**GIT BASIC – FILE STAGES**

Here’s a structured set of exercises to help you understand the different stages a file goes through in Git: **untracked**, **modified**, **staged**, and **committed**. These exercises will walk you through each stage and how files move between them.

**1. Untracked File Stage**

* **Objective**: Understand what an untracked file is and how to track it.
* **Steps**:
  1. Create a new Git repository in an empty folder:

mkdir git-stage-exercise

cd git-stage-exercise

git init

* 1. Create a new file:

echo "Hello, Git!" > hello.txt

* 1. Run git status to see that hello.txt is untracked.
  2. Move the file from untracked to tracked (staging) by using:

git add hello.txt

* 1. Run git status again to see that the file is now staged for commit.
  2. Finally, commit the file:

git commit -m "Added hello.txt"

**2. Modified File Stage**

* **Objective**: Understand how Git tracks changes to files and how a modified file differs from a staged file.
* **Steps**:
  1. Modify the hello.txt file:

echo "Another line!" >> hello.txt

* 1. Run git status to see that Git marks hello.txt as **modified**.
  2. Run git diff to see the specific changes that were made.
  3. Stage the changes:

git add hello.txt

* 1. Check git status again to see that the file is now staged.

**3. Staged File**

* **Objective**: Learn what happens to files when they are staged and how to unstage changes.
* **Steps**:
  1. With the file already modified and staged, run git status to confirm it's ready to be committed.
  2. Now unstage the file (move it back to the **modified** state):

git restore --staged hello.txt

* 1. Run git status again to see that the file is no longer staged but is still modified.
  2. Stage the changes again using git add and commit them:

git add hello.txt

git commit -m "Added another line to hello.txt"

**4. Committed File**

* **Objective**: Understand what happens when a file is committed and how to check the commit history.
* **Steps**:
  1. Run git log to see the history of commits.
  2. Open the hello.txt file again and make a new change:

echo "Third line!" >> hello.txt

* 1. Run git status to see that the file is marked as modified but not yet staged.
  2. Stage and commit the change:

git add hello.txt

git commit -m "Added third line to hello.txt"

* 1. Check git log again to see all the commits and how hello.txt has moved through the stages (untracked -> modified -> staged -> committed).

**5. Staging Part of a File**

* **Objective**: Learn how to stage only part of a file using Git.
* **Steps**:
  1. Modify the hello.txt file again, adding two new lines:

echo "Fourth line!" >> hello.txt

echo "Fifth line!" >> hello.txt

* 1. Run git diff to see the changes.
  2. Stage only part of the changes by using:

git add -p hello.txt

* 1. Git will present each change and ask if you want to stage it. Choose y to stage one part and n to leave another part unstaged.
  2. Run git status to see that only part of the file is staged for commit.
  3. Commit the staged change:

git commit -m "Added fourth line"

* 1. Stage the remaining change and commit it:

git add hello.txt

git commit -m "Added fifth line"

**6. Restoring a Modified File**

* **Objective**: Learn how to restore a modified file back to the last committed state.
* **Steps**:
  1. Modify the hello.txt file again:

echo "This line will be discarded!" >> hello.txt

* 1. Run git status to see the file marked as modified.
  2. Restore the file to its last committed state (discard the changes):

git restore hello.txt

* 1. Check git status again to confirm that the file is back to its committed state and no longer modified.

**7. Staging and Unstaging Multiple Files**

* **Objective**: Learn how to stage and unstage multiple files at once.
* **Steps**:
  1. Create two new files:

touch file1.txt file2.txt

* 1. Add content to both files:

echo "Content for file1" > file1.txt

echo "Content for file2" > file2.txt

* 1. Run git status to see both files are untracked.
  2. Stage both files at once:

git add file1.txt file2.txt

* 1. Run git status to confirm both files are staged.
  2. Unstage both files:

git restore --staged file1.txt file2.txt

* 1. Finally, add the files back and commit them:

git add .

git commit -m "Added file1.txt and file2.txt"

**8. Checking Differences Between Stages**

* **Objective**: Learn how to view the differences between the working directory, staging area, and the last commit.
* **Steps**:
  1. Modify file1.txt by adding a new line:

echo "Another line for file1" >> file1.txt

* 1. Run git status to see the file is modified.
  2. Run git diff to see the changes that have not yet been staged.
  3. Stage the changes with git add file1.txt.
  4. Run git diff --staged to see the differences between the staging area and the last commit.
  5. Commit the staged changes:

git commit -m "Updated file1.txt"

**9. Removing a File**

* **Objective**: Understand how to remove a file from Git's tracking.
* **Steps**:
  1. Remove a file (e.g., file2.txt) from your project:

rm file2.txt

* 1. Run git status to see that Git recognizes the file has been deleted.
  2. Stage the removal of the file:

git add file2.txt

* 1. Commit the deletion:

git commit -m "Removed file2.txt"

**10. Ignoring Files with .gitignore**

* **Objective**: Learn how to ignore files so they don’t appear in Git’s staging area.
* **Steps**:
  1. Create a .gitignore file:

touch .gitignore

* 1. Add patterns to ignore certain files (e.g., to ignore all .log files and a file named secret.txt):

echo "\*.log" >> .gitignore

echo "secret.txt" >> .gitignore

* 1. Create the files:

touch error.log secret.txt

* 1. Run git status to see that these files are untracked.
  2. Add the .gitignore file itself to Git and commit it:

git add .gitignore

git commit -m "Added .gitignore to ignore log and secret files"

* 1. Run git status again to verify that the .log and secret.txt files are now ignored.

**Additional Notes:**

* **Use git status frequently** to understand the current state of your files.
* **Use git diff to check differences** between stages, like between the working directory and the staging area or between the staging area and the last commit.
* **The stages of a file** in Git are crucial for understanding how your changes move from untracked to committed. These exercises will help you internalize the process.

**GIT BASIC COMMANDS (LOCAL REPO ONLY)**

Here’s a structured set of exercises to help you learn and practice using git init, git commit, git branch, git switch, and git merge.

**1. Initialize a Repository with git init**

* **Objective**: Create a new Git repository and make your first commit.
* **Steps**:
  1. Create a new folder for your project: mkdir my\_project && cd my\_project.
  2. Run git init to initialize an empty Git repository.
  3. Create a file (e.g., touch README.md) and add some content to it.
  4. Run git status to see the status of your working directory.
  5. Add the file to the staging area: git add README.md.
  6. Commit the changes: git commit -m "Initial commit with README".
  7. Run git log to see the commit history.

**2. Committing Changes with git commit**

* **Objective**: Practice making and committing changes.
* **Steps**:
  1. Modify the README.md file by adding new content.
  2. Run git status to see the changes.
  3. Stage the changes with git add README.md.
  4. Commit the changes with a descriptive message: git commit -m "Added project description to README".
  5. Modify another file (or create a new one).
  6. Use git add . to stage all changes.
  7. Commit with git commit -m "General updates".

**3. Working with Branches using git branch**

* **Objective**: Create, list, and delete branches.
* **Steps**:
  1. Check the current branch with git branch.
  2. Create a new branch: git branch new-feature.
  3. List all branches using git branch to confirm the new branch was created.
  4. Switch to the new branch using git switch new-feature.
  5. Delete the branch (while on a different branch): git branch -d new-feature.

**4. Switching Between Branches with git switch**

* **Objective**: Practice switching between branches.
* **Steps**:
  1. Make sure you are on the main branch: git switch main.
  2. Create a new branch: git branch feature-xyz.
  3. Switch to the new branch: git switch feature-xyz.
  4. Create or modify a file and commit the changes on feature-xyz.
  5. Switch back to the main branch: git switch main.
  6. Make changes on the main branch and commit them.
  7. Switch back to feature-xyz to see the difference between branches.

**5. Merging Branches with git merge**

* **Objective**: Merge changes from one branch into another.
* **Steps**:
  1. Make sure you're on the main branch: git switch main.
  2. Create and switch to a new branch: git switch -c hotfix-123.
  3. Modify some files and commit the changes on hotfix-123.
  4. Switch back to the main branch: git switch main.
  5. Merge the hotfix-123 branch into the main branch using git merge hotfix-123.
  6. Check the changes with git log --oneline and git status to confirm the merge.
  7. Resolve any merge conflicts if necessary (modify conflicting files, git add, and commit).

**6. Handling Merge Conflicts**

* **Objective**: Learn how to resolve merge conflicts.
* **Steps**:
  1. On the main branch, create a file conflict.txt and add some content, then commit it.
  2. Create a new branch git switch -c feature-branch and modify the same conflict.txt file with different content, then commit the changes.
  3. Switch back to the main branch: git switch main.
  4. Modify conflict.txt in the main branch with different content and commit.
  5. Merge the feature-branch into main using git merge feature-branch.
  6. Git will report a conflict in conflict.txt. Open the file, resolve the conflict by choosing which changes to keep, then stage the file with git add conflict.txt.
  7. Complete the merge by committing the resolved conflict: git commit.

**7. Creating and Merging Feature Branches**

* **Objective**: Practice using multiple feature branches.
* **Steps**:
  1. Create a new branch: git switch -c feature1.
  2. Modify a file, commit the changes.
  3. Switch back to the main branch: git switch main.
  4. Create another branch: git switch -c feature2.
  5. Modify a different file and commit the changes.
  6. Switch back to main and merge both branches: git merge feature1, then git merge feature2.
  7. Verify the result using git log and check that both features were successfully integrated.

**8. Fast-forward Merge**

* **Objective**: Understand fast-forward merges.
* **Steps**:
  1. Make sure you are on the main branch.
  2. Create a new branch and switch to it: git switch -c fast-forward-branch.
  3. Modify a file and commit the changes.
  4. Switch back to main and merge the fast-forward-branch: git merge fast-forward-branch.
  5. Since there have been no changes on main since the branch was created, the merge will be fast-forwarded. Verify this with git log --oneline.

**9. Non Fast-forward Merge (No Fast-forward Option)**

* **Objective**: Force Git to create a merge commit even if a fast-forward merge is possible.
* **Steps**:
  1. Create and switch to a new branch: git switch -c no-ff-branch.
  2. Modify a file and commit the changes.
  3. Switch back to main.
  4. Merge the branch with the --no-ff option: git merge --no-ff no-ff-branch.
  5. Check git log to see the explicit merge commit created.

**10. Reverting a Merge**

* **Objective**: Learn how to revert a merge.
* **Steps**:
  1. Merge a branch into main following the steps from a previous exercise.
  2. After the merge, decide to undo it: git revert -m 1 <commit-hash> (replace <commit-hash> with the hash of the merge commit).
  3. Git will create a new commit that reverses the changes from the merge.
  4. Check the log with git log to verify that the merge has been reverted.

**Additional Tips:**

* **Use git log --graph --oneline --all** to visualize the commit history with branches and merges.
* Practice regularly with different branch and merge scenarios to get comfortable handling both simple and complex workflows.

These exercises will help you become proficient in using git init, git commit, git branch, git switch, and git merge in a variety of practical scenarios.

**GIT BASIC (REMOTE)**

Here’s a structured set of exercises to help you learn and practice using git clone, git pull, git fetch, and git push.

**1. Cloning a Repository with git clone**

* **Objective**: Learn how to clone a remote repository.
* **Steps**:
  1. Find or create a repository on GitHub or another Git hosting platform (or use a public repository for practice).
  2. Run git clone <repository-url> to clone the repository to your local machine.
  3. Navigate to the newly created directory with cd <repository-name>.
  4. Verify the cloned repository’s status with git status.
  5. Check the remote information using git remote -v.

**2. Pulling Changes from a Remote Repository with git pull**

* **Objective**: Learn how to pull changes from a remote repository.
* **Steps**:
  1. Make sure your cloned repository is up to date by running git pull.
  2. Verify that the repository is updated by running git log --oneline and comparing it to the remote repository.
  3. (Optional) On the remote repository, make some changes or have someone else commit to the remote repository.
  4. Run git pull again to fetch and merge those new changes into your local repository.
  5. Check the updated files and commit log to see the pulled changes.

**3. Fetching Changes without Merging with git fetch**

* **Objective**: Understand the difference between git pull and git fetch.
* **Steps**:
  1. Ensure your repository is behind the remote by having someone else or yourself make changes and commit them to the remote repository.
  2. Run git fetch to fetch the changes from the remote without merging them into your working directory.
  3. Run git log --oneline origin/main (assuming main is your main branch) to see the fetched changes from the remote.
  4. Compare it with git log --oneline (without specifying origin/main) to see the difference between your local branch and the remote.
  5. Run git pull to merge the fetched changes into your local branch.

**4. Pushing Local Changes to a Remote Repository with git push**

* **Objective**: Push local changes to a remote repository.
* **Steps**:
  1. Modify a file in your cloned repository (e.g., edit the README.md or add a new file).
  2. Stage the changes: git add <filename>.
  3. Commit the changes: git commit -m "Updated README with new section".
  4. Push the changes to the remote repository: git push origin main (assuming main is your branch).
  5. Verify the changes on the remote repository (e.g., by checking on GitHub).
  6. Practice pushing to different branches by creating a new branch (git switch -c feature-branch), making changes, and pushing to origin/feature-branch.

**5. Simulating Collaborating with git pull and git push**

* **Objective**: Practice working in a team where multiple people are making changes.
* **Steps**:
  1. Clone a repository and have a collaborator (or yourself on another machine or local copy) clone the same repository.
  2. Make changes in the repository (e.g., add a new file or update an existing file) and commit the changes.
  3. Push the changes to the remote repository: git push origin main.
  4. On the collaborator's machine, try pushing changes without pulling first. Git will give an error.
  5. Run git pull on the collaborator's machine to pull the latest changes and merge them with their local copy.
  6. After resolving any conflicts (if applicable), push the changes: git push origin main.

**6. Handling Merge Conflicts after a git pull**

* **Objective**: Learn to resolve conflicts when pulling changes from a remote repository.
* **Steps**:
  1. Clone the repository twice into two separate directories (e.g., repo1 and repo2).
  2. In repo1, modify a file (e.g., README.md), commit the change, and push it to the remote repository.
  3. In repo2, modify the same file in a different way, commit the changes, and try to push them: git push origin main.
  4. You will get a push rejection, so run git pull origin main to pull and merge the remote changes.
  5. Git will indicate a merge conflict. Open the conflicting file, manually resolve the conflict, and mark it as resolved by staging the file (git add).
  6. Complete the merge by committing the resolution: git commit.
  7. Push the merged changes: git push origin main.

**7. Exploring git fetch with Branch Tracking**

* **Objective**: Understand how to fetch changes from a remote branch without merging them.
* **Steps**:
  1. Clone a repository and switch to a new branch: git switch -c feature-branch.
  2. Push the new branch to the remote: git push -u origin feature-branch.
  3. On another local copy or by collaborating with another person, create a different branch (e.g., git switch -c another-feature) and push it to the remote.
  4. In your original repository, run git fetch to fetch all branches and changes from the remote.
  5. Run git branch -r to see all the remote branches.
  6. Switch to the newly fetched branch with git switch another-feature.
  7. Inspect the fetched changes without merging them into the current branch.

**8. Tracking and Fetching Remote Changes without a git pull**

* **Objective**: Use git fetch and merge changes manually.
* **Steps**:
  1. Clone a repository and work on the main branch.
  2. Make sure changes are made in the remote repository by another collaborator.
  3. Run git fetch to fetch the changes without merging them.
  4. Run git log --oneline origin/main to view the remote commits.
  5. Run git merge origin/main to manually merge the fetched changes into your local branch.
  6. Verify the result by running git log and inspecting the commit history.

**9. Pushing to a New Remote Branch with git push**

* **Objective**: Push changes to a new remote branch.
* **Steps**:
  1. Create a new branch: git switch -c experimental-feature.
  2. Make some changes, commit them, and push the branch to the remote: git push -u origin experimental-feature.
  3. Check the remote repository (e.g., on GitHub) to verify that the branch was created and your changes were pushed.
  4. Switch back to the main branch: git switch main.
  5. Pull the latest changes from the remote: git pull.

**10. Deleting a Remote Branch with git push**

* **Objective**: Learn how to delete a remote branch.
* **Steps**:
  1. Ensure that you have a remote branch that you no longer need (e.g., feature-xyz).
  2. Run git push origin --delete feature-xyz to delete the branch from the remote repository.
  3. Run git fetch --prune to clean up the deleted remote branch from your local repository.
  4. Verify that the branch is no longer available remotely by checking on GitHub or using git branch -r.

**Additional Tips:**

* **Always run git status before and after running any Git commands** to monitor the state of your repository.
* **Use git log --oneline --graph --all** to visualize the branching and merging history of your repository.
* **Check git push --dry-run** to simulate a push operation without actually sending any changes, useful for testing commands.

These exercises will help you become proficient in using git clone, git pull, git fetch, and git push in a variety of scenarios, particularly when collaborating on projects or managing remote repositories.

**GIT CLONE (OPTIONS)**

Git clone is designed to download the entire history. In some cases (for example a clone done to execute a step on a pipeline) this would be time consuming (and resource consuming) without adding value. o clone a repository without downloading the full history, you can use the git clone command with a depth argument to create a **shallow clone**. This way, you only download the latest snapshot of the project without the full commit history.

Here’s how you can achieve that:

**Step 1: Clone with a shallow copy (latest status only)**

You can use the --depth option to specify the depth of the clone. For example, to get only the latest commit:

git clone --depth 1 <repository-url>

This will download only the latest snapshot of the repository, ignoring all previous history.

**Step 2: Download the full history later**

I

f you decide later that you need the full history of the repository, you can convert the shallow clone into a full clone by fetching the remaining history:

1. Navigate to the directory of the shallow clone:

cd <repository-folder>

1. Fetch all the commits and convert the repository into a full clone:

git fetch --unshallow

Alternatively, if you only want to fetch a specific number of additional commits, you can specify a new depth:

git fetch --depth <new-depth>

For example, git fetch --depth 10 would fetch the last 10 commits.

**Summary:**

* To clone only the latest commit (shallow clone):  
  git clone --depth 1 <repository-url>
* To convert the shallow clone to a full clone:  
  git fetch --unshallow  
  or  
  git fetch --depth <new-depth> to fetch a specific number of additional commits.

**GIT STASH**

Here’s a structured set of exercises to help you learn and practice using the git stash command effectively:

**1. Basic Stash**

* **Objective**: Learn how to stash changes and retrieve them.
* **Steps**:
  1. Make some changes to a file in your Git repository (e.g., edit a README or another file).
  2. Run git status to see the modified files.
  3. Run git stash to save your changes temporarily.
  4. Run git status again to confirm the working directory is clean.
  5. Use git stash list to see your stashed changes.
  6. Run git stash apply to re-apply the changes from the stash.
  7. Check git status to verify the changes have been restored.

**2. Stash with a Message**

* **Objective**: Learn how to add a message to your stash for better tracking.
* **Steps**:
  1. Modify a file in your project.
  2. Run git stash save "Updated README for new feature" to stash the changes with a message.
  3. Run git stash list to see the message.
  4. Use git stash apply to apply the changes back.

**3. Stashing Untracked Files**

* **Objective**: Understand how to stash untracked files.
* **Steps**:
  1. Create a new file (e.g., new\_file.txt) and do not commit it.
  2. Run git stash and observe that the untracked file is not stashed.
  3. Now run git stash --include-untracked.
  4. Check git status to see that the untracked file is now also stashed.
  5. Use git stash apply to restore both tracked and untracked files.

**4. Apply vs Pop**

* **Objective**: Understand the difference between git stash apply and git stash pop.
* **Steps**:
  1. Make changes to a file and stash them with git stash.
  2. Use git stash apply to bring the changes back, but leave the stash intact.
  3. Check git stash list to confirm the stash is still there.
  4. Make another change, stash it again.
  5. Use git stash pop to bring back the changes and remove the stash at the same time.
  6. Run git stash list to confirm the stash is no longer present.

**5. Dropping a Stash**

* **Objective**: Learn how to remove a specific stash.
* **Steps**:
  1. Make and stash some changes.
  2. Use git stash save "Some temporary changes" to stash another set of changes with a message.
  3. Check the stash list with git stash list.
  4. Use git stash drop stash@{0} (or whichever stash you want to remove).
  5. Confirm the stash is gone by running git stash list again.

**6. Stash Branch**

* **Objective**: Create a new branch from your stashed changes.
* **Steps**:
  1. Make changes to your working directory and stash them with git stash.
  2. Run git stash branch <branch-name> to create a new branch and apply the stashed changes to it.
  3. Verify that the new branch has your changes.
  4. Run git stash list to confirm the stash has been removed.

**7. Stashing Specific Files**

* **Objective**: Stash changes to specific files.
* **Steps**:
  1. Modify two files in your repository.
  2. Stash changes to only one file using git stash push -m "Message" <filename>.
  3. Use git stash list to see that the stash was created with only one file.
  4. Apply the stash using git stash apply.

**8. Clearing All Stashes**

* **Objective**: Remove all stashes.
* **Steps**:
  1. Create multiple stashes with various changes.
  2. Run git stash list to see all the stashes.
  3. Run git stash clear to remove all stashes.
  4. Run git stash list again to confirm the stash is empty.

By following these exercises, you'll get hands-on practice with the different ways git stash can be used to manage your changes effectively.

**GIT CONFIG**

Here’s a structured set of exercises to help you learn and practice using git config, which is used to configure Git's settings both globally and per project.

**1. Set Your Username and Email**

* **Objective**: Learn how to set and check your username and email, which will be associated with your commits.
* **Steps**:
  1. Run the following commands to set your name and email globally (for all repositories):

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

git config --global user.email [your.email@example.com](mailto:your.email@example.com)

* 1. Verify the configuration using:

git config --global user.name

git config --global user.email

* 1. Optionally, override these settings for a specific repository:

git config user.name "Repo-Specific Name"

git config user.email [repo-specific.email@example.com](mailto:repo-specific.email@example.com)

* 1. Check the local repository settings by running:

git config user.name

git config user.email

**2. Configure Default Text Editor**

* **Objective**: Learn how to set a default text editor for Git commit messages and other Git commands.
* **Steps**:
  1. Set the default text editor globally:
     + For Vim:

git config --global core.editor "vim"

* + - For Nano:

git config --global core.editor "nano"

* + - For Visual Studio Code:

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

* 1. Test the configuration by running git commit without the -m option and see if it opens in your configured editor.
  2. To check your current editor, run:

git config --global core.editor

**3. Configure Aliases**

* **Objective**: Learn how to create custom Git command aliases.
* **Steps**:
  1. Set up an alias for git status as git st:

git config --global alias.st status

* 1. Set up an alias for git log with a simplified format:

git config --global alias.lg "log --oneline --graph --all"

* 1. Test the aliases by running:

git st

git lg

* 1. Check the alias configuration with:

git config --global alias.st

git config --global alias.lg

* 1. Remove an alias if you no longer need it:

git config --global --unset alias.st

**4. View All Configurations**

* **Objective**: Understand how to view all Git configuration settings.
* **Steps**:
  1. Run the following command to view all global Git settings:

git config --global –list

* 1. View the local repository-specific settings (while inside a Git repository):

git config –list

* 1. View system-level settings (these are rarely modified but are good to know about):

git config --system –list

**5. Change Default Merge and Diff Tool**

* **Objective**: Learn how to configure Git to use a specific merge or diff tool.
* **Steps**:
  1. Set a diff tool (e.g., Vimdiff):

git config --global diff.tool vimdiff

git config --global difftool.prompt false

* 1. Set a merge tool (e.g., Vimdiff):

git config --global merge.tool vimdiff

* 1. Test the configuration by running a diff and merge command:
     + Make some changes to your repository and compare them with git difftool.
     + Create a conflict between two branches, then run git mergetool to resolve the conflict.
  2. To check the configuration, run:

git config --global diff.tool

git config --global merge.tool

**6. Setting the Default Branch Name**

* **Objective**: Learn how to set the default branch name when initializing a new repository.
* **Steps**:
  1. Set the default branch name (e.g., main) globally:

git config --global init.defaultBranch main

* 1. Create a new repository and verify the default branch is set to main:

git init my\_new\_repo

cd my\_new\_repo

git branch

* 1. To check the current default branch setting, run:

git config --global init.defaultBranch

**7. Configuring Line Endings (CRLF vs LF)**

* **Objective**: Learn how to manage line endings between different operating systems.
* **Steps**:
  1. Set Git to automatically handle line endings for Windows and Unix systems:

git config --global core.autocrlf true # On Windows

git config --global core.autocrlf input # On Linux/macOS

* 1. Verify the configuration:

git config --global core.autocrlf

**8. Color Configuration**

* **Objective**: Learn how to configure Git to show colored output for different commands.
* **Steps**:
  1. Enable colored output globally:

git config --global color.ui auto

* 1. Enable or disable color for specific commands (e.g., status, branch, diff):

git config --global color.status auto

git config --global color.branch auto

git config --global color.diff auto

* 1. Test the configuration by running:

git status

git branch

git diff

**9. Configure GPG Signatures for Commits – OPTIONAL EXERCISE**

* **Objective**: Learn how to sign commits using GPG (GNU Privacy Guard).
* **Steps**:
  1. Generate a GPG key if you don't already have one:

gpg --gen-key

* 1. Set Git to use your GPG key for signing commits:

git config --global user.signingkey <your-gpg-key-id>

git config --global commit.gpgSign true

* 1. Create and sign a commit:

git commit -S -m "Signed commit"

* 1. Verify the commit signature:

git log --show-signature

**10. Set Default Push Behavior**

* **Objective**: Learn how to configure Git's behavior when pushing changes.
* **Steps**:
  1. Configure Git to push the current branch only (safer option):

git config --global push.default current

* 1. Alternatively, configure Git to push all branches:

git config --global push.default matching

* 1. Test the configuration by making a change, committing it, and running git push without specifying a branch.
  2. To check the current push behavior, run:

git config --global push.default

**11. Use Conditional Git Configurations**

* **Objective**: Learn how to set conditional configurations based on different directories or environments.
* **Steps**:
  1. Add conditional configuration to use a specific email for a certain directory:

git config --global includeIf."gitdir:/path/to/repo/".user.email [specific.email@example.com](mailto:specific.email@example.com)

* 1. Test it by navigating to /path/to/repo and checking the configured email:

git config user.email

**Additional Tips:**

* Use git config --help to learn more about the options available.
* If you ever need to reset your Git configuration, simply edit or delete the .gitconfig file located in your home directory (~/.gitconfig for global settings).

**GIT DIFF**

Here is a set of exercises to help you understand and practice using the git diff command. These exercises will cover comparing differences in working directories, staged areas, and across commits.

**1. Basic git diff in the Working Directory**

* **Objective**: Learn how to view changes in your working directory.
* **Steps**:
  1. Create a new Git repository and add a file:

git init diff-exercise

cd diff-exercise

echo "Line 1" > file.txt

git add file.txt

git commit -m "Initial commit with file.txt"

* 1. Modify the file.txt file by adding new content:

echo "Line 2" >> file.txt

* 1. Use git diff to see the changes between your working directory and the last commit:

git diff

* 1. Analyze the output to see what lines were added or changed.
  2. Modify the file again by adding another line and run git diff again to see the differences.

**2. Viewing Staged vs Unstaged Changes**

* **Objective**: Understand the difference between changes in the working directory and the staging area.
* **Steps**:
  1. Add a new line to file.txt:

echo "Line 3" >> file.txt

* 1. Stage the file with git add:

git add file.txt

* 1. Run git diff to see the changes in the working directory (since everything is staged, there should be no output).
  2. Use git diff --staged (or git diff --cached) to see the changes between the staged area and the last commit:

git diff --staged

* 1. Modify the file again and run both git diff and git diff --staged to see the difference between unstaged changes and staged changes.

**3. Comparing Specific Files**

* **Objective**: Learn how to compare changes for a specific file.
* **Steps**:
  1. Modify multiple files in the repository:

echo "Change in file1" > file1.txt

echo "Change in file2" > file2.txt

* 1. Run git diff to see changes across all modified files.
  2. Now, focus on the changes in a specific file by running:

git diff file1.txt

* 1. Compare the staged changes for a specific file by running:

git diff --staged file2.txt

**4. Comparing Across Commits**

* **Objective**: Learn how to compare changes between different commits.
* **Steps**:
  1. Make multiple commits:

echo "Line 4" >> file.txt

git add file.txt

git commit -m "Added Line 4"

echo "Line 5" >> file.txt

git add file.txt

git commit -m "Added Line 5"

* 1. Use git log --oneline to find the commit hashes of the last few commits.
  2. Compare the differences between two commits:

git diff <commit1> <commit2>

Replace <commit1> and <commit2> with actual commit hashes from the log.

* 1. Compare the working directory with an earlier commit:

git diff <commit-hash>

**5. Comparing a Branch with Another Branch**

* **Objective**: Compare changes between two branches.
* **Steps**:
  1. Create a new branch:

git switch -c feature-branch

echo "Feature change" >> file.txt

git add file.txt

git commit -m "Feature change on feature-branch"

* 1. Switch back to the main branch:

git switch main

* 1. Compare the changes between main and feature-branch:

git diff feature-branch

* 1. You can also compare the two branches explicitly:

git diff main feature-branch

**6. Comparing Commits on a Specific File**

* **Objective**: Compare the changes to a specific file between different commits.
* **Steps**:
  1. Modify file.txt again:

echo "Another line on file.txt" >> file.txt

git add file.txt

git commit -m "Another change to file.txt"

* 1. Use git log --oneline file.txt to see the commit history for file.txt.
  2. Pick two commit hashes and compare the changes made to file.txt between them:

git diff <commit1> <commit2> file.txt

**7. Viewing Word Differences**

* **Objective**: Learn how to view word-level differences instead of line-level differences.
* **Steps**:
  1. Modify file.txt by editing a few words:

echo "This is a sentence with some changes." > file.txt

git add file.txt

git commit -m "Modified sentence in file.txt"

echo "This is a sentence with major changes." > file.txt

* 1. Use git diff --word-diff to see word-level differences:

git diff --word-diff

* 1. Experiment with modifying other parts of the file and using --word-diff to see more granular differences.

**8. Comparing Files with a Specific Branch**

* **Objective**: Compare a file with its version in another branch.
* **Steps**:
  1. Switch to the main branch:

git switch main

* 1. Modify file.txt on the main branch.
  2. Compare file.txt on the current branch (main) with the version in the feature-branch:

git diff feature-branch file.txt

**9. Using git diff for Historical Changes**

* **Objective**: See how to compare a file's state from a previous commit to its current state.
* **Steps**:
  1. Use git log --oneline to find a previous commit hash where file.txt was modified.
  2. Compare the current state of file.txt with its state in a previous commit:

git diff <commit-hash> file.txt

* 1. Analyze the differences and use git log to understand what changed between those points in time.

**10. Comparing Working Directory with Staging Area**

* **Objective**: Understand how to view differences between the working directory and the staging area.
* **Steps**:
  1. Modify file.txt by adding a new line:

echo "Unstaged change" >> file.txt

* 1. Stage the changes:

git add file.txt

* 1. Modify file.txt again without staging it:

echo "Unstaged change 2" >> file.txt

* 1. Use git diff to see the difference between the working directory and the staging area:

git diff

* 1. Use git diff --staged to see the difference between the staged changes and the last commit:

git diff --staged

**Additional Tips:**

* **Use git log --oneline --graph** to visualize the commit history before using git diff to compare commits or branches.
* **Use git diff --stat** to see a summary of changes, including how many lines were added or deleted in each file.
* **Combine git diff with grep** to filter specific types of changes (e.g., searching for changes related to a specific function or word).

**GIT CHECKOUT ON TAG**

Let’s walk through the entire process, including how to add a tag to a commit, create new commits, and then use git checkout to create a new branch from that tagged commit. Here's the detailed example.

**Steps:**

**1. Initialize a New Git Repository**

First, create a new Git repository to simulate this process.

mkdir git-hotfix-example

cd git-hotfix-example

git init

**2. Create and Commit a File**

Create a file and make the initial commit.

echo "Version 1.0 Initial content" > file.txt

git add file.txt

git commit -m "Initial commit for version 1.0"

**3. Tag the Commit**

Now, assign a tag to this commit to mark it as v1.0.

git tag v1.0

This tag points to the commit you just made. You can verify the tag by running:

git show v1.0

This will display the commit information and the tag.

**4. Make New Changes and Commits**

Next, simulate new development by making some changes and adding new commits.

echo "New feature for version 2.0" >> file.txt

git add file.txt

git commit -m "Added new feature for version 2.0"

echo "Another new feature" >> file.txt

git add file.txt

git commit -m "Added another new feature"

At this point, your main branch has moved forward with new commits, but the v1.0 tag still points to the initial commit.

**5. Checkout the Tagged Commit to Create a Hotfix Branch**

Now, you need to create a hotfix for version 1.0. To do that, you will create a branch from the v1.0 tag.

git checkout -b hotfix-v1.0 v1.0

Here’s what this command does:

* **-b hotfix-v1.0**: Creates a new branch named hotfix-v1.0.
* **v1.0**: Specifies that this branch should be created from the commit tagged with v1.0.

You are now in a new branch, starting from the v1.0 commit, not the latest commits on main.

**6. Apply the Hotfix**

Make the necessary hotfix changes and commit them to this new branch.

echo "Hotfix for version 1.0" >> file.txt

git add file.txt

git commit -m "Applied hotfix to version 1.0"

**7. Push the Hotfix Branch (Optional)**

If you want to share this hotfix branch with others, push it to the remote repository:

git push origin hotfix-v1.0

**8. Merge the Hotfix Back to main (Optional)**

Once the hotfix is ready, you may want to merge it back into the main branch (depending on your development workflow):

1. Switch to the main branch:

git checkout main

1. Merge the hotfix:

git merge hotfix-v1.0

1. Push the updated main branch:

git push origin main

**9. Tag the Hotfix (Optional)**

If the hotfix represents a new version, you can tag it. For example, if this is version v1.0.1, tag it as:

git tag v1.0.1

git push origin v1.0.1

This marks the new hotfix version with a tag for future reference.

**Summary of Commands:**

1. **Tag a commit:**

git tag v1.0

1. **Create new commits on main:**

echo "New feature for version 2.0" >> file.txt

git add file.txt

git commit -m "Added new feature for version 2.0"

1. **Create a hotfix branch from the v1.0 tag:**

git checkout -b hotfix-v1.0 v1.0

1. **Apply the hotfix and commit:**

echo "Hotfix for version 1.0" >> file.txt

git add file.txt

git commit -m "Applied hotfix to version 1.0"

1. **Push the hotfix branch (optional):**

git push origin hotfix-v1.0

1. **Merge the hotfix back to main (optional):**

git checkout main

git merge hotfix-v1.0

git push origin main

1. **Tag the hotfix (optional):**

git tag v1.0.1

git push origin v1.0.1

**GIT REBASE AND GIT REVERTS**

Here are a set of exercises to help you understand and practice using **git rebase** and **git revert**. These exercises will guide you through common scenarios where you would use these commands.

**Part 1: Understanding git rebase**

**1. Basic Rebase: Moving Commits**

* **Objective**: Learn how to rebase one branch onto another.
* **Steps**:
  1. Create a new Git repository:

git init rebase-exercise

cd rebase-exercise

* 1. Create a file and make a commit:

echo "First commit on main" > file.txt

git add file.txt

git commit -m "First commit on main"

* 1. Create a new branch feature-branch and make a commit on that branch:

git switch -c feature-branch

echo "Commit on feature-branch" >> file.txt

git add file.txt

git commit -m "Commit on feature-branch"

* 1. Switch back to main and make a new commit:

git switch main

echo "Second commit on main" >> file.txt

git add file.txt

git commit -m "Second commit on main"

* 1. Now, rebase feature-branch onto the updated main:

git switch feature-branch

git rebase main

* 1. Run git log --oneline --graph to visualize the new history after the rebase.
  2. Check the contents of file.txt to see that the changes from both branches have been applied.

**2. Interactive Rebase: Rewriting History**

* **Objective**: Learn how to rewrite commit messages and squash commits with interactive rebasing.
* **Steps**:
  1. Make a series of commits on feature-branch:

echo "Commit 1" >> file.txt

git add file.txt

git commit -m "First commit"

echo "Commit 2" >> file.txt

git add file.txt

git commit -m "Second commit"

echo "Commit 3" >> file.txt

git add file.txt

git commit -m "Third commit"

* 1. Use interactive rebase to combine the last three commits into one:

git rebase -i HEAD~3

* 1. In the text editor that opens, change the second and third commits from pick to squash (or s), leaving the first commit as pick.
  2. Save and close the editor. Git will now allow you to edit the commit message. Combine the messages or create a new message.
  3. Check the commit history with git log --oneline to see the result of squashing the commits.

**3. Rebase Conflict Resolution**

* **Objective**: Learn how to handle conflicts during a rebase.
* **Steps**:
  1. Go back to the main branch and make another change:

git switch main

echo "Conflicting line" >> file.txt

git add file.txt

git commit -m "Conflicting commit on main"

* 1. Switch back to feature-branch and make a conflicting change to the same file:

git switch feature-branch

echo "Different conflicting line" >> file.txt

git add file.txt

git commit -m "Conflicting commit on feature-branch"

* 1. Rebase the feature-branch onto main:

git rebase main

* 1. Git will stop and inform you of the conflict. Open file.txt and manually resolve the conflict.
  2. After resolving the conflict, add the file back to the staging area:

git add file.txt

* 1. Continue the rebase:

git rebase –continue

* 1. Check the commit history to ensure the rebase completed successfully.

**4. Abort a Rebase**

* **Objective**: Learn how to abort a rebase and return to the original branch state.
* **Steps**:
  1. Start a rebase as in the previous exercise, but before resolving the conflict, abort the rebase:

git rebase –abort

* 1. Check git log to see that the branch is back to its original state and the rebase has been cancelled.

**Part 2: Understanding git revert**

**5. Basic Revert: Undo a Commit**

* **Objective**: Learn how to revert a commit while keeping the commit history intact.
* **Steps**:
  1. On main, create a new commit:

echo "New feature" >> feature.txt

git add feature.txt

git commit -m "Added new feature"

* 1. Realize the commit was a mistake and revert it:

git revert HEAD

* 1. Run git log --oneline to see that a new commit was created to undo the previous commit.
  2. Check the file feature.txt to see that the changes have been undone.

**6. Reverting a Specific Commit**

* **Objective**: Learn how to revert a commit from further back in the history.
* **Steps**:
  1. Create several commits on main:

echo "Commit A" >> file.txt

git add file.txt

git commit -m "Commit A"

echo "Commit B" >> file.txt

git add file.txt

git commit -m "Commit B"

echo "Commit C" >> file.txt

git add file.txt

git commit -m "Commit C"

* 1. Revert commit B while leaving the others intact:

git log --oneline

Copy the hash of "Commit B" and then run:

bash

Copy code

git revert <hash-of-commit-B>

* 1. Check the commit history and file contents to verify that commit B was undone, but the history remains intact.

**7. Reverting a Merge Commit**

* **Objective**: Learn how to revert a merge commit.
* **Steps**:
  1. Create two branches feature-1 and feature-2 and make commits on each branch.
  2. Merge feature-1 into main:

git merge feature-1

* 1. Realize the merge was a mistake, and revert the merge:

git revert -m 1 <merge-commit-hash>

Here, -m 1 tells Git to use the first parent (the main branch) in the revert.

* 1. Check the commit history and verify that the merge was reverted.

**8. Reverting Multiple Commits**

* **Objective**: Learn how to revert a series of commits in one command.
* **Steps**:
  1. Make a series of commits on the main branch:

echo "Commit X" >> file.txt

git add file.txt

git commit -m "Commit X"

echo "Commit Y" >> file.txt

git add file.txt

git commit -m "Commit Y"

echo "Commit Z" >> file.txt

git add file.txt

git commit -m "Commit Z"

* 1. Revert multiple commits in one command:

git revert HEAD~2..HEAD

This will revert the last two commits (Y and Z) in one go.

* 1. Check the history to see that both commits have been reverted in a single commit.

**9. Revert with Conflict Resolution**

* **Objective**: Learn how to resolve conflicts during a revert.
* **Steps**:
  1. Make conflicting changes on two branches and merge them into main (as done in previous exercises).
  2. Revert the merge commit, which will trigger a conflict:

git revert -m 1 <merge-commit-hash>

* 1. Resolve the conflict manually by editing the conflicting files.
  2. Stage the resolved files and complete the revert:

git add .

git revert –continue

* 1. Verify the revert was successful and that the conflict was resolved.

**Additional Tips:**

* **Use git log --oneline --graph --all** to visualize the history and understand the changes made by rebase and revert.
* **Always practice on a test branch** when using rebase and revert, as they modify the commit history.

**CHERRY PICK**

**git cherry-pick** is a Git command that allows you to apply specific commits from one branch into another, without merging the entire branch. It’s useful when you want to pick a few specific commits from another branch and apply them to your current branch. Unlike git merge, which merges all changes, git cherry-pick only applies the selected commits.

Here’s a set of exercises to help you understand and practice using git cherry-pick.

**1. Basic Cherry-pick: Apply a Specific Commit**

**Objective:**

Learn how to apply a specific commit from one branch to another.

**Steps:**

1. **Set up the repository:**
   * Initialize a new Git repository:

git init cherry-pick-exercise

cd cherry-pick-exercise

* + Create a file and make an initial commit on the main branch:

echo "Initial content" > file.txt

git add file.txt

git commit -m "Initial commit on main"

* + Create a new branch feature-branch and make a new commit:

git switch -c feature-branch

echo "Feature commit 1" >> file.txt

git add file.txt

git commit -m "Feature commit 1"

1. **Make another commit on feature-branch:**

echo "Feature commit 2" >> file.txt

git add file.txt

git commit -m "Feature commit 2"

1. **Switch back to the main branch:**

git switch main

1. **Cherry-pick a commit from feature-branch:**
   * Use git log on feature-branch to get the hash of the commit you want to cherry-pick:

git log feature-branch --oneline

* + Copy the commit hash for "Feature commit 1".
  + Run the following command to cherry-pick "Feature commit 1" onto the main branch:

git cherry-pick <commit-hash>

1. **Verify the cherry-pick:**
   * Run git log --oneline to see that "Feature commit 1" has been applied to the main branch.
   * Check the content of file.txt to confirm the change has been applied.

**2. Cherry-pick Multiple Commits**

**Objective:**

Learn how to cherry-pick multiple commits at once.

**Steps:**

1. **Make additional commits on feature-branch:**
   * Switch back to feature-branch and make more commits:

git switch feature-branch

echo "Feature commit 3" >> file.txt

git add file.txt

git commit -m "Feature commit 3"

echo "Feature commit 4" >> file.txt

git add file.txt

git commit -m "Feature commit 4"

1. **Switch back to main branch:**

git switch main

1. **Cherry-pick multiple commits:**
   * Use git log feature-branch to get the commit hashes of the last two commits.
   * Cherry-pick both commits by specifying a range:

git cherry-pick <commit1>^..<commit2>

Replace <commit1> and <commit2> with the appropriate commit hashes (in this case, for "Feature commit 3" and "Feature commit 4").

1. **Verify the cherry-pick:**
   * Run git log --oneline to see that both commits have been applied to the main branch.
   * Check file.txt to confirm the changes.

**3. Cherry-pick with Conflicts**

**Objective:**

Learn how to resolve conflicts that may arise during cherry-picking.

**Steps:**

1. **Create a conflicting commit on main:**
   * Modify file.txt on the main branch to create a conflict:

echo "Conflicting line on main" >> file.txt

git add file.txt

git commit -m "Conflicting commit on main"

1. **Cherry-pick a conflicting commit from feature-branch:**
   * Switch back to main and cherry-pick the commit from feature-branch that modifies the same line in file.txt:

git cherry-pick <conflicting-commit-hash>

1. **Resolve the conflict:**
   * Git will stop the cherry-pick and show the conflict.
   * Open file.txt and manually resolve the conflict.
   * Once the conflict is resolved, add the file to the staging area:

git add file.txt

1. **Continue the cherry-pick:**

git cherry-pick –continue

1. **Verify the cherry-pick:**
   * Run git log --oneline to see that the cherry-pick was successful.
   * Check the contents of file.txt to confirm the conflict resolution.

**4. Abort a Cherry-pick**

**Objective:**

Learn how to abort a cherry-pick in progress if you decide not to apply the commit.

**Steps:**

1. **Start a cherry-pick that results in a conflict:**
   * Try to cherry-pick a commit from feature-branch that conflicts with the main branch (as in the previous exercise).
2. **Abort the cherry-pick:**
   * If you decide that you don’t want to proceed with the cherry-pick, use:

git cherry-pick –abort

1. **Verify the result:**
   * Check that the cherry-pick was aborted by running git log and verifying that no new commit was applied.
   * Check git status to confirm that the working directory is clean.

**5. Cherry-pick with Commit Message Editing**

**Objective:**

Learn how to edit the commit message while cherry-picking.

**Steps:**

1. **Cherry-pick a commit with a modified message:**
   * Pick any commit from feature-branch and use the -e option to edit the commit message during the cherry-pick:

git cherry-pick -e <commit-hash>

* + Git will open the default editor, allowing you to modify the commit message.

1. **Edit the message and save the commit:**
   * Modify the message as needed, save, and close the editor.
2. **Verify the result:**
   * Run git log --oneline to see the new commit on the main branch with the edited message.

**6. Cherry-pick a Commit from a Detached HEAD State**

**Objective:**

Learn how to cherry-pick a commit when in a detached HEAD state (useful when you're working with specific commits).

**Steps:**

1. **Switch to a specific commit on feature-branch (detached HEAD state):**
   * Use the git log command to find a commit hash from feature-branch.
   * Check out that specific commit:

git checkout <commit-hash>

1. **Cherry-pick the current commit to main:**
   * Even though you are in a detached HEAD state, you can cherry-pick the current commit to the main branch:

git switch main

git cherry-pick <commit-hash>

1. **Verify the result:**
   * Run git log --oneline to see that the commit was applied to main.

**7. Cherry-pick Multiple Non-Sequential Commits**

**Objective:**

Learn how to cherry-pick non-sequential commits from a branch.

**Steps:**

1. **Make several commits on feature-branch:**
   * Create multiple commits with different content on feature-branch:

echo "Feature commit A" >> file.txt

git add file.txt

git commit -m "Feature commit A"

echo "Feature commit B" >> file.txt

git add file.txt

git commit -m "Feature commit B"

echo "Feature commit C" >> file.txt

git add file.txt

git commit -m "Feature commit C"

1. **Switch to main branch:**

git switch main

1. **Cherry-pick specific non-sequential commits:**
   * Suppose you only want to cherry-pick commits A and C (but not B):

git cherry-pick <commit-hash-A>

git cherry-pick <commit-hash-C>

1. **Verify the result:**
   * Run git log --oneline to see that only commits A and C were applied to main.
   * Check file.txt to confirm the specific changes were applied.

**Summary of Commands:**

* **git cherry-pick <commit-hash>**: Apply a specific commit from another branch.
* **git cherry-pick <commit1>^..<commit2>**: Apply a range of commits.
* **git cherry-pick --continue**: Continue after resolving conflicts.
* **git cherry-pick --abort**: Abort the cherry-pick operation.

----------

Git doesn't allow you to directly cherry-pick individual files from a commit, as the git cherry-pick command works at the commit level, applying the entire commit. However, you can achieve a similar effect by cherry-picking the entire commit and then selectively staging only the file(s) you want.

Here’s how you can cherry-pick changes for specific files from a commit:

**Steps to Cherry-pick Specific Files from a Commit:**

**1. Start the Cherry-pick Operation**

* Cherry-pick the entire commit, but use the --no-commit option to stop Git from creating a commit immediately:

git cherry-pick <commit-hash> --no-commit

* This applies the changes from the specified commit to your working directory, but it doesn’t create a new commit yet.

**2. Reset Unwanted Changes**

* If the commit contains changes to other files that you don't want, you can reset them by using git restore to discard the unwanted changes:

git restore --staged <unwanted-file>

git restore <unwanted-file>

* This will remove those files from the staging area and revert them back to their original state in the working directory.

**3. Stage the File(s) You Want**

* Now, stage only the file(s) you want to include in your commit:

git add <file-you-want>

**4. Commit the Changes**

* Once you have staged only the desired file(s), you can create the commit:

git commit -m "Cherry-picked specific file(s) from commit <commit-hash>"

**Example**

Let’s say you have a commit abc1234 that modifies two files, file1.txt and file2.txt, but you only want to cherry-pick the changes made to file1.txt.

1. **Cherry-pick the commit without committing:**

git cherry-pick abc1234 --no-commit

1. **Unstage and discard changes from file2.txt:**

git restore --staged file2.txt

git restore file2.txt

1. **Stage only file1.txt:**

git add file1.txt

1. **Commit the changes:**

git commit -m "Cherry-picked changes to file1.txt from commit abc1234"

**Alternative: Using git checkout (for more control)**

Another way to achieve this is by using git checkout (or git restore in newer versions of Git) to apply changes from a specific file in another commit without using cherry-pick at all.

1. **Check out a specific file from the commit:**

git checkout <commit-hash> -- <file-you-want>

This brings the specific file’s changes from the commit into your working directory.

1. **Stage and commit the file:**

git add <file-you-want>

git commit -m "Applied changes from <commit-hash> to <file-you-want>"

By following these steps, you can effectively "cherry-pick" individual files from a commit without applying all the changes from the commit.

**GIT HOOKS**

Git hooks are scripts that Git executes before or after specific events like committing, pushing, or receiving changes. These hooks allow you to automate tasks and enforce policies in your Git workflows.

Here are a few exercises to help you understand how to use Git hooks effectively.

**Exercise 1: Create a Pre-Commit Hook**

**Objective:**

Create a pre-commit hook that checks for TODO comments in the code and prevents the commit if any are found.

**Steps:**

1. **Navigate to your Git repository:**

cd /path/to/your/git/repository

1. **Locate the hooks directory:** Inside the .git folder, there is a hooks directory with sample hook scripts.

cd .git/hooks

1. **Create a pre-commit hook:**
   * Open or create the pre-commit file in the .git/hooks directory:

touch pre-commit

chmod +x pre-commit

1. **Edit the pre-commit hook:**
   * Add the following script to check for TODO comments in staged files:

#!/bin/bash

# Check for TODO comments

if git diff --cached | grep -q 'TODO'; then

echo "Error: You have TODO comments in your code. Please remove them before committing."

exit 1

fi

exit 0

1. **Test the hook:**
   * Add a TODO comment in one of your files:

echo "// TODO: fix this issue" >> file.txt

git add file.txt

git commit -m "Test commit"

* + The hook should now prevent you from committing until the TODO comment is removed.

**Exercise 2: Create a Post-Commit Hook**

**Objective:**

Create a post-commit hook that automatically prints the latest commit details.

**Steps:**

1. **Create a post-commit hook:**
   * Navigate to the hooks directory and create the post-commit hook:

cd .git/hooks

touch post-commit

chmod +x post-commit

1. **Edit the post-commit hook:**
   * Add the following script to display the latest commit information:

#!/bin/bash

# Show the latest commit details

echo "Latest commit details:"

git log -1 --pretty=format:"%h - %s by %an on %ad"

1. **Test the hook:**
   * Make a new commit and check if the latest commit details are printed automatically:

echo "New line" >> file.txt

git add file.txt

git commit -m "Added a new line"

**Exercise 3: Create a Pre-Push Hook**

**Objective:**

Create a pre-push hook that ensures all commits are signed using GPG.

**Steps:**

1. **Set up GPG for signing commits** (if not done already):
   * Generate a GPG key if you don’t have one:

gpg --gen-key

* + Configure Git to use your GPG key:

git config --global user.signingkey <your-gpg-key-id>

git config --global commit.gpgSign true

1. **Create the pre-push hook:**
   * Navigate to the .git/hooks directory and create the pre-push hook:

cd .git/hooks

touch pre-push

chmod +x pre-push

1. **Edit the pre-push hook:**
   * Add the following script to check if all commits are signed:

#!/bin/bash

# Check if all commits in the push are signed

unsigned\_commits=$(git log origin/$(git rev-parse --abbrev-ref HEAD)..HEAD --pretty=format:"%h" --no-merges --not --grep='^gpg:')

if [ ! -z "$unsigned\_commits" ]; then

echo "Error: The following commits are not signed:"

echo "$unsigned\_commits"

exit 1

fi

exit 0

1. **Test the hook:**
   * Make a new unsigned commit:

git commit --no-gpg-sign -m "Unsigned commit"

git push

* + The hook should prevent the push because the commit is not signed.

**Exercise 4: Create a Post-Merge Hook**

**Objective:**

Create a post-merge hook that runs unit tests after a successful merge.

**Steps:**

1. **Create the post-merge hook:**
   * Navigate to the .git/hooks directory and create the post-merge hook:

cd .git/hooks

touch post-merge

chmod +x post-merge

1. **Edit the post-merge hook:**
   * Add the following script to run a test suite after a merge:

#!/bin/bash

echo "Running tests after merge..."

# Assuming you have a test command, e.g., npm test, or a custom script

./run\_tests.sh

if [ $? -ne 0 ]; then

echo "Tests failed! Please fix before continuing."

exit 1

else

echo "All tests passed!"

fi

1. **Test the hook:**
   * Simulate a merge and verify that the tests run automatically after the merge:

git merge some-branch

* + If you don't have a test script, you can replace ./run\_tests.sh with something like:

echo "Tests passed!" # Simulating a successful test

**Exercise 5: Create a Pre-Commit Hook to Enforce a Code Style**

**Objective:**

Create a pre-commit hook that checks if all .js files follow a specific code style (using eslint).

**Prerequisite:**

* Ensure eslint is installed in your project:

npm install eslint --save-dev

**Steps:**

1. **Create the pre-commit hook:**
   * Navigate to the .git/hooks directory and create the pre-commit hook:

cd .git/hooks

touch pre-commit

chmod +x pre-commit

1. **Edit the pre-commit hook:**
   * Add the following script to run eslint on staged .js files:

#!/bin/bash

# Check all staged .js files

js\_files=$(git diff --cached --name-only --diff-filter=ACM | grep '\.js$')

if [ "$js\_files" != "" ]; then

echo "Running ESLint on the following files:"

echo "$js\_files"

# Run ESLint

./node\_modules/.bin/eslint $js\_files

# Check if ESLint passed

if [ $? -ne 0 ]; then

echo "ESLint failed. Please fix the errors before committing."

exit 1

fi

fi

exit 0

1. **Test the hook:**
   * Stage a .js file with code style violations:

echo "var x = 1;" >> bad\_code.js

git add bad\_code.js

git commit -m "Test ESLint hook"

* + The hook should run ESLint and prevent the commit if there are any issues.

**Exercise 6: Create a Post-Checkout Hook**

**Objective:**

Create a post-checkout hook that notifies the user whenever they switch branches.

**Steps:**

1. **Create the post-checkout hook:**
   * Navigate to the .git/hooks directory and create the post-checkout hook:

cd .git/hooks

touch post-checkout

chmod +x post-checkout

1. **Edit the post-checkout hook:**
   * Add the following script to notify the user when a branch change occurs:

#!/bin/bash

previous\_branch=$1

new\_branch=$(git rev-parse --abbrev-ref HEAD)

echo "You have switched from $previous\_branch to $new\_branch."

1. **Test the hook:**
   * Switch branches and see the message:

git checkout some-branch

**Summary of Common Git Hooks:**

* **pre-commit**: Runs before a commit. Useful for validating code (linting, tests).
* **post-commit**: Runs after a commit. Useful for notifications or additional logging.
* **pre-push**: Runs before a push. Useful for checking if commits meet certain criteria (e.g., signed commits).
* **post-merge**: Runs after a successful merge. Useful for running tests or cleaning up.
* **post-checkout**: Runs after checking out a branch. Useful for notifying users or setting up environments.