**Theory Questions**

**Please answer the following questions briefly by providing use cases.**

**Q1] Why do we create branches in a repository and why do we create pull requests instead of merging directly?**

**Ans]** We create Branches in a repository because of following use cases:

* **Feature Development:** A developer branches out from the main codebase to focus on a new feature they are creating for an application. This ensures that the main codebase remains stable until the feature is finished and tested by enabling them to make changes without impacting the main branch.
* **Bug Fixes**: If a bug is found in the production version of the software, a developer can create a branch from the production code to fix the bug. This isolates the bug fix from other ongoing developments.
* **Experiments:** Developers can create branches to experiment with new technologies or approaches without risking the stability of the main project.

We create pull requests instead of merging directly because of the following use cases:

* **Code Review**: Before merging, developers can ask their peers to examine the code changes they've made by submitting a pull request. This procedure aids in finding errors, upholding coding conventions, and enhancing the overall caliber of the code.
* **Collaboration**: Pull requests help team members talk about the modifications with each other. Individual lines of code can be discussed, tweaks can be made, and the implementation can be discussed by the team.
* **Change Tracking:** Pull requests provide a clear history of what changes were made, why they were made, and who made them. This documentation is useful for future reference and debugging

**Q2] What is the difference between git add . and git add <filename>? What will we use when we have changes in multiples but we are not required to add some files?**

**Ans] git add .**: This command stages all the changes in the current directory and its subdirectories, including new, modified, and deleted files.

**Use Case**: When you want to stage all the changes in the project for the next commit, ensuring that every modified or new file is included.

**git add <filename>:** This command stages only the specified file.

**Use Case:** When you only want to stage changes for a particular file, leaving other changes in the working directory uncommitted.

If there are changes in multiple files but we are not required to add some files then we will use following methods

**1] Selective staging by filename**

Command: git add <filename1> <filename2> <filename3>

This is used to explicitly add only the files you want to stage.

**2]Interactive staging with patch mode**

* Use git add -p to interactively review and stage changes.
* This command presents each change in your working directory one at a time, allowing you to decide whether to stage each change, skip it, or split it into smaller hunks.

**Use Case Example**

**Scenario:** You have changes in index.html, styles.css, and script.js, but you only want to stage index.html and specific parts of styles.css.

1. **Stage Specific Files:**

* git add index.html

1. **Interactively Stage Parts of a File:**

* git add -p styles.css
* You will be prompted with each change in styles.css and can choose to stage (y), skip (n), or split (s) the changes.

These methods provide you with control over which changes are staged for the next commit, ensuring that only the desired modifications are included.

**Q3] What is the difference between git fetch and git pull?**

**Ans] Git Pull:**

* The "git pull" command is a combination of two other Git commands: "git fetch" and "git merge." It fetches changes from the remote repository and automatically merges them into the current branch.

**Example**: git pull origin master

In this example, we are pulling changes from the "master" branch of the remote repository named "origin" into the current branch.

**Git Fetch:**

* The "git fetch" command only retrieves changes from the remote repository but does not automatically merge them into the current branch. Instead, it updates the remote-tracking branches.

**Example:** git fetch origin

In this example, we are fetching changes from the remote repository named "origin" without merging them into the current branch.

**Differences:**

* **Automatic Merge:** The most significant difference between "git pull" and "git fetch" is that "git pull" automatically merges the fetched changes into the current branch, while "git fetch" does not. This makes "git pull" a more convenient command if you want to quickly update your local branch with changes from the remote repository.
* **Local Branch vs. Remote-Tracking Branch:** When using "git pull," the changes are merged directly into the local branch. On the other hand, "git fetch" updates the remote-tracking branches, which are references to the state of the remote branches in the remote repository.
* **Safety:** Using "git fetch" can be considered safer than "git pull" since it does not modify your working directory or current branch until you explicitly merge the changes. This gives you more control over when and how you want to integrate the fetched changes into your code.

**Q4] What is a head in a repository and what does it do?**

**Ans]** HEAD is the reference to the most recent commit in the current branch. This means HEAD is just like a pointer that keeps track of the latest commit in your current branch.

HEAD answers the question: **Where am I right now in the repository?**

HEAD pointer can be in either of two states: **attached or detached**.

The default state is attached, where any manipulation of the history is automatically recorded to the branch HEAD is currently referencing.

In a detached state, experimental changes can be made without impacting any existing branch, as HEAD is referencing the underlying commit directly and is not "attached" to a particular branch.

With a good understanding of the HEAD pointer, you can swiftly navigate the history of your repository and perform operations as you see fit.

It contains an immutable snapshot of your entire code base at a given time.

In summary, HEAD is a crucial pointer in Git that helps manage the state of the repository by tracking the current branch and commit, facilitating operations like commits, checkouts, merges, and more.

**Q5] What is the .git folder in a repository?**

**Ans]** The **.git** folder contains all information that is necessary for the project and all information relating commits, remote repository address, etc. It also contains a log that stores the commit history. This log can help you to roll back to the desired version of the code.

The hidden **.git** folder can be viewed using the following command in the git bash terminal − $ ls −a

Following are a few important sub−directories and files in the **.git** folder −

* **hooks** − This folder contains script files. Git hooks are the scripts that are executed before or after events like commit, push etc.
* **objects** − This folder represents an object database of Git.
* **config** − This is the local configuration file.
* **refs** − This folder stores information about tags and branches.
* **HEAD** − This file stores reference to the current branch. It points to the master branch by default.
* **index** − This is a binary file and stores staging information

The **.git** folder will contain details of every single change made to the code base. All snapshots of the modifications will be recorded in this folder like a database, which makes it possible to undo the changes and rollback to the desired version of the code.

The **.git** folder is hidden to prevent accidental deletion or modification of the folder. The version history of the code base will be lost if this folder is deleted. This means, we will not be able to rollback changes made to the code in future.

**Q6] What are commit hashes and its use cases?**

**Ans]** A commit hash (also known as a commit ID or SHA-1 hash) is a unique identifier generated by Git for each commit. It is a 40-character string that is computed based on the contents of the commit, including the changes, author information, timestamp, and parent commits.

Example: e3c7a5b7c9ad6d3ebbb1a9b7eafacb57e7eaf733

**Use Cases of Commit Hashes:**

1. **Identifying Specific Commits**:
   * **Purpose**: Uniquely identify and reference a specific commit in the repository's history.
   * **Example**: You can checkout a specific commit using its hash:
   * git checkout e3c7a5b7c9ad6d3ebbb1a9b7eafacb57e7eaf733
2. **Reviewing Changes**:
   * **Purpose**: Examine the changes introduced by a specific commit.
   * **Example**: View the differences introduced by a commit:
   * git show e3c7a5b7c9ad6d3ebbb1a9b7eafacb57e7eaf733
3. **Creating References**:
   * **Purpose**: Use commit hashes in branches, tags, and other Git operations to create references to specific commits.
   * **Example**: Tag a specific commit:
   * git tag v1.0 e3c7a5b7c9ad6d3ebbb1a9b7eafacb57e7eaf733
4. **Tracking Changes**:
   * **Purpose**: Keep a precise record of the history and track the changes introduced in each commit.
   * **Example**: List commit history along with hashes:
   * git log --oneline
5. **Merging and Rebasing**:
   * **Purpose**: Use commit hashes to identify commits during merge and rebase operations.
   * **Example**: Rebase a branch onto a specific commit:
   * git rebase e3c7a5b7c9ad6d3ebbb1a9b7eafacb57e7eaf733
6. **Debugging and Problem Resolution**:
   * **Purpose**: Pinpoint and resolve issues by identifying and reverting problematic commits.
   * **Example**: Revert a specific commit:
   * git revert e3c7a5b7c9ad6d3ebbb1a9b7eafacb57e7eaf733
7. **Collaboration and Code Review**:
   * **Purpose**: Share commit hashes with team members for collaboration, code reviews, and discussions.
   * **Example**: Share a commit hash in a code review tool to discuss specific changes.

**Q7] Different ways of syncing a branch with origin**

**Ans**] Different ways of syncing a branch with origin are:

* git pull: Fetches and merges changes from the remote branch.
* git fetch and git merge: Separately fetches changes and then merges them, allowing review before merging.
* git fetch and git rebase: Fetches changes and replays local commits on top of them, creating a cleaner history.
* git pull --rebase: Combines fetching and rebasing in one step.

**Practical Questions**

**Scenario 1: Conflict Resolution**

**Task: Imagine you and your teammate are working on the same file, project.txt. Both of you make changes to the same lines and commit them. Let’s say your teammate’s changes are merged first. And now when you try to merge your changes, a merge conflict occurs.**

**Expected result: Changes from both branches should be included in the file in the main branch. The changes should be included in the sequence: first your teammate’s changes and then the changes made by you.**

**Solution**

**Step 1: Create a repository and add a file named project.txt.**

**Intialising a repository**: git init

This creates a new subdirectory named .git that contains all of your necessary repository files

**To track files**

git add . (to add all the files)

git add project.txt (to add specific file)

**To move the files from staging area to your commit**

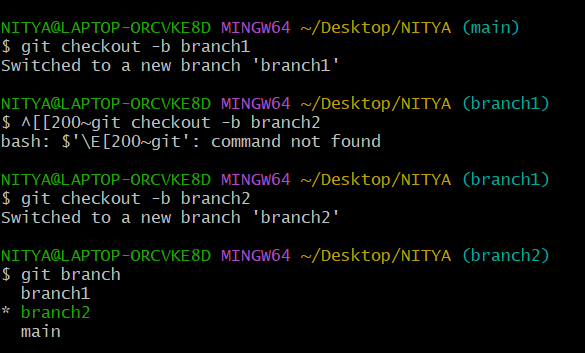
Use the command git commit -m “first commit”



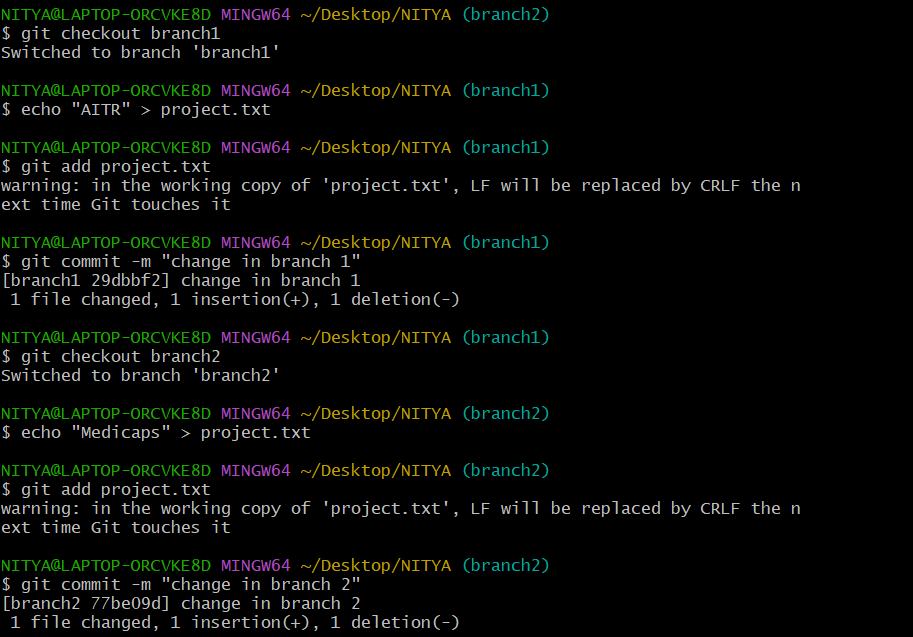
**Step 2: Create two branches in the repository.**

To create and check out that branch we will use the command

Git checkout -b <branch name>



**Step 3: Make changes to the same line and in the same file in both branches.**

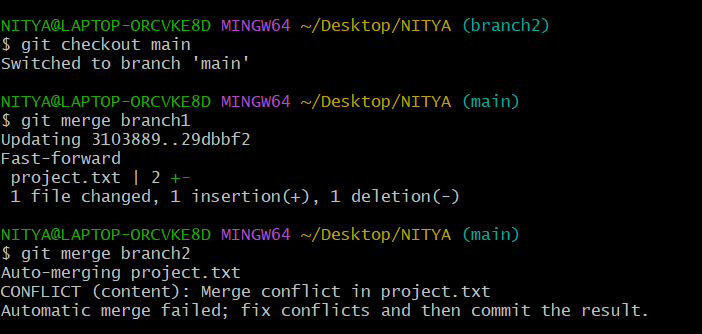


In both the branches we have made the changes at the same place. The change in branch 1 is done by teammates and the change in branch 2 is done by me.

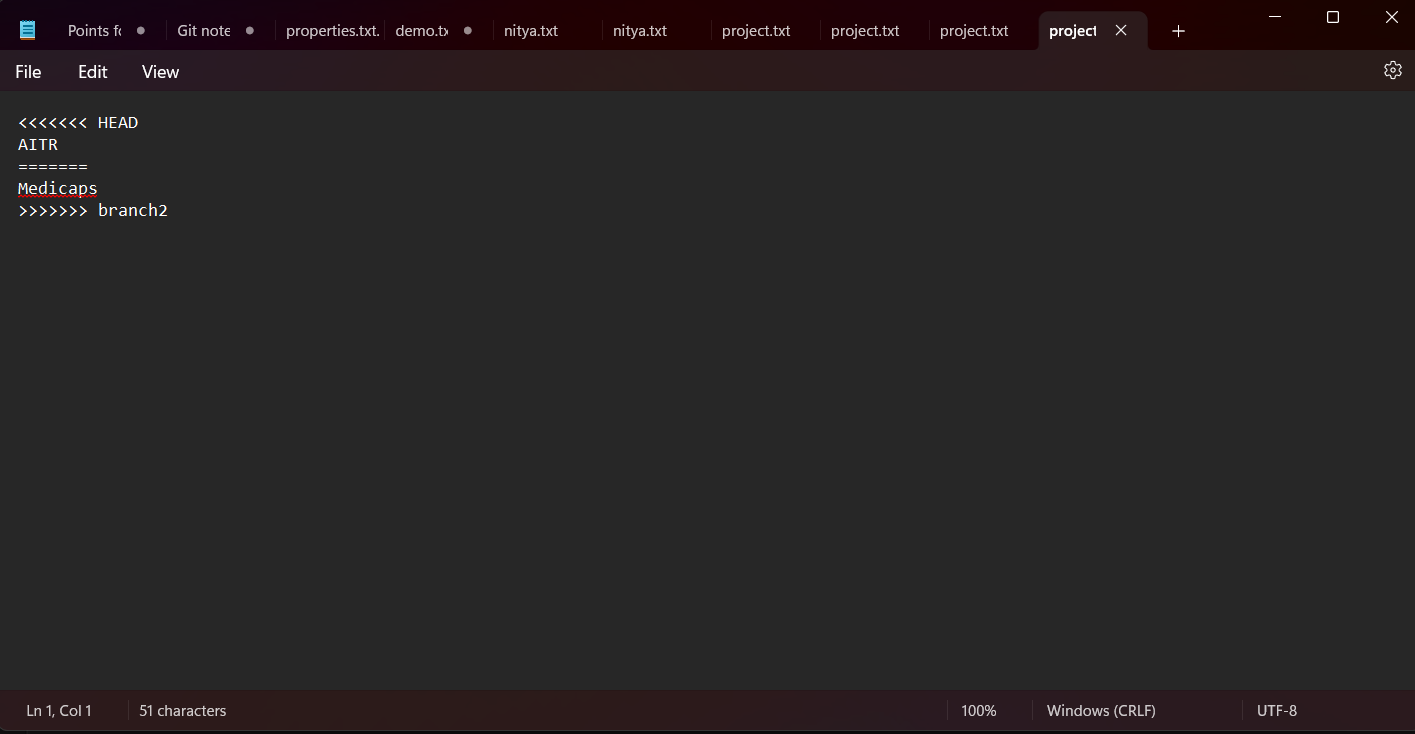
**Step 4 : Commit the changes in both branches.**

After making the changes we have committed the changes as shown in above ss.

**Step 5: Merge one branch into the main first and then try to merge the second branch.**

****

Git detects that both branches have modified the same lines in project.txt, resulting in a merge conflict. The conflicted state is indicated in the terminal, and Git marks the conflict in the file.

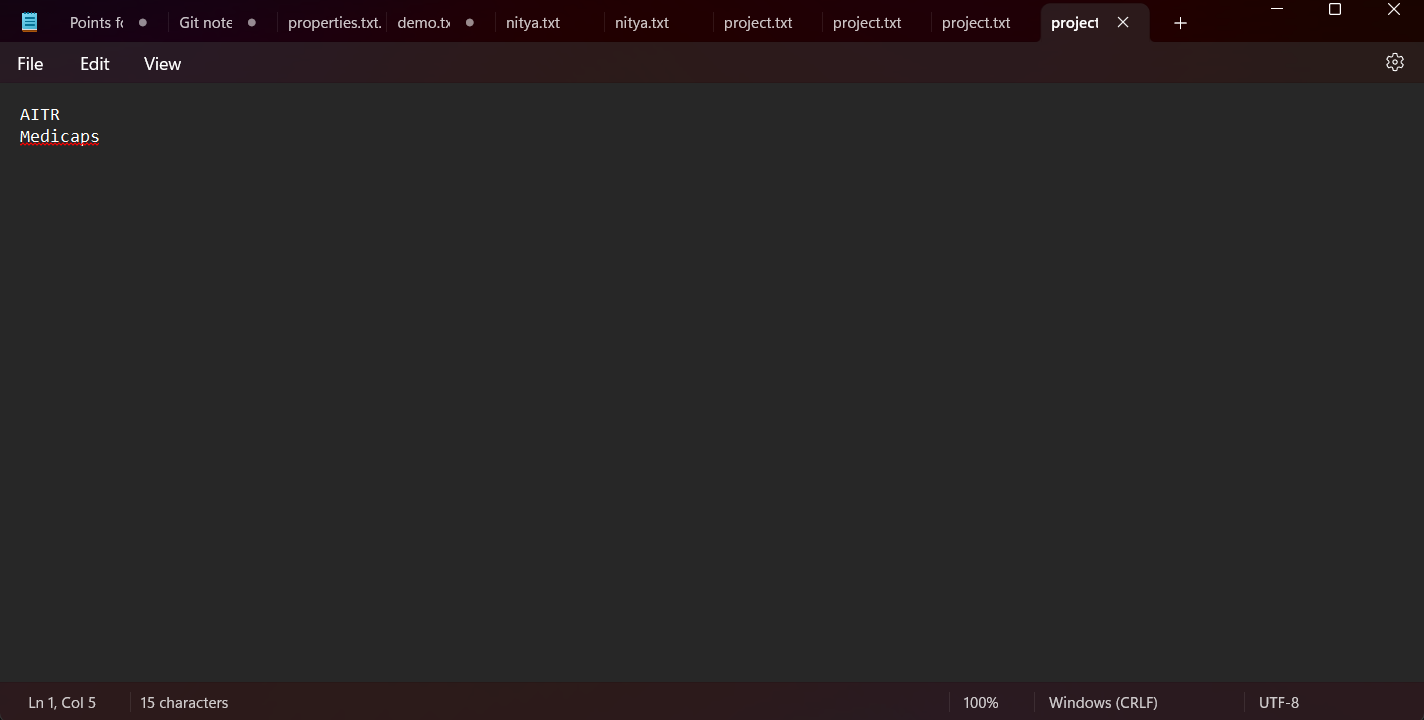


We will resolve the conflict by editing the file to include both changes,

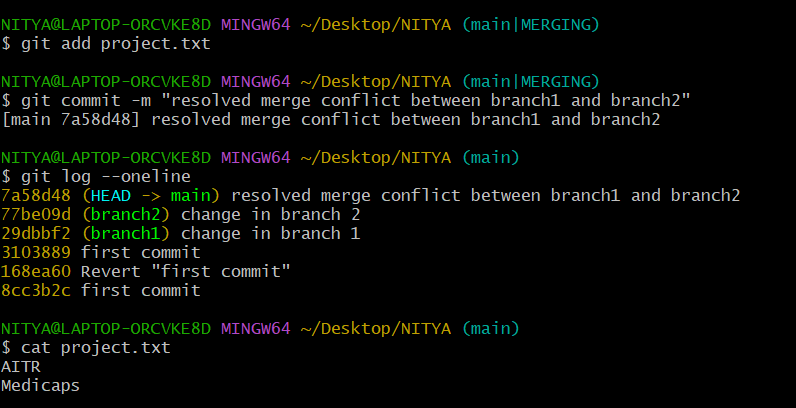
Delete the conflict markers (<<<<<<< HEAD, =======, and >>>>>>> branch2).

Choose which changes to keep or combine them.

For example, you might resolve it to



Now we will add the resolved file and commit our changes



Summary

By following these steps, we created a repository, introduced conflicting changes in two branches, and successfully resolved the merge conflict by including changes from both branches. The final state of the repository reflects the changes made by both the teammate and the user.

**Scenario 2: Add Files and Commit Together**

**Task: You have worked on multiple files and want to add them and commit using a single command.**

**Steps:**

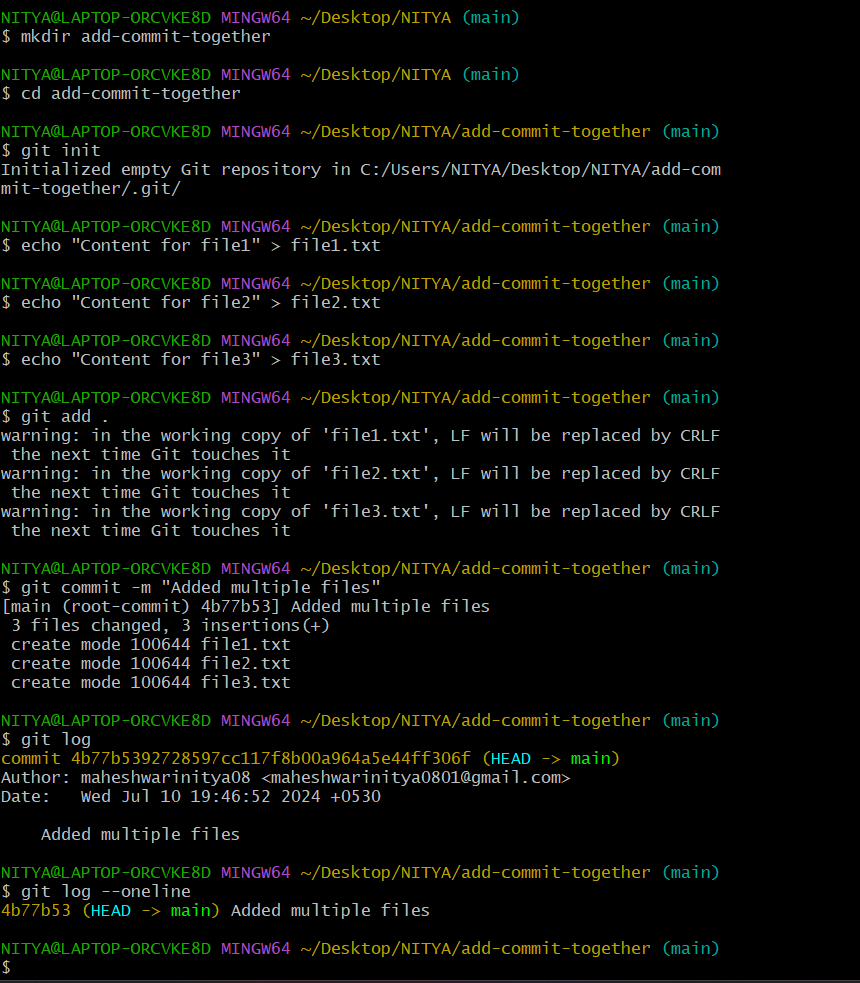
1. **Add all files to the staging area and commit them with a single command.**
2. **Verify the commit by checking the commit log.**

**Solution**

1. Create a new directory for the repository
2. Initialize the Git repository.
3. Create several files.
4. Add all files to the staging area and commit them with a single command:

* git add . stages all changed files in the current directory.
* git commit -m "Added multiple files" commits all staged changes with the specified commit message.

1. Check the commit log: This will display a list of commits with their hash and commit messages. You will see the commit message "Added multiple files" in the log.



**Scenario 3: Include Additional Commit in the Previous Commit Message**

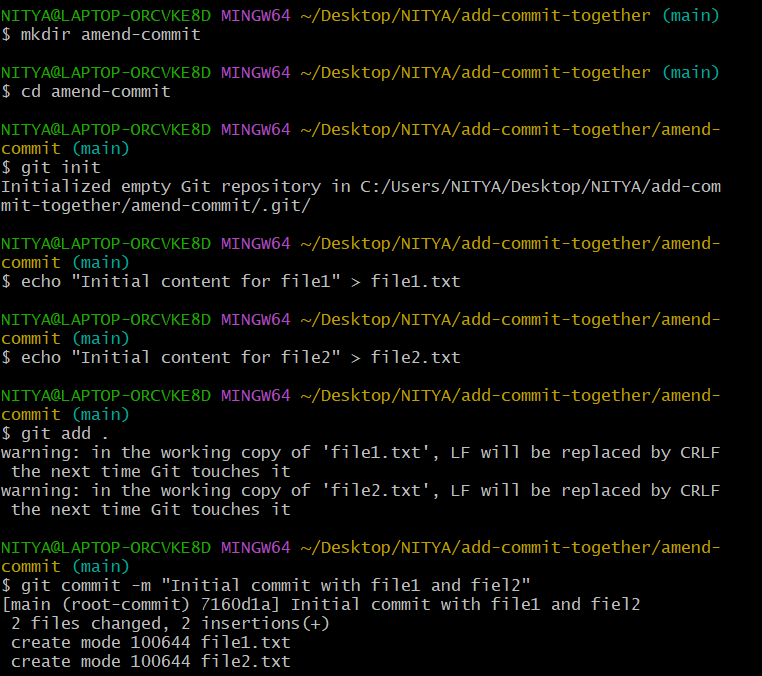
**Task: You have just made a commit but realize you forgot to include a file and make a small typo correction. Instead of creating a new commit, you can amend the last commit to include these changes without altering the commit message. Also verify the amended commit by checking the commit log.**

**Steps:**

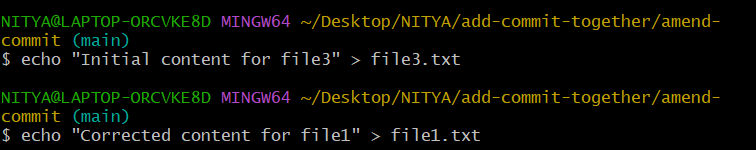
1. **Create and commit initial files.**
2. **Make additional changes.**
3. **Stage the changes and amend the last commit.**
4. **Verify the amended commit by checking the commit log.**

**Solution:**

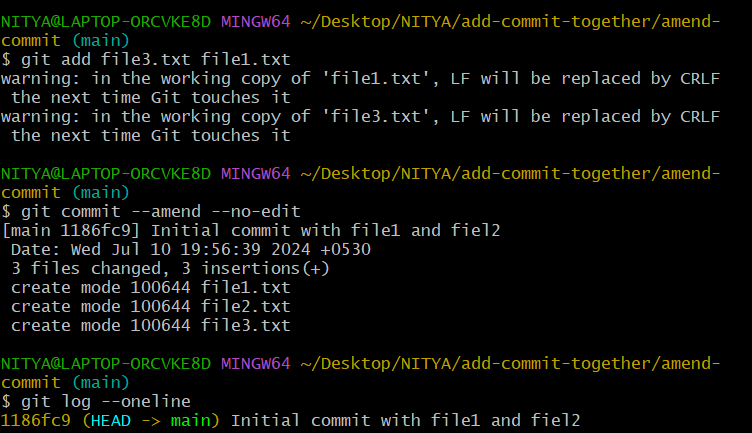
We will create a new directory, initialize it as a Git repository, create two text files, add them to the staging area, and commit them with a message.

****

Now we will create a new file (file3.txt) and update an existing file (file1.txt).



* git add file3.txt file1.txt: Stages the new file and the modified file.
* git commit --amend --no-edit: Amends the last commit to include these staged changes without changing the commit message.
* Git log --oneline : It will show a brief log of commits displaying the commit hash and commit message.



**Scenario 4: Stash Command**

**Task: You are working on a feature but need to quickly switch to another task without committing your current changes.**

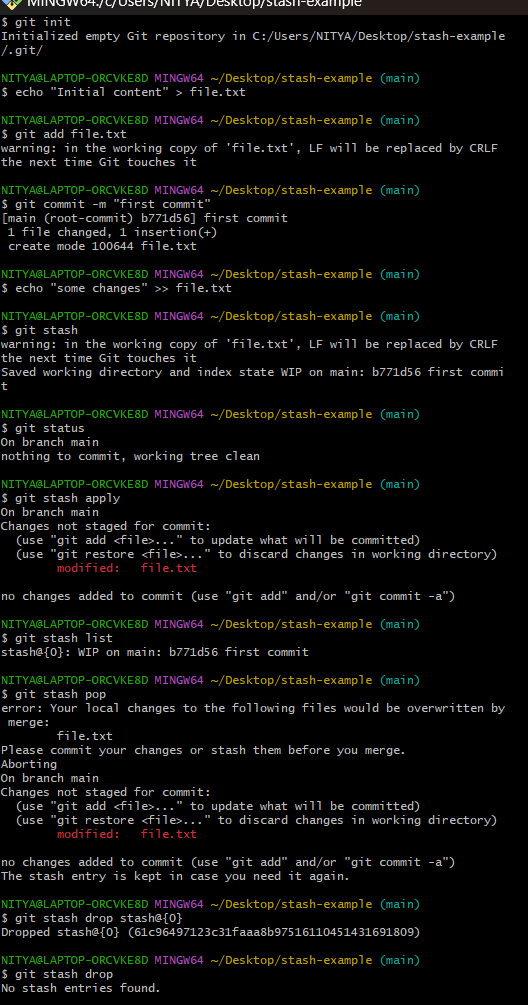
**Steps:**

1. **Make changes to a file but do not commit them.**
2. **Stash the changes.**
3. **Verify that your working directory is clean.**
4. **Apply the stashed changes back.**
5. **Experiment with other stash commands like stash list, stash pop, and stash drop.**

Using the Git stash command allows you to save your current work temporarily so you can switch to another task without committing your changes.

We will use the stash command to save and apply changes without committing them. This is useful when you need to switch tasks quickly without losing your current work

* Git stash: It saves your changes and leaves your working directory clean.
* git stash apply: re-applies the stashed changes but keeps the stash in the list.
* git stash pop: re-applies the stashed changes and removes the stash from the list.
* git stash drop: removes a stash from the list without applying it.



**Scenario 5: Use of .gitignore**

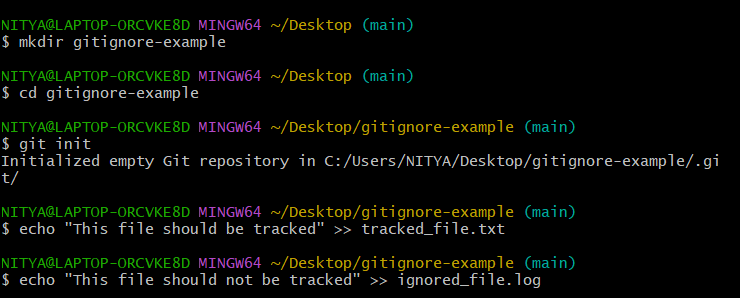
**Task: You have files in your project that should not be tracked by Git, such as log files or build artifacts.**

**Steps:**

1. **Create a new repository with files that should and should not be tracked (tracked\_file.txt, ignored\_file.log).**
2. **Create a .gitignore file and add patterns to exclude the appropriate files.**
3. **Verify that the ignored files are not tracked by Git.**
4. **Explain real-life use cases of .gitignore. What type of files are included in it?**

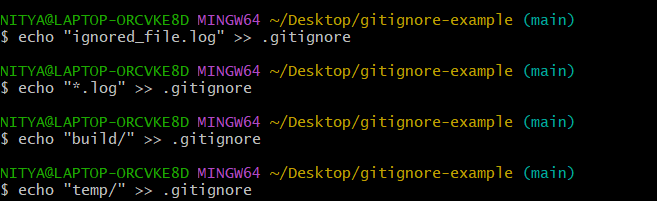
Using a .gitignore file in your Git repository allows you to specify files and directories that Git should ignore and not track. This is useful for excluding files that are not relevant to the version history, such as logs, build artifacts, temporary files, and more.

**Step 1: Set Up the Repository and Create Files**

****

**Step 2: Create a .gitignore File**

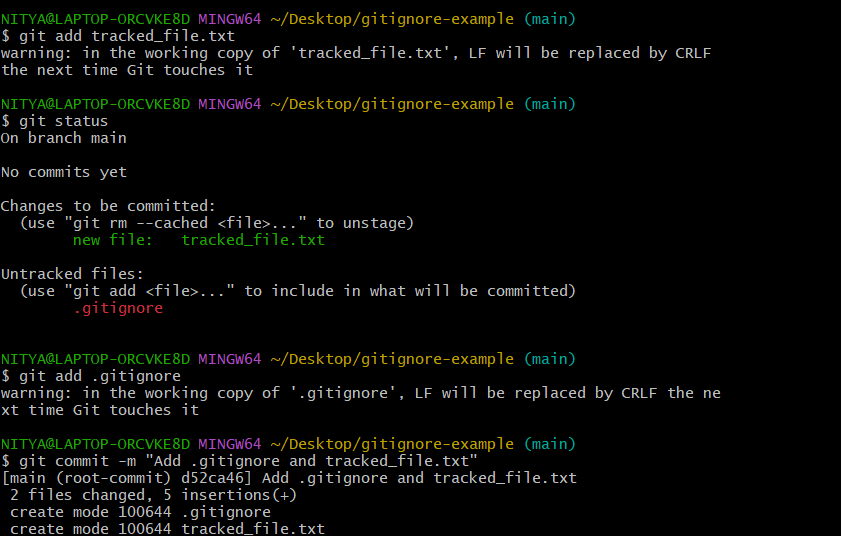
1. **Create a .gitignore file and add patterns to exclude the appropriate files**

****

This .gitignore file will ignore:

* All files with the .log extension.
* The build directory and its contents.
* The temp directory and its contents.

**Step 3: Verify that the Ignored Files are Not Tracked by Git**

****

**Real-Life Use Cases of .gitignore**

A .gitignore file is used to exclude files and directories that are not relevant to the version control system. Here are some common types of files typically included in a .gitignore:

1. **Log Files**:
   * \*.log
   * logs/
2. **Build Artifacts**:
   * build/
   * dist/
3. **Temporary Files**:
   * \*.tmp
   * \*.temp
4. **Operating System Files**:
   * .DS\_Store (macOS)
   * Thumbs.db (Windows)
5. **Dependency Directories**:
   * node\_modules/ (for Node.js projects)
   * vendor/ (for PHP projects using Composer)
6. **Environment Files**:
   * .env
7. **IDE/Editor Specific Files**:
   * .vscode/
   * .idea/
   * \*.suo
   * \*.user

**Scenario 6: Revert to Previous Commits**

**Task: You made a mistake in a recent commit and need to revert the changes.**

**Steps:**

**Create a file and commit changes to the file.**

**Make another commit with a mistake.**

**Revert to the previous commit, undoing the mistake.**

**Verify the state of the repository to ensure the revert was successful.**

**Solution:**

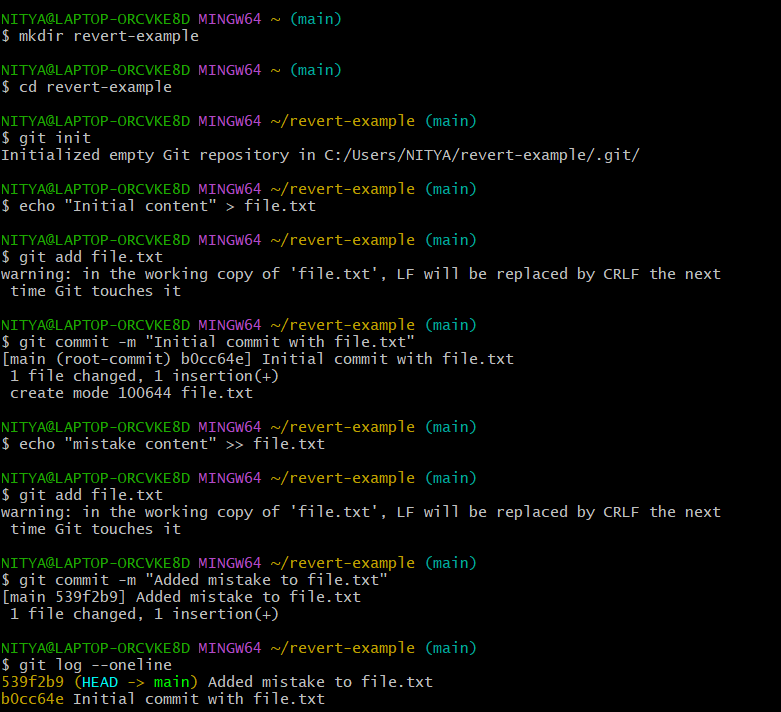
Initialize a new Git repository with mkdir revert-example, move to that directory: cd revert-example & initialize a git repository: git init.

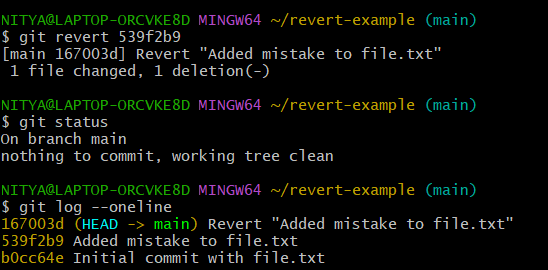
Create a file and make the initial commit with echo "Initial content" > file.txt, then stage it: git add file.txt & then commit it using: git commit -m "Initial commit with file.txt".

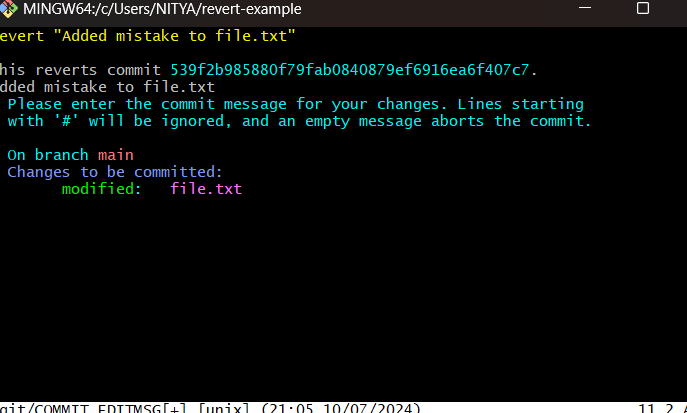
Make another commit with a mistake using echo "Mistake content" >> file.txt , add that: git add file.txt & then commit that: git commit -m "Added mistake to file.txt".

Revert the previous commit with git log --oneline to find the commit hash of the mistake, followed by git revert <commit-hash-of-mistake-commit>, where <commit-hash-of-mistake-commit> is replaced with the actual hash (e.g., git revert b7e7d0e).

Save and exit the text editor (in Vim: press Esc, type :wq, and press Enter). Finally, verify the state of the repository with git status && git log --oneline.







**Scenario 7: Create a Pull Request**

**Task: You have made changes on a feature branch and want to merge them into the main branch via a pull request.**

**Steps:**

1. **Create a new branch from the main branch and make changes.**
2. **Push the branch to the remote repository.**
3. **Create a pull request to merge your feature branch into the main branch.**
4. **Describe the changes in the pull request.**

I created a pull request to merge changes from new-feature-branch into main branch. This branch was created to develop a new feature, and it includes the addition of a new file named filenew.I initiated the pull request on GitHub. In the pull request description, then I provided the title and description. After the review process I will merge the pull request into main.

