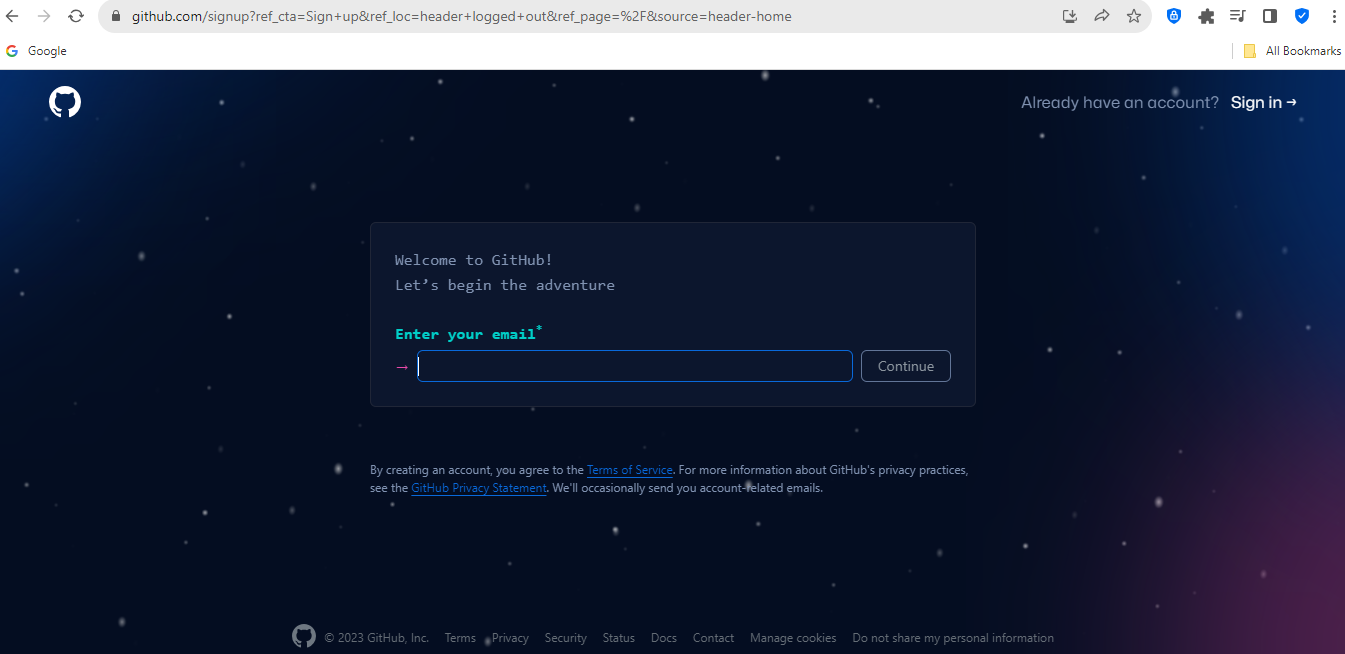
(HOW TO CREATE GITHUB ACCOUNT)

<https://www.youtube.com/watch?v=Gn3w1UvTx0A>

**Step 2: Sign Up**

Click on the "Sign up" button in the upper-right corner of the GitHub homepage.





**Step 3: Enter Account Information**

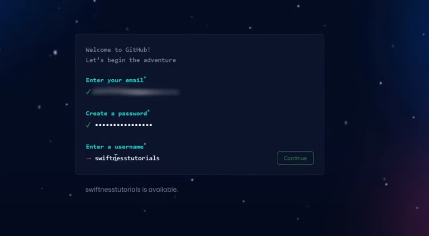
Fill out the required information in the sign-up form:

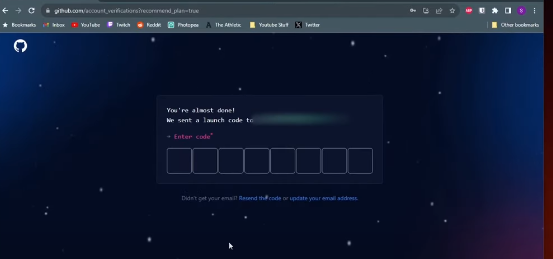
Username: Choose a unique username for your GitHub account.

Email address: Provide a valid email address.

Password: Create a strong password.

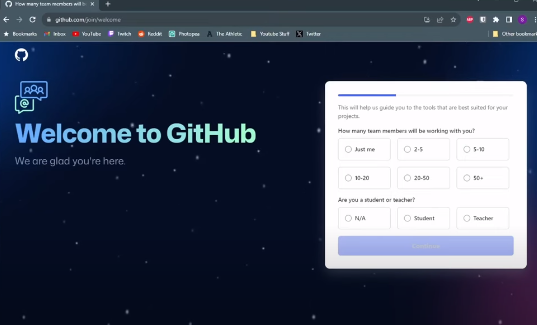
Click on the "Verify" button to prove that you're not a robot by solving a simple puzzle.





**Step 4: Choose Plan**

GitHub offers both free and paid plans. For most users, the free plan is sufficient. Choose the free plan by clicking on the "Continue" button.



**Step 5: Tailor Your Experience (Optional)**

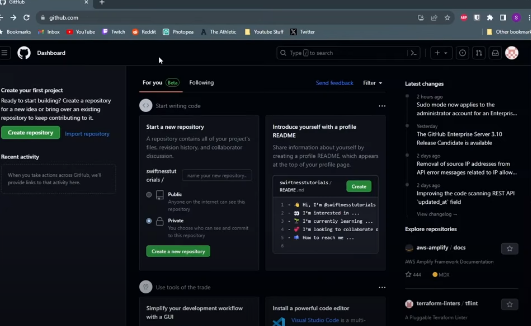
GitHub will ask you about your experience level and the type of work you'll be doing. Answer the questions based on your preferences. You can also skip this step if you're not sure.

**Step 6: Complete Setup**

GitHub might ask you to verify your email address. Check your email inbox for a verification email from GitHub and follow the instructions to verify your email.

**Step 7: Explore GitHub (Optional)**

Once your account is set up, you can explore GitHub. You can follow users, star repositories, and get familiar with the platform.



**Step 8: Create a New Repository (Optional)**

If you have code or files you want to share or manage with Git, you can create a new repository on GitHub:

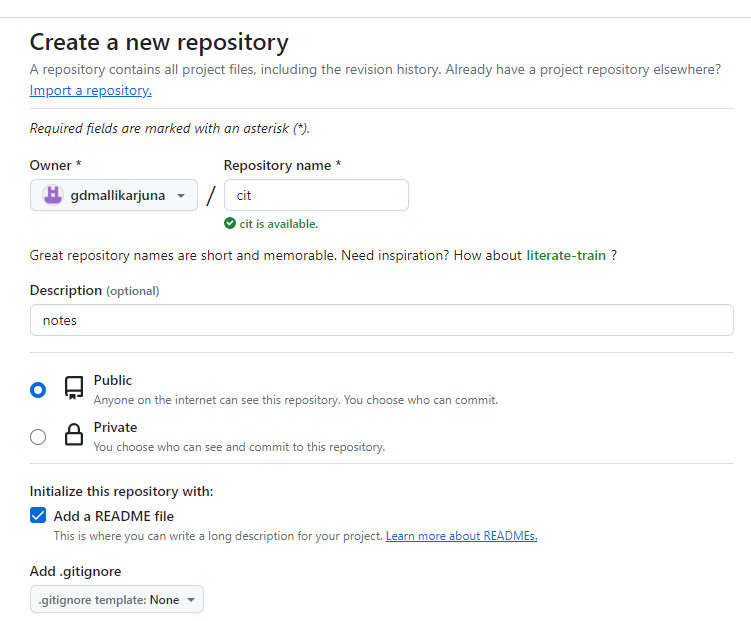
Click the "+" sign in the upper-right corner of the GitHub page.

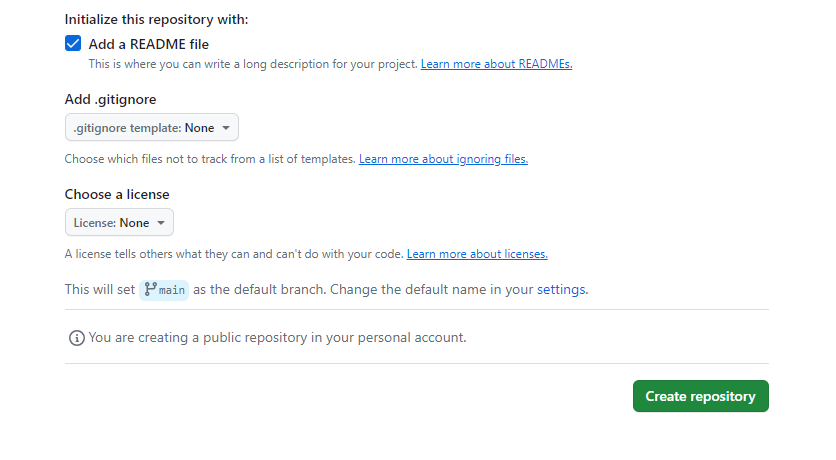
Choose "New repository."

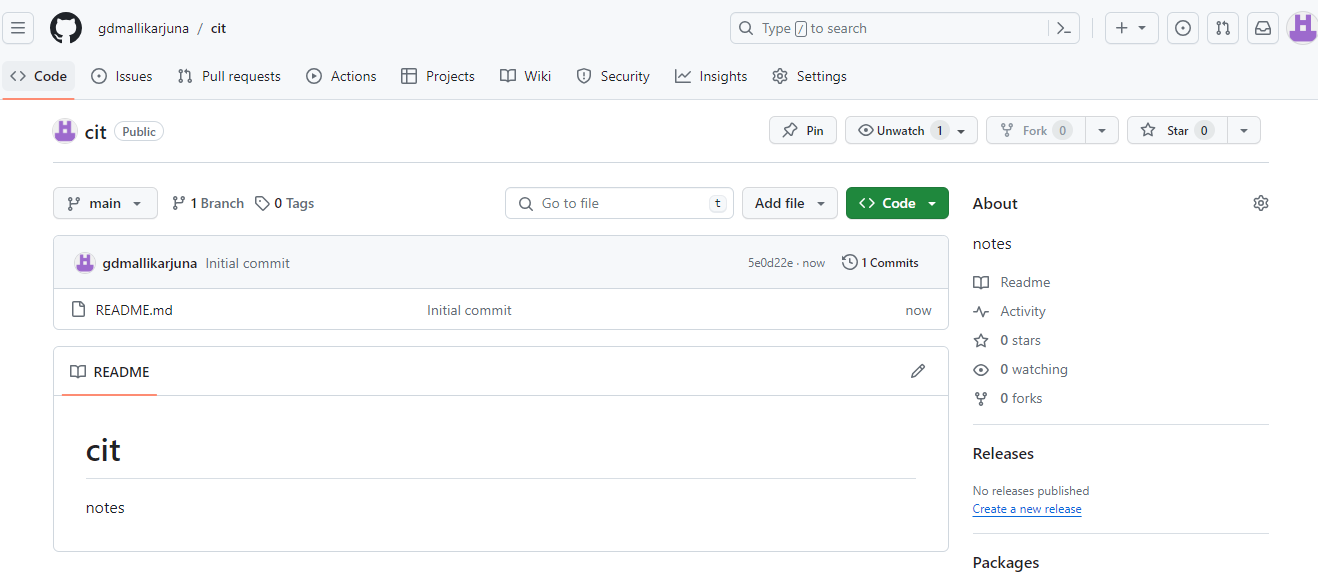
Fill out the repository name, description, and other settings.

Optionally, initialize the repository with a README file.

Click "Create repository."







CIT Repository created

**Step 9: Set Up Git Remote (Optional)**

If you've created a new repository on GitHub and you have existing code locally, you'll want to set up the connection between your local repository and the one on GitHub. Follow the instructions provided by GitHub after creating the repository.

Now you have successfully set up a GitHub account and can start using it to collaborate on projects, contribute to open source, or host your code repositories.

**"main," "master," and "origin" Branches :**

the terms "main," "master," and "origin" are related to the Git version control system and are used to refer to different concepts:

**Main Branch:**

* The "main" branch in GitHub is the default branch for new repositories.
* It represents the primary line of development and is often used to hold the latest stable version of the project.
* The use of "main" as the default branch name is part of an industry-wide effort to move away from the term "master" due to its historical association with slavery.

**Master Branch:**

* Historically, "master" was the default name for the primary branch in Git and GitHub.
* Some projects and organizations continue to use "master" as the default branch name.
* However, there has been a push to replace "master" with more neutral and inclusive alternatives like "main" to promote diversity and inclusivity in the tech community.

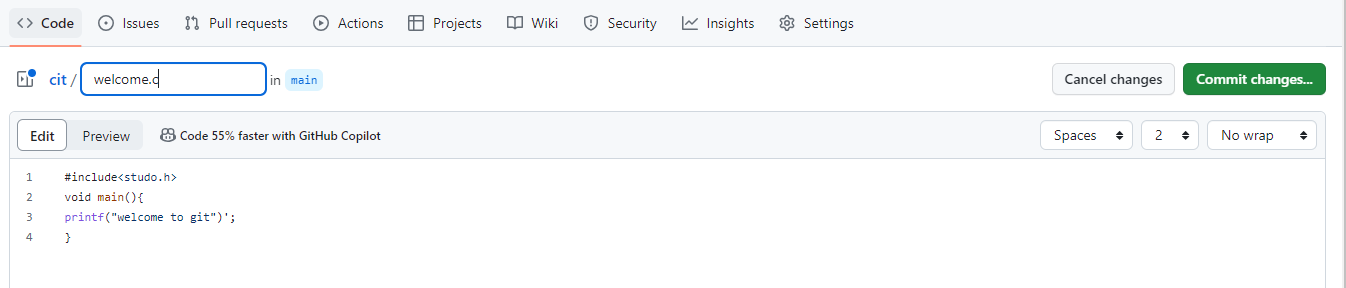
**Origin:**

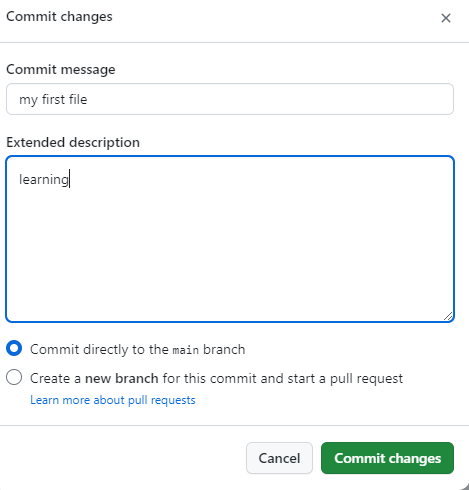
* "Origin" is a default remote name in Git, and it typically refers to the remote repository from which a local repository was cloned.
* When you clone a repository using Git, the remote repository is assigned the name "origin" by default.
* You might see references to "origin" when pushing changes to or pulling changes from the remote repository.

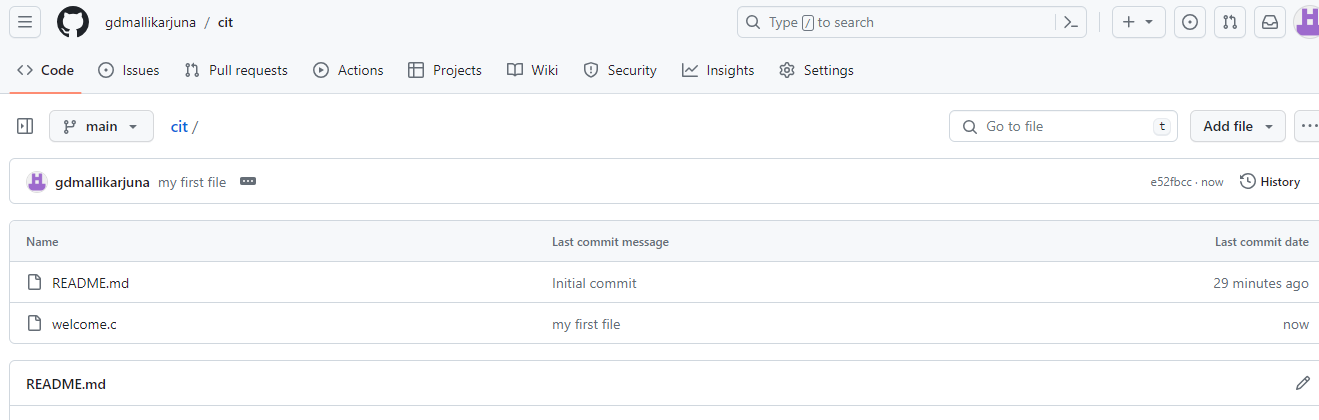
"Main" and "Master" Branches: These refer to the primary development branch in a Git repository. "Main" is often used as a more inclusive alternative to "master," and GitHub defaults to using "main" for new repositories.

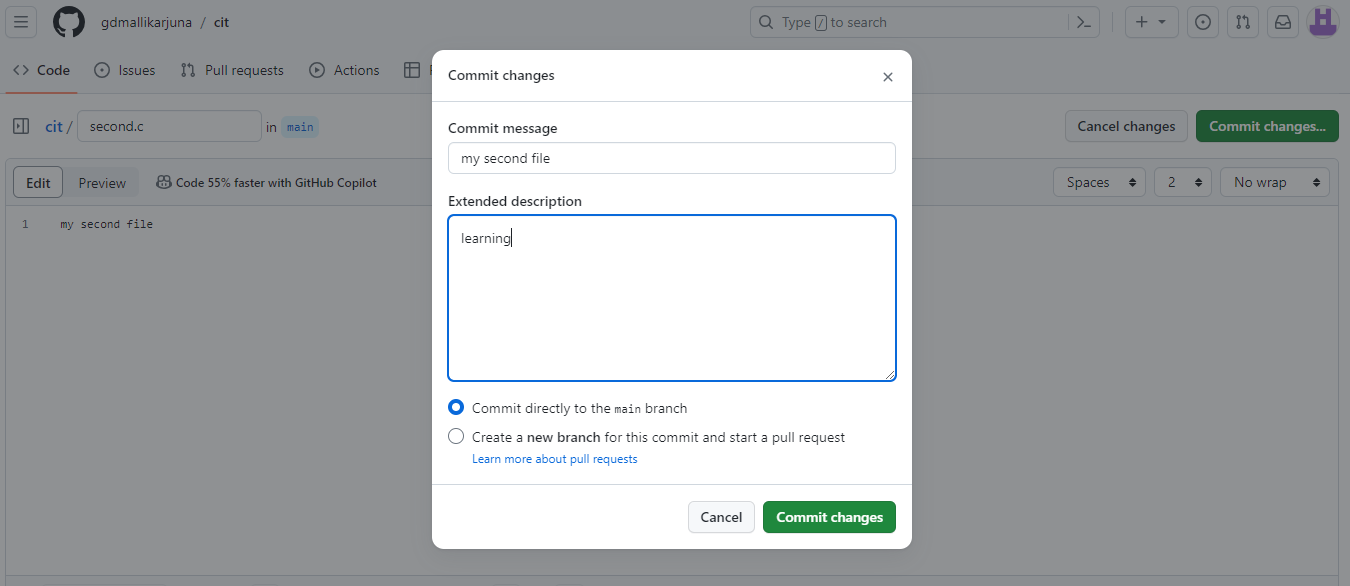
"Origin": This is the default remote name that points to the repository from which you cloned your local repository. It is used when interacting with the remote repository, such as fetching or pushing changes.

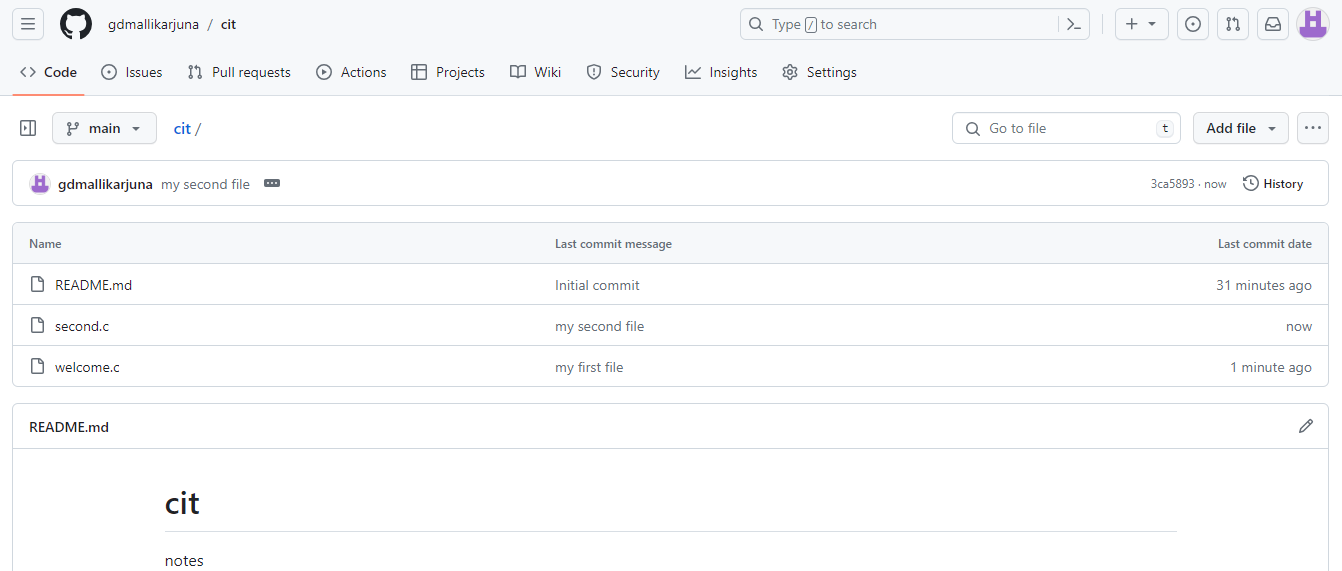
**Step 9: create a sample file**





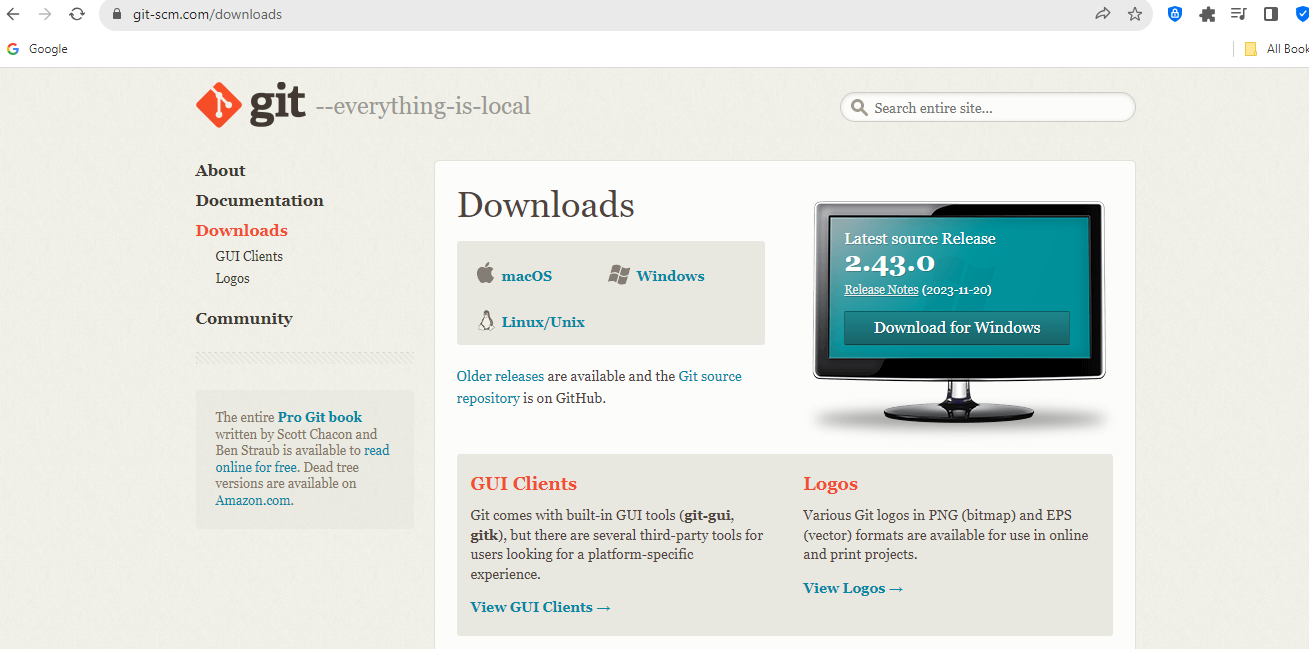






**LABSET 2: GIT INSTALLATION AND SETUP**

[**https://www.youtube.com/watch?v=JgOs70Y7jew**](https://www.youtube.com/watch?v=JgOs70Y7jew)



**Step 1: Download and Install Git**

**Linux:**

Most Linux distributions come with Git pre-installed. If it's not installed, you can use your package manager to install it.

For Debian/Ubuntu:

sudo apt-get update

sudo apt-get install git

For Fedora:

sudo dnf install git

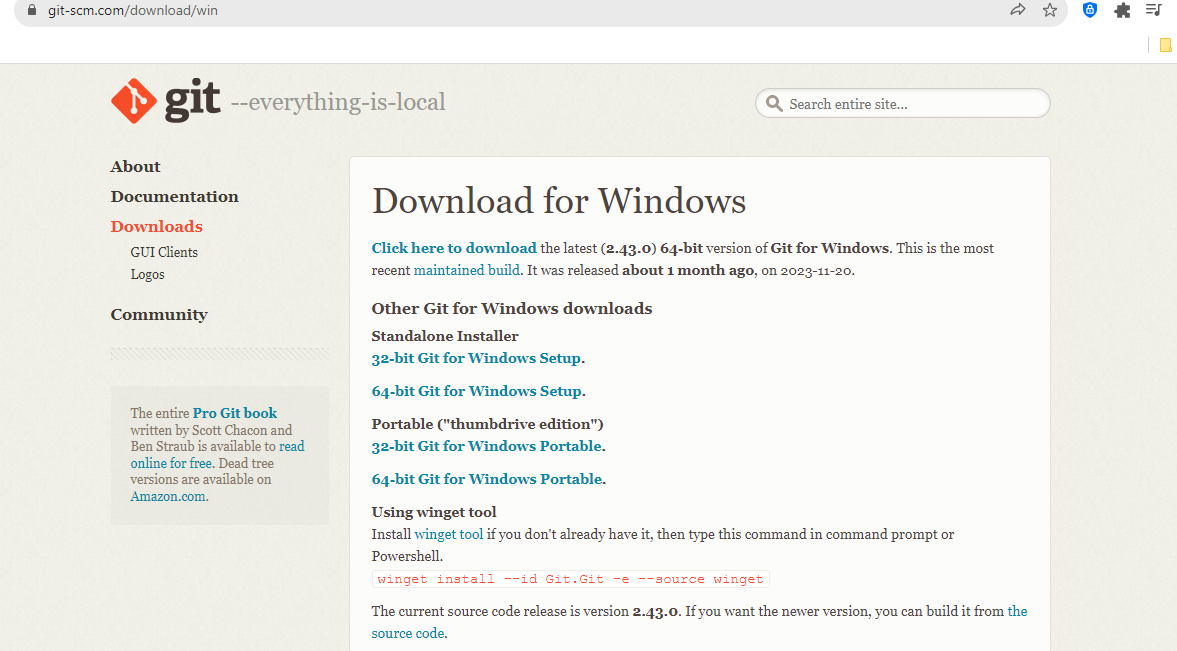
**Mac:**

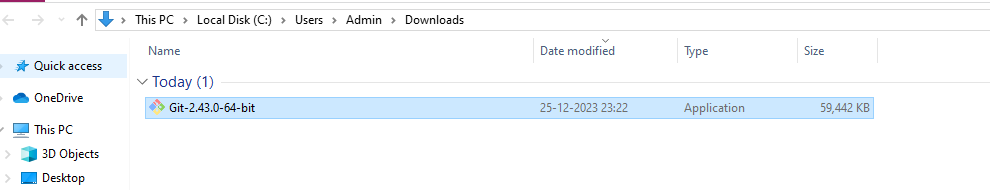
You can install Git on macOS using Homebrew. If you don't have Homebrew installed, you can install it by following the instructions on the Homebrew website.

brew install git

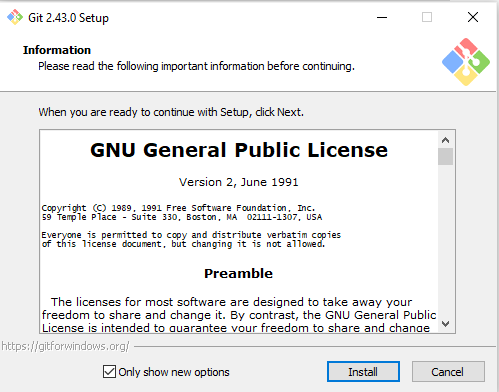
**Windows:**

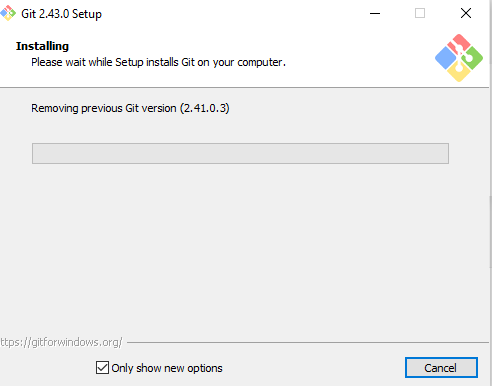
Download the Git installer from the official website: Git for Windows. Run the installer and follow the on-screen instructions.

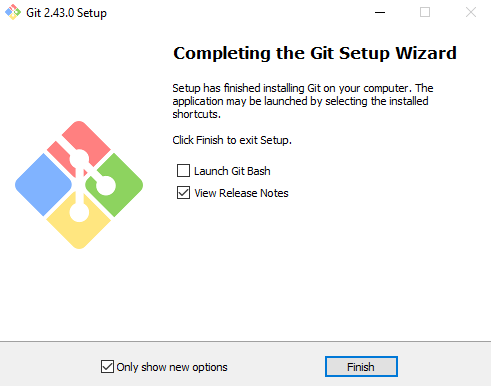


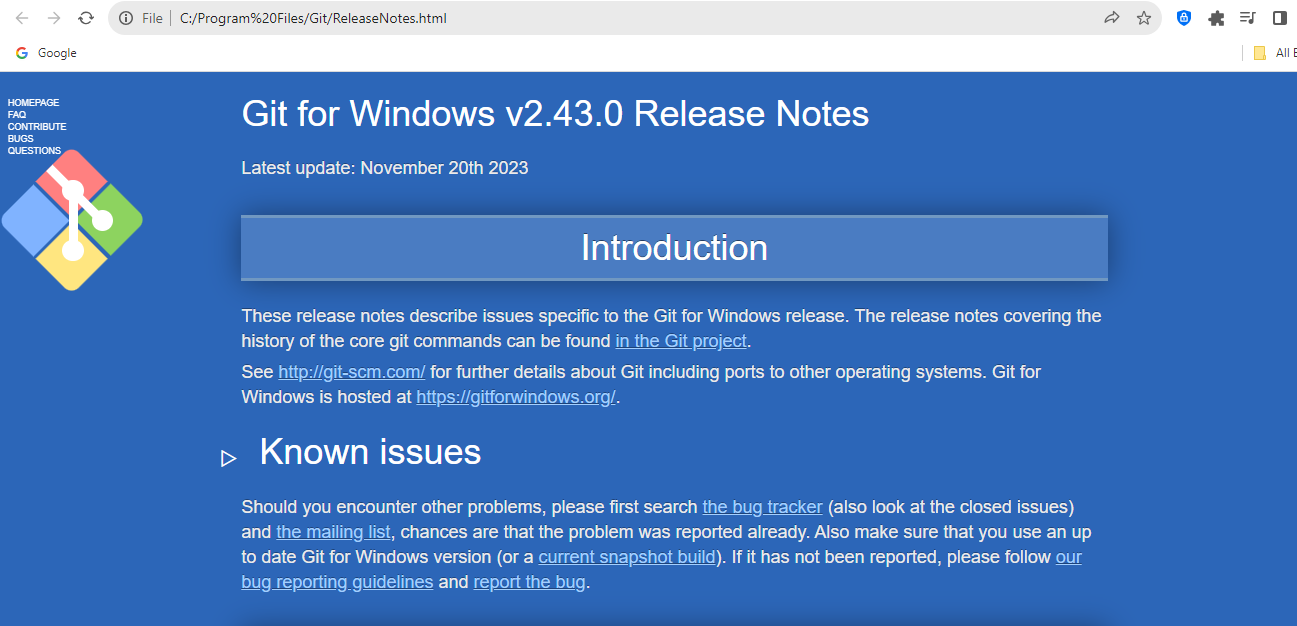


Copy to the appropriate folder start install with default configuration as windows installer

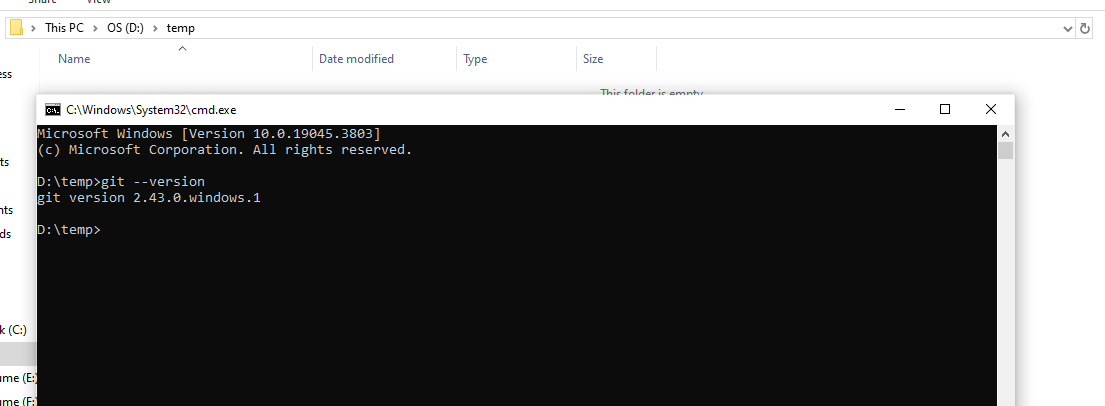








Check the installation



**Step 2: Configure Git**

After installing Git, you need to configure it with your name and email. Open a terminal or command prompt and run the following commands:

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

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

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

D:\temp>git config --global user.name "Mallikarjuna G D"

D:\temp>git config --global user.email "gdmallikarjuna@gmail.com"

**Step 3: Verify Configuration**

To verify that your configuration was successful, you can use the following command:

git config --global --list

This should display a list of your Git configuration settings.

D:\temp>git config --global --list

user.username=gdmallikarjuna

user.email=gdmallikarjuna@gmail.com

user.name=Mallikarjuna G D

usesr.name=gdmallikarjuna

**Step 4: Create a New Git Repository (Optional)**

If you're starting a new project, navigate to your project's root directory and initialize a new Git repository:

cd /path/to/your/project

git init

**Step 6: Start Using Git**

Now that Git is set up, you can start using it for version control. Here are some basic commands to get you started:

git add <filename>: Add a file to the staging area.

git commit -m "Your commit message": Commit changes to the repository.

git push origin <branch>: Push changes to a remote repository.

git pull origin <branch>: Pull changes from a remote repository.

Remember to replace <filename> and <branch> with your actual file or branch names.

That's it! You've successfully set up Git on your system.

D:\temp>git add test1.txt.txt

D:\temp>git commit -m "first code"

[master (root-commit) 4917ca2] first code

1 file changed, 1 insertion(+)

create mode 100644 test1.txt.txt

D:\temp>git push origin main

error: src refspec main does not match any

error: failed to push some refs to 'origin'

D:\temp>git clone https://github.com/gdmallikarjuna/cit.git

Cloning into 'cit'...

remote: Enumerating objects: 9, done.

remote: Counting objects: 100% (9/9), done.

remote: Compressing objects: 100% (6/6), done.

remote: Total 9 (delta 0), reused 0 (delta 0), pack-reused 0

Receiving objects: 100% (9/9), done.

D:\temp\cit>git add .

D:\temp\cit>git status

On branch main

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

Changes to be committed:

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

new file: welcome123.txt

D:\temp\cit>git commit -m "welcome to learn"

[main 2a8d934] welcome to learn

1 file changed, 1 insertion(+)

create mode 100644 welcome123.txt

D:\temp\cit>git push origin main

Enumerating objects: 4, done.

Counting objects: 100% (4/4), done.

Delta compression using up to 4 threads

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 291 bytes | 291.00 KiB/s, done.

Total 3 (delta 1), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (1/1), completed with 1 local object.

To https://github.com/gdmallikarjuna/cit.git

3ca5893..2a8d934 main -> main

**LABSET 3: Setting Up and Basic Commands**

**Initialize a new Git repository in a directory. Create a new file and add it to the staging area**

**and commit the changes with an appropriate commit message.**

**Step 1: Initialize a New Git Repository**

Open a terminal or command prompt and navigate to the directory where you want to create the new Git repository:

cd /path/to/your/directory

Initialize a new Git repository:

git init

**Step 2: Create a New File**

Create a new file in the repository. For example, let's create a file called "example.txt". You can use any text editor to create the file:

touch example.txt

**Step 3: Add the File to the Staging Area**

Add the newly created file to the staging area:

git add example.txt

**Step 4: Commit the Changes**

Commit the changes with an appropriate commit message. Replace "Your commit message" with a meaningful message describing the changes you made:

git commit -m "Initial commit - added example.txt"

**Recap:**

Here's a summary of the commands:

# Navigate to your directory

cd /path/to/your/directory

# Initialize a new Git repository

git init

# Create a new file (e.g., example.txt)

touch example.txt

# Add the file to the staging area

git add example.txt

# Commit the changes with a commit message

git commit -m "Initial commit - added example.txt"

Now you've initialized a new Git repository, created a new file, added it to the staging area, and committed the changes with an appropriate commit message

Demonstrate some more commands

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **GIT COMMANDS** | | | | |
| sr no | commands | Description | syntax | example |
| 1 | git init | starts the new git repository | git init [repository name] | git int learn\_git |
| 2 | git config | the command sets the author name and email address to commit | git config --global user.name "[name]"  git config --global user.email "[email address]" | git config --global user.name "gdmallikarjuna" git config --global user.email "gdmallikarjuna@gmail.com" |
| 3 | git clone | This command is used to obtain a repository from an existing URL. | git clone [url] | git clone https://github.com/gdmallikarjuna/training.git |
| 4 | git add | adds the file to staging area | git add [file] | git add welcome123.txt |
|  |  | adds more than one file | git add \* | git add \* |
| 5 | git commit | This command records or snapshots the file permanently in the version history. | git commit -m "[ message]" | git commit -m "Welcome message" |
|  |  | The git commit -a command is a shortcut in Git that allows you to stage and commit changes in one step. | git commit -a | git commit -a |
|  |  |  |  |  |
| 6 | git diff | The git diff command in Git is used to show changes between different commits, between the working directory and the staging area, or between the staging area and a specific commit. It provides a way to inspect and understand the differences in the codebase.. | **View Changes in the Working Directory:**  git diff | git diff |
| ``` |  | shows the differences between the files in the staging area and the latest version present | git diff --staged  git diff commit1 commit2  git diff file1 file2 | git diff –staged |
|  |  | compare branches | git diff [branch-a] [branch-b] | git diff master dev |
| 7 | git reset | This command unstages the file, but it preserves the file contents. | git reset [file] | git reset challenge.txt |
|  |  | This command undoes all the commits after the specified commit and preserves the changes locally | git reset [commit] | git reset <hashode> |
|  |  | command discards all history and goes back to the specified | git reset –hard [commit] |  |
| 8 | git status | This command lists all the files that have to be committed. | git status | git status |
| 9 | git rm | this command deletes the file from working directory and stages deletion | git rm [file] | git rm challenges.txt |
| 10 | git log | it list the version history of the current branch | git log | git log |
| 11 | git show | shows the metadata and content changes of the specified commit | git show [commit] | git show b7254b5a437ee4eae59c2ea5478ea0ad12107ec2 |
| 12 | git tag | This command is used to give tags to the specified commit | git tag [commitID] | git tag b7254b5a437ee4eae59c2ea5478ea0ad12107ec2 |
| 13 | git branch | lists all the branches | git branch | git branch |
|  |  | create new branch | git branch [branch name] | git branch dev |
|  |  | delete a branch | git branch -d [branch name] | git branch -d dev |
| 14 | git checkout | switching from one branch to another | git checkout [branchname] | git checkout dev |
|  |  | create a new branch and also switch to it | git checkout -b [new branch name] | git checkout -b test |
| 15 | git merge | merged to branch history to the current directory | git merge [branch name] | git merge dev |
| 16 | git remote | This command is used to connect your local repository to the remote server | git remote add [variable\_name] [Remote server link] | git remote add mallik https://github.com/gdmallikarjuna/training.git |
| 17 | git push | the committed changes of master branch to your remote repository. | git push [variable name] master | git push origin master |
|  |  | it pushes all branches | git push --all [variable name] | git push --all |
| 18 | git pull | it pulls the changes from remote link | git pull [Repository Link] | git pull https://github.com/gdmallikarjuna/training.git |
| 19 | git stash | git stash temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on | git stash save | git stash save |
|  |  | restores most recently files | git stash pop | git stash pop |
|  |  | lists all stashed changesets | git stash list | git stash list |
|  |  | discard the most recently stashed | git stash drop | git stash drop |
|  |  |  |  |  |

**GIT TAG**: In Git, a tag is a reference to a specific commit in the version history of a repository. Tags are used to mark specific points in history, often to signify releases, versions, or other important milestones. Unlike branches, tags are generally not meant to move; they are a way to "bookmark" a specific commit for easy reference.

Here's a detailed step-by-step explanation of how to create and use Git tags:

**Step 1: Navigate to Your Git Repository**

Open a terminal or command prompt and navigate to the directory of your Git repository:

cd /path/to/your/repo

**Step 2: List Existing Tags (Optional)**

To see if there are any existing tags in your repository, you can use:

git tag

This command lists all the tags in the repository.

**Step 3: Create a Lightweight Tag**

To create a lightweight tag (a simple pointer to a specific commit), use the following command:

git tag v1.0.0

Replace "v1.0.0" with the version or name you want to give to your tag. This tags the current commit with the specified name.

**Step 4: Create an Annotated Tag**

To create an annotated tag (includes a message and additional information), use the following command:

git tag -a v1.1.0 -m "Release version 1.1.0"

This creates an annotated tag named "v1.1.0" with the specified message.

**Step 5: View Tag Information**

To view information about a specific tag, use the following command:

git show v1.1.0

This command displays details about the tag, including the commit it points to and the tag message.

**Step 6: List Tags Again**

To verify that the new tags are created, list all tags again:

git tag

**Step 7: Push Tags to Remote**

To share your tags with others or push them to a remote repository, use the following commands:

git push origin v1.0.0

git push origin v1.1.0

Or, to push all tags:

git push origin --tags

**Step 8: Checkout a Specific Tag**

To check out a specific tag and create a detached HEAD state (not on a branch), use the following command:

git checkout v1.1.0

Now, you are in a detached HEAD state corresponding to the "v1.1.0" tag.

**Step 9: Create a Branch from a Tag (Optional)**

If you want to make changes based on a specific tag, you might want to create a branch from that tag:

git checkout -b my\_feature\_branch v1.1.0

This creates a new branch named "my\_feature\_branch" based on the commit pointed to by the "v1.1.0" tag.

**GIT MERGE:** In Git, the git merge command is used to integrate changes from one branch into another. Merging is a fundamental operation in Git that allows you to combine the work from different branches into a single branch, typically bringing changes from a source branch into a target branch.

Here's a basic overview of how git merge works:

Basic Merge:

Switch to the Target Branch:

Before merging, you need to be on the branch where you want to merge changes. For example, to merge changes from the feature-branch into main, you would switch to the main branch:

git checkout main

Or, using a more recent Git command:

git switch main

Run the Merge Command:

Now, you can run the git merge command, specifying the branch you want to merge

git merge feature-branch

This command combines the changes from the feature-branch into the current branch (main in this case).

Fast-Forward Merge:

If there are no conflicting changes between the two branches, Git performs a "fast-forward" merge. This means that the branch pointer of the target branch is simply moved forward to the latest commit of the source branch.

# Assuming you are on the target branch (e.g., main)

git merge feature-branch

Three-Way Merge:

If there are conflicting changes (i.e., changes in both branches that cannot be automatically reconciled), Git performs a "three-way merge." It creates a new commit that represents the combination of both branches.

# Assuming you are on the target branch (e.g., main)

git merge feature-branch

Recursive Merge Strategy:

Git uses a recursive merge strategy by default. This strategy is capable of handling complex merge scenarios and is the most common strategy used.

# Assuming you are on the target branch (e.g., main)

git merge feature-branch

Manual Resolution of Conflicts:

If conflicts occur during the merge, Git will pause and ask you to resolve the conflicts manually. You can do this by editing the conflicted files and then completing the merge with:

git merge --continue

Or, to abort the merge:

git merge --abort

Visualizing Merge History:

You can use tools like git log or Git visualization tools to view the commit history and understand how branches have been merged over time.

git log --graph --oneline --all

This command displays a compact graph of the commit history, showing branch merges and their relationships.

Merging is an essential part of collaborative development in Git, allowing multiple developers to work on different branches and later integrate their changes into a common branch.

**GIT STASH :** In Git, the git stash command is used to temporarily save changes that are not ready to be committed, allowing you to switch to a different branch or address an urgent issue without committing incomplete work. The stashed changes are stored in a stack, and you can later apply or drop them as needed.

Here's an explanation of how git stash works with examples:

**Example 1: Stashing Changes**

Suppose you are working on a branch, and you have some uncommitted changes that you want to set aside temporarily:

# Save your changes to the stash

git stash save "Work in progress"

# The working directory is now clean

In this example, the git stash save command saves your changes with a message ("Work in progress"). The working directory is then clean, and you can switch branches or perform other operations.

**Example 2: Listing Stashes**

You can list your stashes to see what you have stashed:

git stash list

This command will show you a list of stashes with their corresponding stash IDs.

Example 3: Applying Stashed Changes

Later, you may want to apply the stashed changes back to your working directory. You can do this using:

# Apply the most recent stash

git stash apply

If you have multiple stashes, you can specify a stash by its ID:

# Apply a specific stash

git stash apply stash@{2}

Example 4: Applying and Dropping Stashed Changes

You can apply and drop the changes from the stash in one command using git stash pop:

# Apply the most recent stash and drop it from the stash list

git stash pop

This is equivalent to git stash apply followed by git stash drop.

Example 5: Creating a Branch from a Stash

If you want to create a new branch from a stash, you can use:

# Create a new branch and apply the stash to it

git stash branch new-branch

This command creates a new branch (new-branch) and applies the most recent stash to it.

Example 6: Dropping Stashed Changes

If you decide you no longer need the changes in a stash, you can drop it:

# Drop the most recent stash

git stash drop

Or, you can drop a specific stash:

# Drop a specific stash

git stash drop stash@{1}

Example 7: Clearing All Stashes

To remove all stashes:

git stash clear

This removes all stashed changes, and the stash stack becomes empty.

The git stash command is a handy tool when you need to temporarily set aside changes without committing them. It provides flexibility in managing your work in progress and is particularly useful when you need to switch between branches or address unexpected issues.