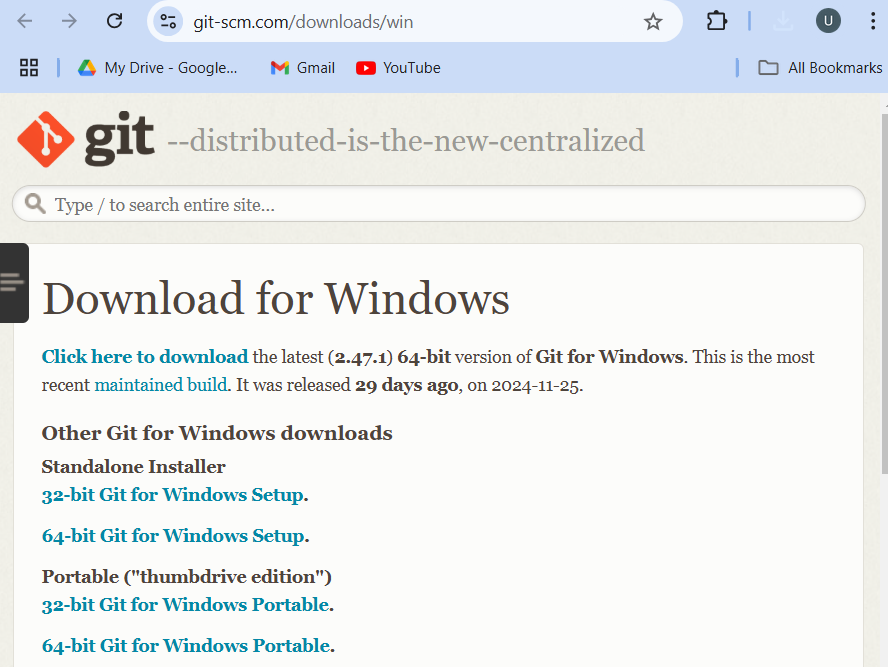
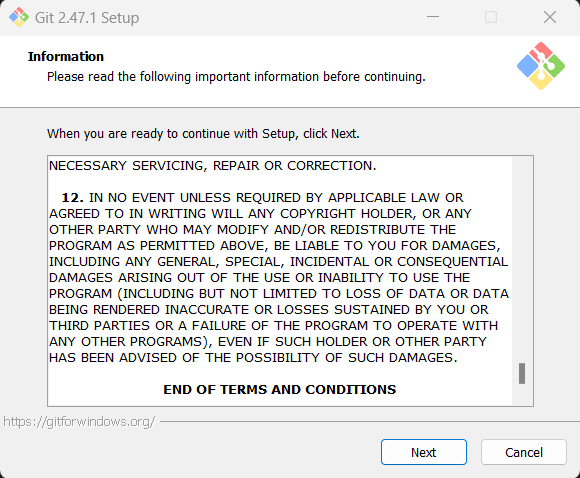
***GIT Assignment Questions***

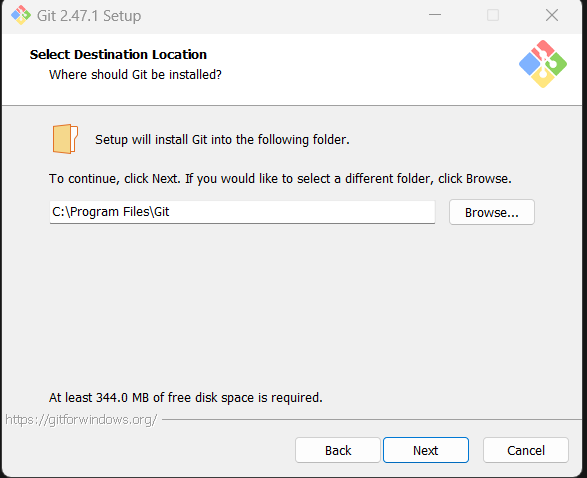
### **1. Setting Up Git:**

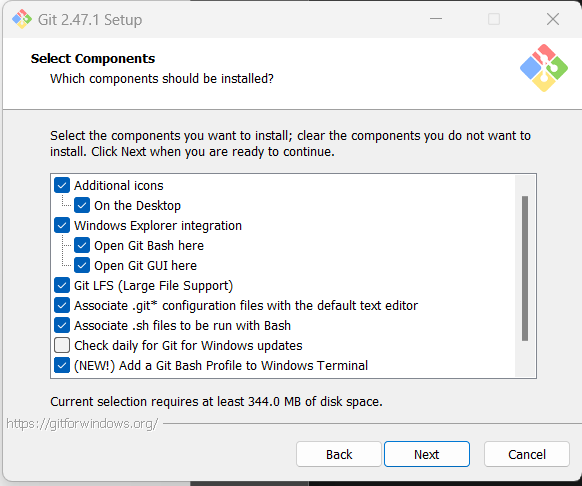
* **Q1:** Install Git on your system and configure your name and email using the following commands:
  + git config --global user.name "Your Name"
  + git config --global user.email "[your.email@example.com](mailto:your.email@example.com)"

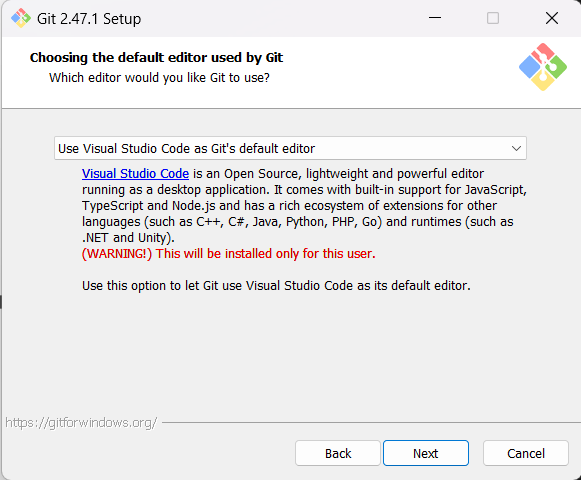
Step 1: Installing Git on computer(For **Windows**, go to [Git's download page](https://git-scm.com/download/win), and it will automatically start downloading. Once it's downloaded, run the installer and follow the instructions.)

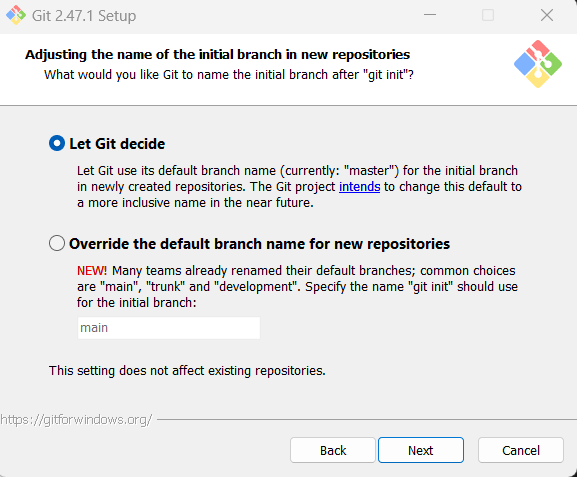


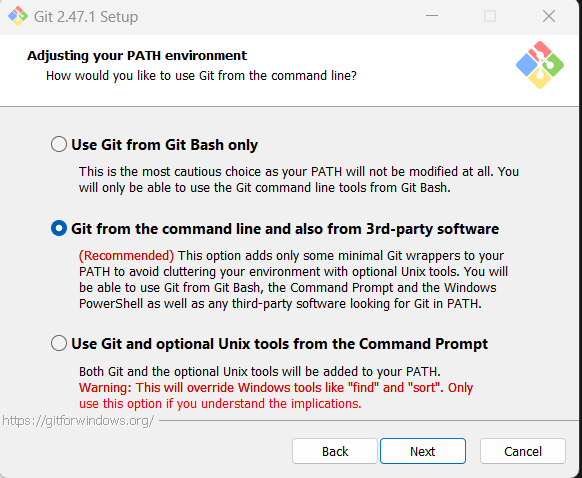












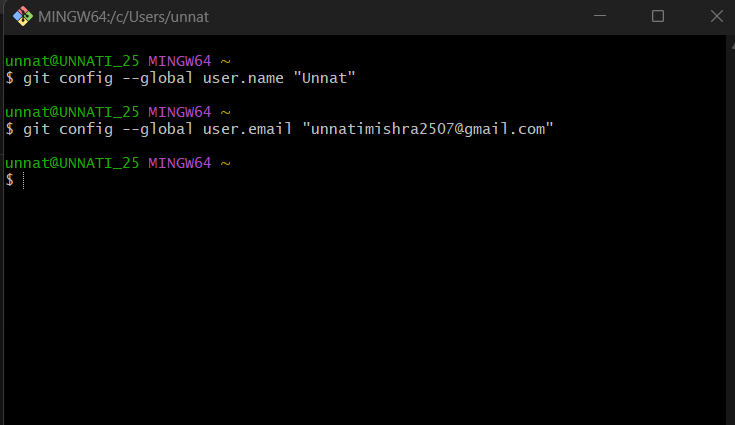
Now that you’ve installed Git

1. **Open Git Bash**: Since you're using Git Bash (MINGW64), open it if it's not already open.
2. **Configure your name**: In the Git Bash terminal, type the following command to set your name:

git config --global user.name "Unnat"

1. **Configure your email**: Now, set your email by typing this command:

git config --global user.email "unnatimishra2507@gmail.com"



* **Q2:** How would you verify that Git has been installed and properly configured? Provide the command and the expected output.

To check if Git is installed and configured correctly, run the following commands in the terminal:

git --version

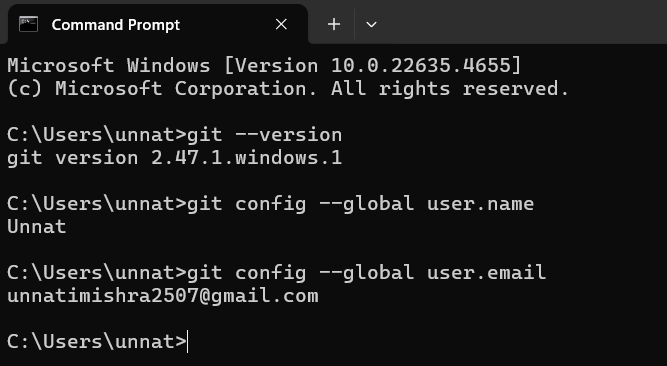
This will tell you the version of Git installed.

Next, verify your name and email with:

git config --global user.name

git config --global user.email

It should display your name and email, like:

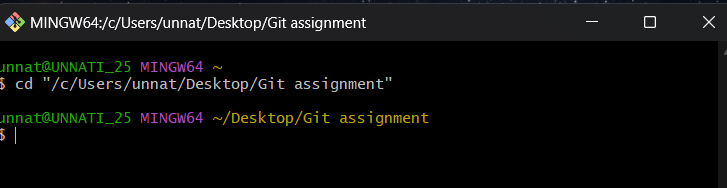


* **Q3:** Initialize a new Git repository in an empty directory on your computer using git init.

To start a new project with Git, you need to create a folder and turn it into a Git repository.

1. **Create a new folder:** Go to your file explorer and create a new folder. Let's call it “Git assignment”.
2. **Navigate to the folder from Git Bash:**

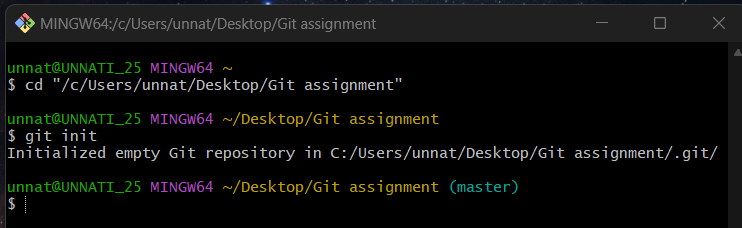
**In the Git Bash terminal, type cd followed by the path to your folder.**

****

1. **Initialize Git in the folder:** In the terminal, type the following command:

git init

This creates a new Git repository in the folder. You'll see a message like:



### **2. Basic Git Operations:**

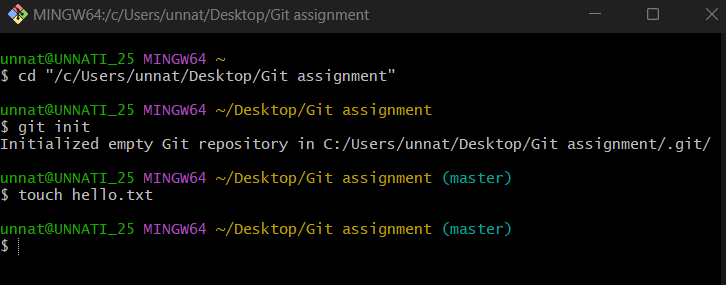
* **Q4:** Create a new text file named hello.txt in your repository. Add some content to it. Then, stage the file for commit using the git add command.

**Step 1: Create a New Text File (hello.txt)**

1. In the Git Bash terminal, make sure you're inside the folder where you've initialized your Git repository.
2. You can create a new text file using the touch command or use a text editor like Notepad.

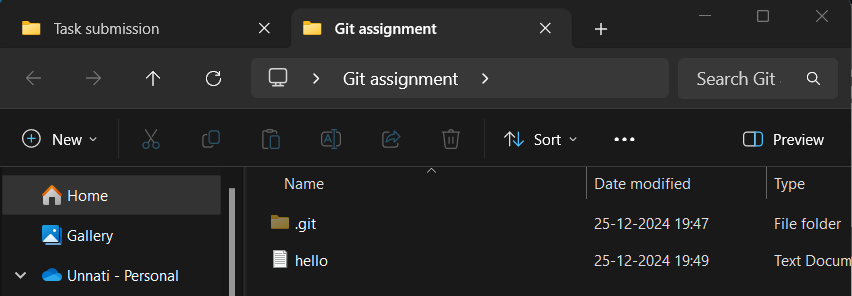
To create the file from Git Bash, run:

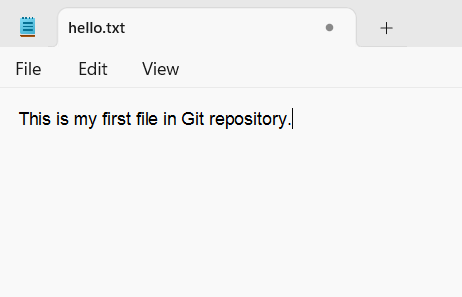
touch hello.txt



**Step 2: Add Content to the File**

1. Open the hello.txt file in a text editor (like Notepad or any editor of your choice).
2. Add some content to the file.
3. Save the file and close the editor.





**Step 3: Stage the File for Commit**

To stage the file for commit, you need to use the git add command:

1. In the Git Bash terminal, run the following command:

git add hello.txt

This command stages the hello.txt file for commit, meaning it's ready to be included in the next commit.

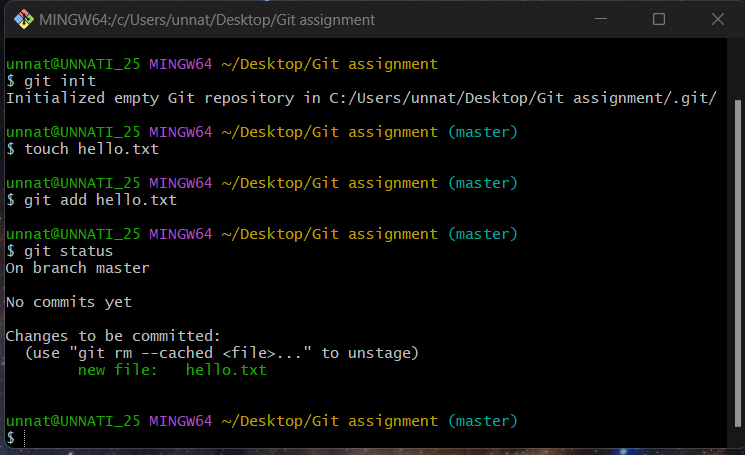
**Verify the Staging**

To verify that the file has been staged, you can use the git status command:

git status

This will show you the status of your repository.

Now your file is staged and ready for the next step (committing).



* **Q5:** Commit the changes you made to the hello.txt file with a meaningful commit message. Provide the Git command to commit and the expected output.

**Step 1: Commit the Changes**

In the Git Bash terminal, run the following command to commit the staged file:

git commit -m "Add hello.txt with initial content"

* git commit: This command creates a new commit with the changes that are staged.
* -m "Add hello.txt with initial content": This part is the commit message. The message should describe what you’ve done in this commit. In this case, it describes adding the hello.txt file with initial content.

**Step 2: Expected Output**

After running the commit command, you should see something like this:

[master (root-commit) fed9eaa] Add hello.txt with initial content

1 file changed, 1 insertion(+)

create mode 100644 hello.txt

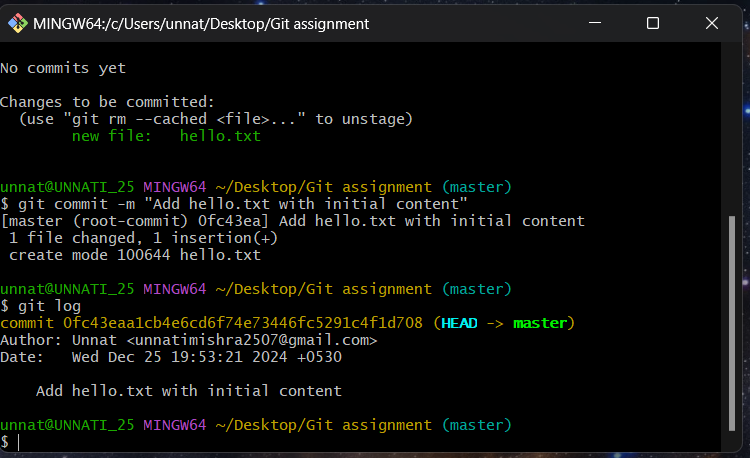
* [master (root-commit) fed9eaa]: This part shows the branch (master), the commit type (root-commit for the first commit), and the unique commit ID (a hash like 1234567).
* 1 file changed: It shows that one file (hello.txt) was changed.
* 1 insertion(+): It indicates that one line was added to hello.txt.
* create mode 100644 hello.txt: This shows that the file hello.txt was created and added to the repository.

**Step 3: Verify the Commit**

You can verify the commit by running:

git log

This will display a log of all commits, including the one you just made, and show the commit message you used.



* **Q6:** After committing your changes, use the git status command to check the state of your repository. Explain the output.

After committing your changes, you can use the git status command to check the state of your repository.

**Run the Command**

git status

**Expected Output**

After you commit your changes, the output of git status should look like this:

On branch master

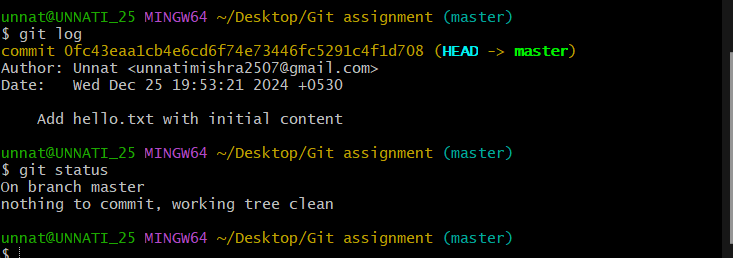
nothing to commit, working tree clean

**Explanation of the Output:**

* **On branch master**: You are on the master branch, which is the default branch in Git.
* **nothing to commit, working tree clean**: This means there are no changes in your working directory that need to be committed. Your repository is in a clean state because you’ve already committed all the changes.

This is a good sign—it means you've successfully committed your changes and there are no untracked or uncommitted changes left in the repository.

Let me know if you'd like to move on to the next task or if you need further clarification!



* **Q7:** How can you view the commit history of a repository? Use the git log command and describe what information it provides.

To view the commit history of a repository, you can use the git log command.

**Step 1: Run the Command**

In your Git Bash terminal, type:

git log

**Step 2: Expected Output**

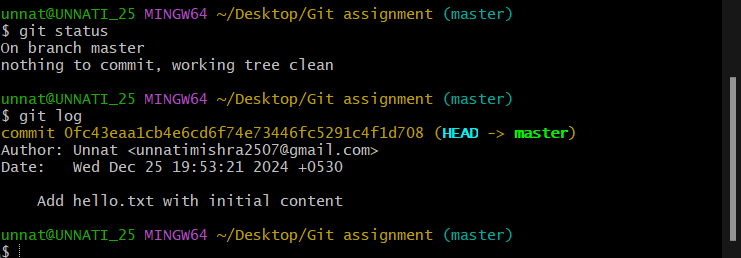
The git log command will display something like this:

commit 123456789abcdef123456789abcdef12345678 (HEAD -> master)

Author: Unnat <unnatimishra2507@gmail.com>

Date: Wed Dec 25 12:34:56 2024 +0530

Add hello.txt with initial content



**Explanation of the Output**

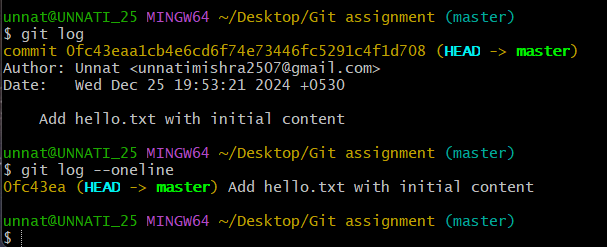
1. **commit 123456789abcdef123456789abcdef12345678**:
   * This is the unique **commit hash** (a long alphanumeric string) that identifies the commit. Each commit has its own hash.
2. **(HEAD -> master)**:
   * HEAD: This indicates the current position in the Git history (where you are currently working).
   * master: The branch where the commit was made.
3. **Author:** Unnat <unnatimishra2507@gmail.com>:
   * The name and email of the person who made the commit, as configured in Git.
4. **Date: Wed Dec 25 12:34:56 2024 +0530**:
   * The date and time when the commit was made, along with the time zone.
5. **Commit Message ("Add hello.txt with initial content")**:
   * The message you provided when making the commit, describing what changes were made.

**Additional Options for git log**

* To see a **simplified one-line history**, you can use:

git log --oneline

This will display each commit as a single line, showing the hash and commit message:



### **3. Branching and Merging:**

* **Q8:** What is the purpose of branching in Git? How do branches help in software development?

**Purpose:** Branching in Git allows developers to **create separate workspaces** (branches) for different tasks, features, or bug fixes without interfering with the main codebase. Each branch is essentially an independent line of development, allowing changes to be made, tested, and finalized before merging them back into the main branch.

How Branches Help in Software Development

1. **Isolated Development**:  
   Developers can work on features or fixes in separate branches without messing up the main code.
2. **Collaboration**:  
   Team members can work on their own branches, avoiding conflicts with others’ work.
3. **Version Control**:  
   Keep separate branches for stable code, ongoing development, or experimental ideas.
4. **Safe Experimentation**:  
   Test risky changes in a branch. If it fails, just delete the branch without affecting the main project.
5. **Code Review**:  
   Before merging changes, branches allow easy testing and review to ensure quality.
6. **CI/CD**:  
   Code in branches can be automatically tested and deployed once it’s ready.

Branches keep everything organized, safe, and collaborative

* **Q9:** Create a new branch called feature-branch and switch to it using the appropriate Git command.

Create a Branch:

git branch feature-login

Switch to the New Branch:

git checkout feature-login

(or combine the two steps: git checkout -b feature-login)

To create a new branch called feature-branch and switch to it, use the following command in Git Bash:

**Command:**

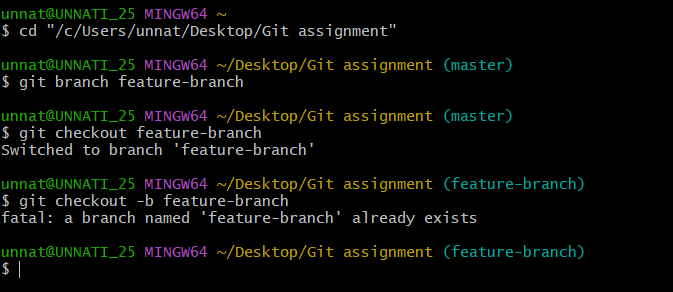
git checkout -b feature-branch

**Explanation:**

* **git checkout -b**: This command creates a new branch and switches to it in one step.
* **feature-branch**: The name of the new branch you're creating.

**Expected Output:**

After running the command, you should see something like this:



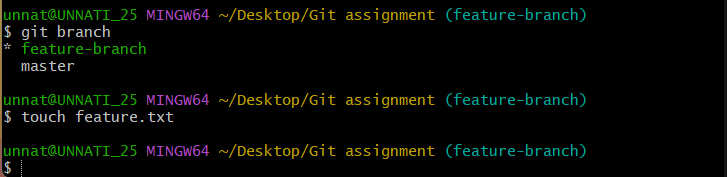
Switched to a new branch 'feature-branch'

This means the new branch feature-branch has been created, and you are now working in that branch.

**Verify the Branch:**

You can confirm which branch you're on by running:

git branch

****

It will list all branches, with the current branch highlighted with an asterisk (\*):

\* feature-branch

master

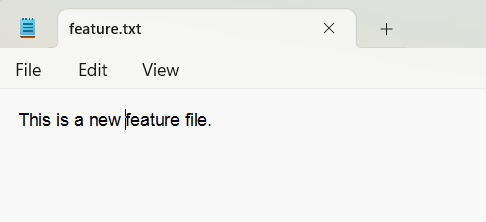
* **Q10:** Create a new file named feature.txt on your new branch and commit the changes. Then, switch back to the main branch.

**Step 1: Create the feature.txt file**

1. Make sure you're on the feature-branch (use git branch to verify).
2. In the Git Bash terminal, run the following command to create the file:

touch feature.txt

1. Open the feature.txt file in a text editor (e.g., Notepad) and add some content, such as:

This is a new feature file. 

1. Save and close the file.

**Step 2: Stage the File for Commit**

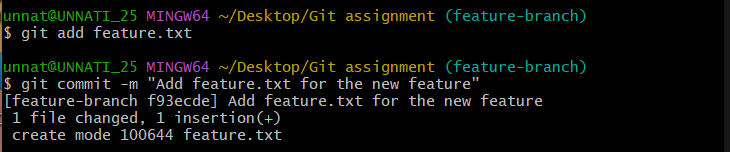
In the Git Bash terminal, stage the new file using:

git add feature.txt

**Step 3: Commit the Changes**

Now, commit the new file with a meaningful commit message:

git commit -m "Add feature.txt for the new feature"



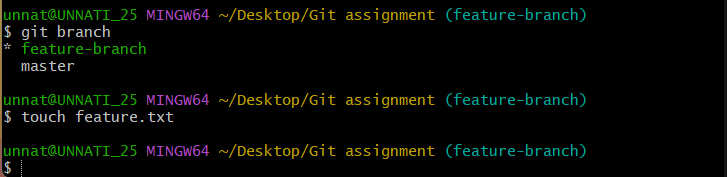
**Step 4: Switch Back to the Main Branch**

After committing the changes on feature-branch, switch back to the main branch using:

git checkout main

**Verify Your Changes**

* **git status**: You can check the status to confirm that you're on the main branch and there are no uncommitted changes.
* **git log**: This will show the commit history, and you’ll see the commit you made on feature-branch in the history when you switch back.



* **Q11:** Merge the feature-branch into the main branch. What command would you use to merge the changes, and what happens if there are no conflicts?

To **merge** the feature-branch into the main branch, follow these steps:

**Step 1: Switch to the main branch**

First, make sure you are on the main branch (or master, depending on your repository's naming convention). You can check the current branch with:

git branch

Then, if needed, switch to the main branch:

git checkout main

**Step 2: Merge the feature-branch into main**

Now, to merge the changes from feature-branch into main, use the following command:

git merge feature-branch

**What Happens if There Are No Conflicts:**

* If there are **no conflicts** between the branches, Git will automatically merge the changes, and you will see output similar to:

Updating 1234567..89abcdef

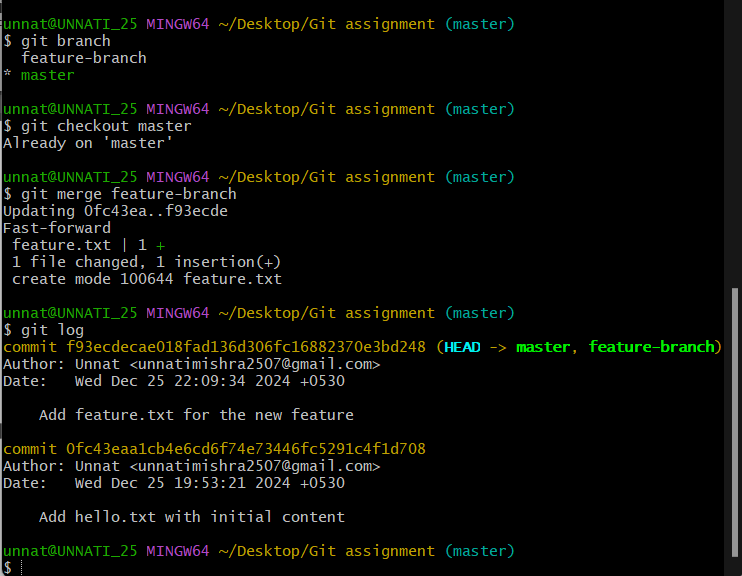
Fast-forward

feature.txt | 1 +

1 file changed, 1 insertion(+)

create mode 100644 feature.txt

* + **Fast-forward merge**: If main was directly behind feature-branch without any changes in main, it will automatically fast-forward the main branch to the feature-branch.
  + **1 file changed**: This shows how many files were modified during the merge.



**Step 3: Verify the Merge**

You can check the commit history to confirm that the merge has been successfully completed:

git log

This should show a new commit in the history, indicating the merge of feature-branch into main.

* **Q12:** What is a merge conflict? Create a scenario where a merge conflict occurs and explain how you would resolve it.

A **merge conflict** occurs when Git is unable to automatically merge changes from two branches because both branches have modifications to the same part of the same file, and Git cannot determine which version should be kept.

Scenario:

Let’s walk through a simple example:

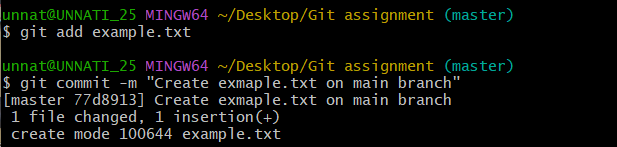
1. **Step 1: Create a File on main Branch**
   * You create a file called example.txt in the main branch and add some text:

Hello from main branch.

* + Commit the changes:

git add example.txt

git commit -m "Create example.txt on main branch"



1. **Step 2: Create and Make Changes in feature-branch**
   * You create and switch to a feature-branch:

git checkout -b feature-branch

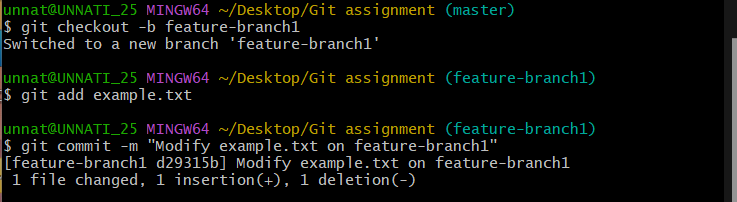
* + You modify the example.txt file in the feature-branch:

Hello from feature-branch.

* + Commit the changes:

git add example.txt

git commit -m "Modify example.txt on feature-branch"



1. **Step 3: Modify the Same File in main Branch**
   * Switch back to the main branch:

git checkout main

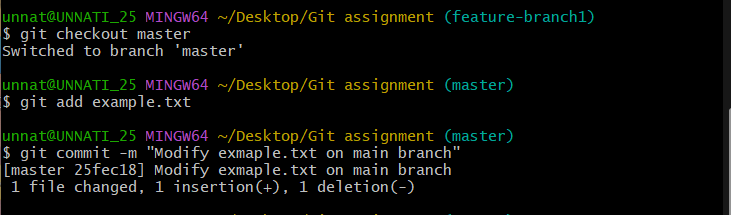
* + Modify the same example.txt file differently:

Hello from main branch with a new change.

* + Commit the changes:

git add example.txt

git commit -m "Modify example.txt on main branch"



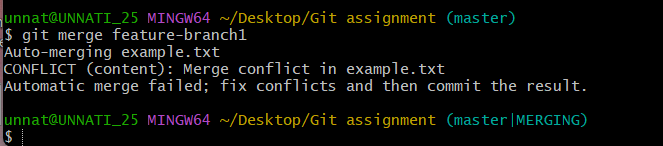
1. **Step 4: Try to Merge feature-branch into main**
   * Now, when you try to merge feature-branch into main:

git merge feature-branch

* + **Merge conflict occurs** because both branches modified the same lines in example.txt, and Git can’t decide which version to keep.

**What Happens During a Merge Conflict:**

Git will show a message like this:

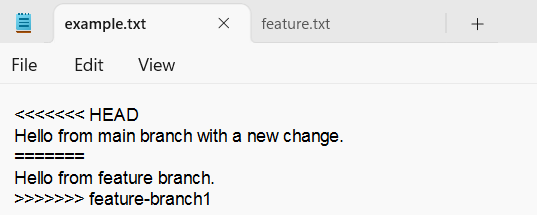


Auto-merging example.txt

CONFLICT (content): Merge conflict in example.txt

Automatic merge failed; fix conflicts and then commit the result.

In the file example.txt, you’ll see something like this:



<<<<<<< HEAD

Hello from main branch with a new change.

=======

Hello from feature-branch.

>>>>>>> feature-branch1

* **<<<<<<< HEAD**: This is the version from the current branch (main).
* **=======**: Separates the conflicting changes.
* **>>>>>>> feature-branch**: This is the version from the branch being merged (feature-branch).

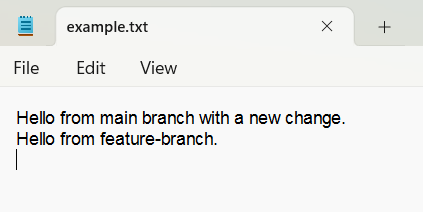
**How to Resolve a Merge Conflict:**

1. **Manually Edit the Conflicting File**:  
   Open example.txt and decide which changes to keep, or combine the changes. For example:

Hello from main branch with a new change.

Hello from feature-branch.

After editing, remove the conflict markers (<<<<<<<, =======, >>>>>>>).

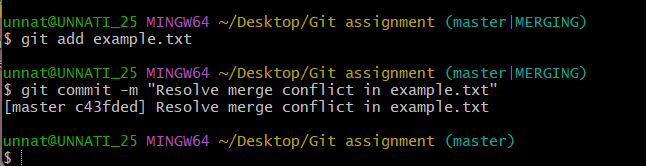


1. **Stage the Resolved File**: Once you've resolved the conflict, stage the file for commit:

git add example.txt

1. **Commit the Merge**: Finally, commit the merge with a message:

git commit -m "Resolve merge conflict in example.txt"



**Summary of Steps:**

* Git shows a conflict when merging changes to the same part of a file.
* You manually resolve the conflict by editing the file and choosing or combining changes.
* After resolving, you stage and commit the changes to complete the merge.

Merge conflicts are common in collaborative work, but they can be easily resolved by carefully reviewing and editing the conflicting changes.

### **4. Working with Remote Repositories:**

* **Q13:** What is a remote repository in Git? How is it different from a local repository?

**What is a Remote Repository in Git?**

A **remote repository** in Git is a version of your project that is stored on a server (online or on a network). It allows multiple people to collaborate on the same project. Common remote repository hosting platforms include GitHub, GitLab, Bitbucket, and others.

Remote repositories are used to **share code** between developers, **back up** work, and **collaborate** on a project from different locations.

**Difference Between Remote and Local Repositories**

1. **Location**:
   * **Local Repository**: This is the Git repository that exists on your own computer. It contains your project’s files and the entire version history.
   * **Remote Repository**: This is the Git repository hosted on a server, often online (e.g., GitHub, GitLab). It is used for sharing your code with others and collaborating.
2. **Access**:
   * **Local Repository**: You can access and work with your local repository directly from your computer without needing an internet connection.
   * **Remote Repository**: You need an internet connection to interact with the remote repository. You push changes to it or pull updates from it.
3. **Purpose**:
   * **Local Repository**: Used for personal development. It’s where you create, modify, and commit your changes before sharing them.
   * **Remote Repository**: Used for collaboration, backup, and sharing. It allows other team members to access your work and contribute to the project.
4. **Git Commands**:
   * **Local Repository**: Operations like git add, git commit, git status, etc., are done in the local repository.
   * **Remote Repository**: You interact with the remote repository using commands like git push (to upload changes), git pull (to download updates), and git clone (to make a local copy of the remote repository).

* **Q14:** Clone a remote repository from GitHub to your local machine using the git clone command. Provide the URL of a public repository to clone.

To **clone a remote repository** from GitHub to your local machine, follow these steps:

**Step 1: Find a Public Repository URL**

Here’s an example of a public repository URL you can use:

https://github.com/octocat/Hello-World.git

This is the URL for GitHub's "Hello-World" repository by their official account.

**Step 2: Use the git clone Command**

In your Git Bash terminal, run the following command to clone the repository:

git clone https://github.com/octocat/Hello-World.git

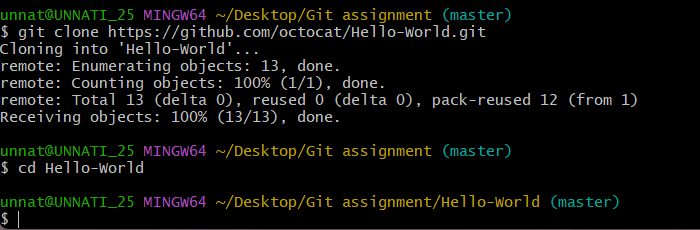
**What Happens After Cloning:**

* Git will create a folder on your local machine with the repository's name (in this case, Hello-World).
* All the files, commit history, and branches from the remote repository will be copied to your local machine.

**Step 3: Navigate to the Cloned Repository**

After cloning, navigate into the newly created folder:

cd Hello-World



Now, you're working with a local copy of the repository, and you can make changes, commit, push, or pull as needed.

* **Q15:** After cloning the repository, make a small change (e.g., edit README.md), and commit the changes to your local repository.

To make a small change after cloning the repository and commit it to your local repository, follow these steps:

**Step 1: Open the Cloned Repository**

First, navigate to the directory of the cloned repository:



**Step 2: Edit the README.md File**

* Open the README.md file in your favorite text editor (e.g., Notepad or Visual Studio Code).
* Add some small change, for example:

This is my small change to the README file.

**Step 3: Stage the Changes**

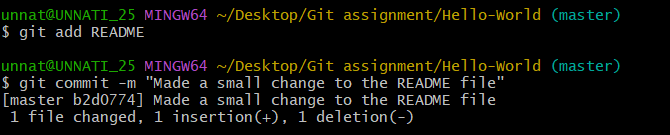
After saving the changes to README.md, go back to the Git Bash terminal and use the following command to stage the changes:

git add README.md

**Step 4: Commit the Changes**

Now, commit the changes with a meaningful message:

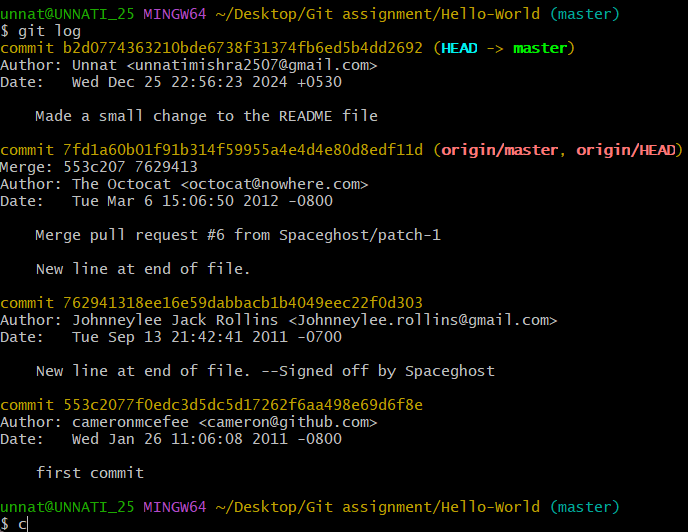
git commit -m "Made a small change to the README.md file"



**Step 5: Verify the Commit**

You can verify that the commit has been made by running:

git log



This will show the commit history, and you should see your commit listed.

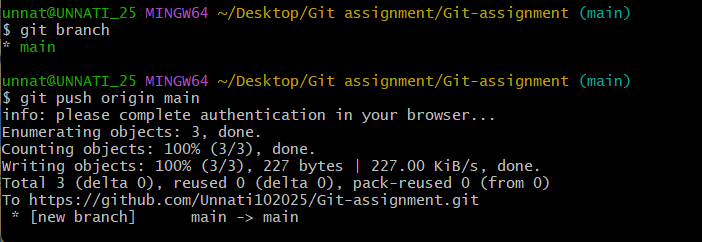
* **Q16:** Push your local commits to the remote repository. What Git command is used to push changes to a remote repository? Explain how you would use it.

After committing your changes locally, the next step is to **push** those changes to the remote repository (e.g., GitHub). The **Git command** used for this is git push.

**Steps to Push Your Changes:**

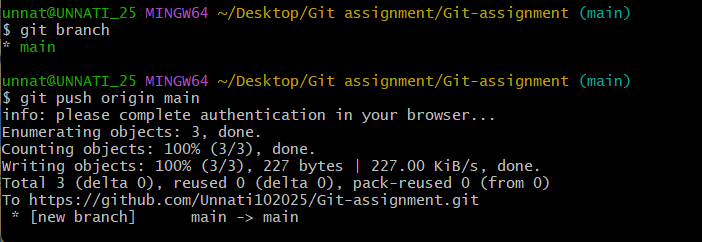
1. **Ensure You're on the Correct Branch:** Before pushing, make sure you're on the branch where you made the changes (usually master or main). You can check your current branch with:

git branch



1. **Push the Changes:** Use the following command to push your local commits to the remote repository:

git push origin master



* + origin: This refers to the default remote repository (where the repository was cloned from).
  + master: This is the name of the branch you're pushing to. If you’re on main (instead of master), replace master with main.

1. **Authentication:** If you're pushing to a remote repository like GitHub, Git may ask for your GitHub username and password (or a personal access token) to authenticate the push.

**What Happens When You Push:**

* **Pushes your local commits** to the remote repository, updating it with your changes.
* If the push is successful, you’ll see a message like:

Counting objects: 10, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (8/8), done.

Writing objects: 100% (10/10), 1.20 KiB | 1.20 MiB/s, done.

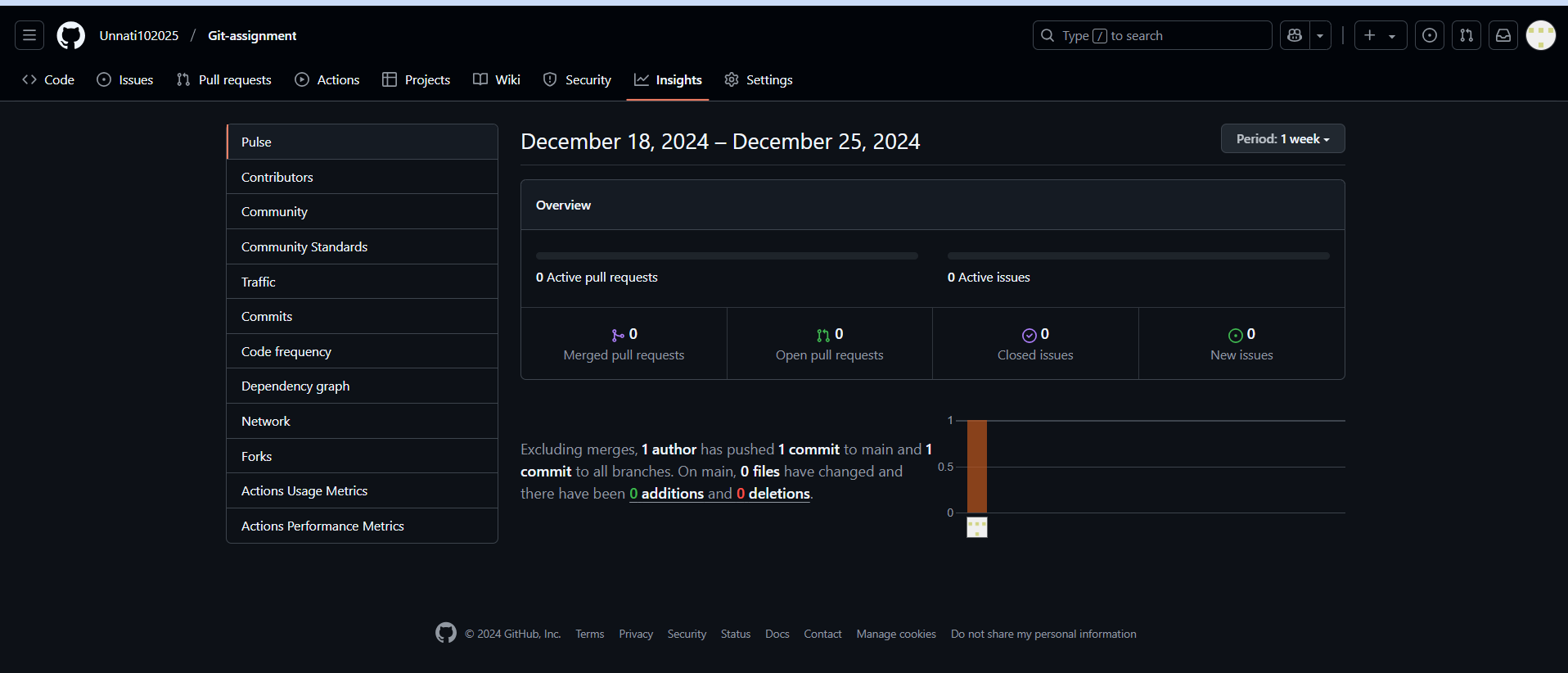
Total 10 (delta 4), reused 0 (delta 0)

To https://github.com/octocat/Hello-World.git

\* [new branch] master -> master

**Verify:**

To confirm your changes have been pushed, you can go to your repository on GitHub (or whichever platform you're using) and check the commit history or the file changes.

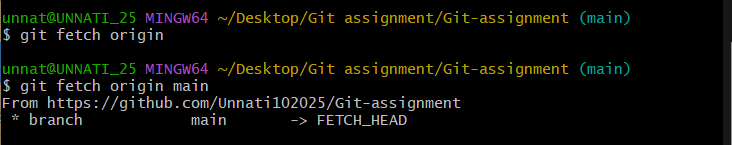


* **Q17:** Fetch the latest changes from the remote repository using the git fetch command. What is the difference between git fetch and git pull?

To fetch the latest changes from the remote repository, you can use the git fetch command.

git fetch origin

This command retrieves updates from the remote repository but **does not merge** them into your local branch automatically. It simply fetches the changes and updates your remote-tracking branches.



**Difference between git fetch and git pull:**

* **git fetch**:
  + Fetches the changes from the remote repository without modifying your local working directory.
  + It downloads all the new data (commits, branches, etc.) but doesn't automatically merge it into your current branch.
  + You will need to merge or rebase the fetched changes manually if you want them in your current branch.

Example:

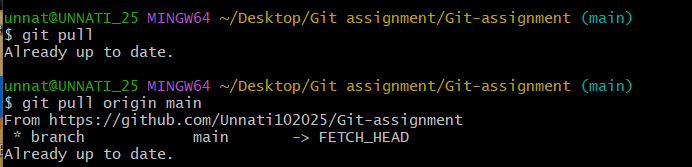
git fetch origin

After fetching, you can inspect changes using git log origin/main, and then merge them using git merge origin/main.

* **git pull**:
  + Combines the functionality of git fetch and git merge.
  + It fetches the latest changes from the remote and **immediately attempts to merge** those changes into your local branch.
  + This command is more convenient for automatically syncing your local branch with the remote branch.

Example:

git pull origin main



**Key Differences:**

* git fetch retrieves the updates but leaves them unmerged, allowing you to review the changes before merging.
* git pull fetches and merges the changes in one step, updating your local branch immediately.

### **5. Undoing Changes in Git:**

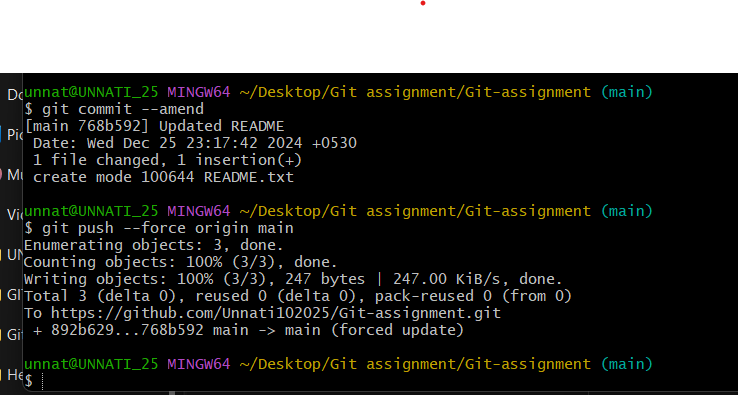
* **Q18:** After making several commits, you realize that a commit message needs to be changed. How can you edit the last commit message using Git?

If you need to change the message of the **last commit**, you can use the git commit --amend command. Here’s how to do it:

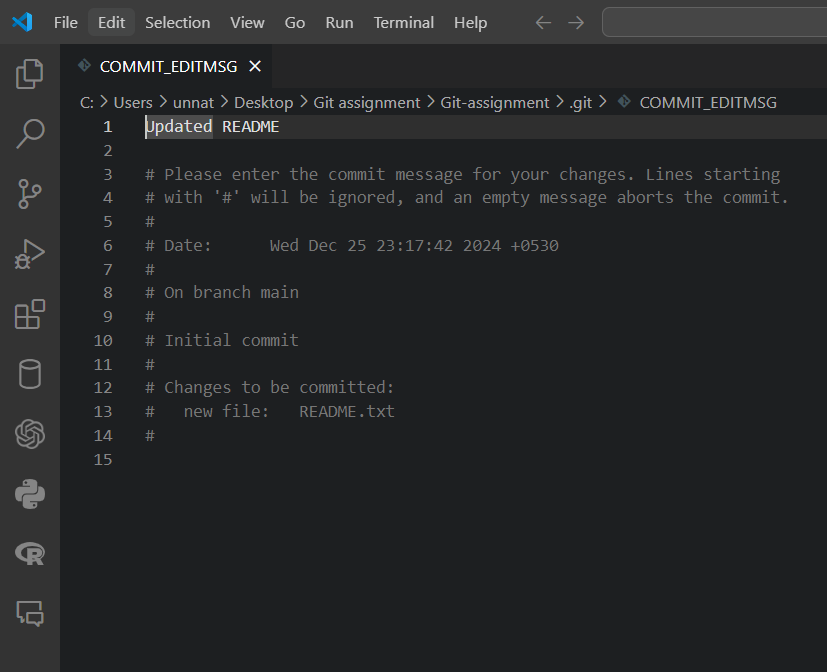
**Steps to Edit the Last Commit Message:**

1. **Amend the Commit Message**: Run the following command to amend the message of the last commit:

git commit –amend



1. **Edit the Commit Message**: This will open your default text editor (like Vim, Nano, or Visual Studio Code) with the last commit message. Modify the message as needed.



1. **Save and Exit**: After editing the commit message, save the changes and close the editor
2. **Force Push (if already pushed to remote)**: If you've already pushed the commit to the remote repository, you will need to **force push** to update the remote commit:

git push --force origin <branch-name>

Replace <branch-name> with the name of your branch (e.g., main or feature-branch).

**Notes:**

* This method only changes the **last commit**. If you need to change a commit further back in history, you will need to use **interactive rebase**.
* Be careful with git push --force if working in a shared repository because it rewrites history, which can cause problems for other collaborators.

