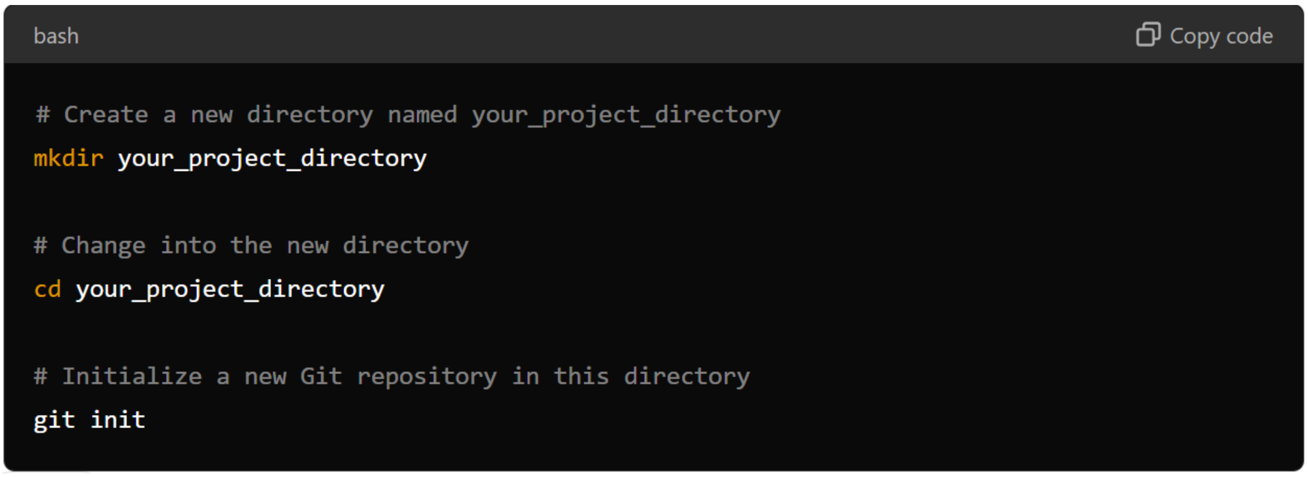
Project Management With GIT (BCS358C)

Lab 1. Setting Up and Basic Commands

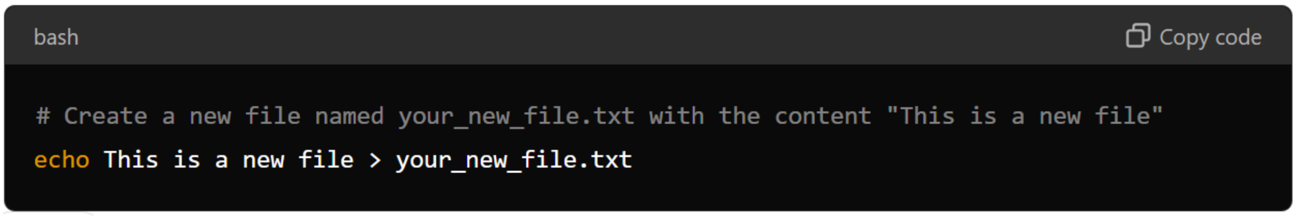
Initialise a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.

Step1: Initialise a new Git repository:



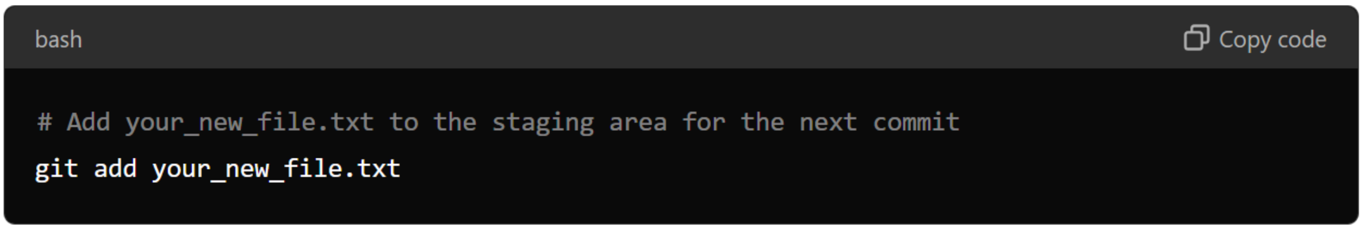
```bash
# Create a new directory named your_project_directory
mkdir your_project_directory

# Change into the new directory
cd your_project_directory

# Initialize a new Git repository in this directory
git init
```

Step 2. Create a new file:



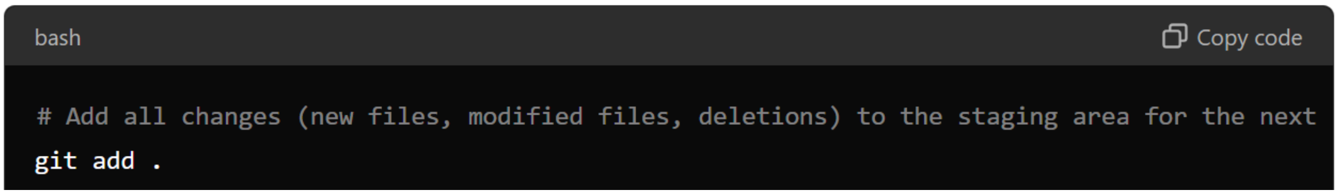
```bash
# Create a new file named your_new_file.txt with the content "This is a new file"
echo This is a new file > your_new_file.txt
```

Step 3. Add the file to the staging area:



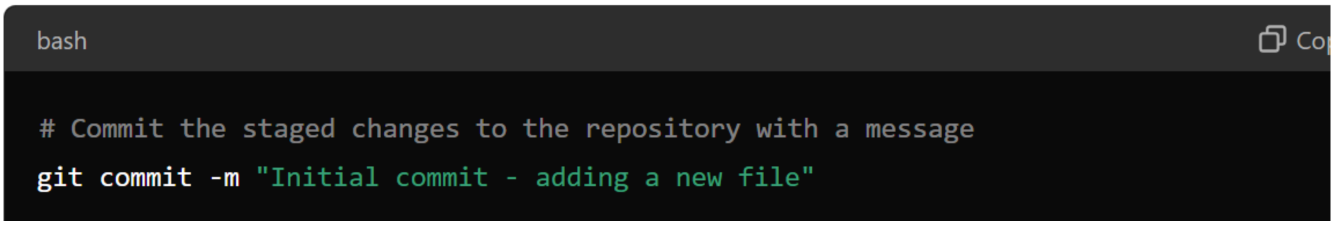
```bash
# Add your_new_file.txt to the staging area for the next commit
git add your_new_file.txt
```

If you want to add all new and modified files to the staging area, you can use:



```bash
# Add all changes (new files, modified files, deletions) to the staging area for the next
git add .
```

Step 4. Commit the changes with a message:



```bash
# Commit the staged changes to the repository with a message
git commit -m "Initial commit - adding a new file"
```

Lab 2. Creating and Managing Branches

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

Step 1: Create a new branch named “feature-branch”:

```bash
# Create a new branch named feature-branch
git branch feature-branch
```

Alternatively, you can create and switch to the new branch in one step.

```bash
# Create a new branch named feature-branch and switch to it
git checkout -b feature-branch
```

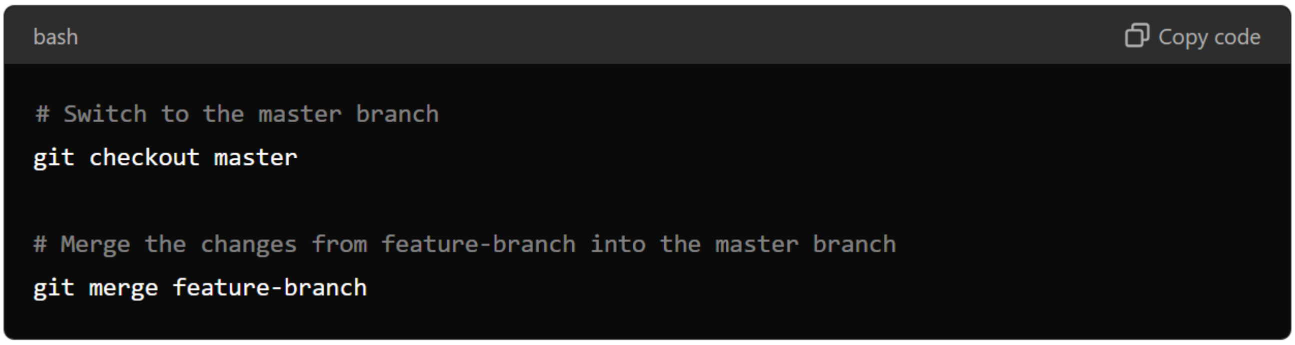
Step 2: Switch to the “master” branch.

```bash
# Switch to the master branch
git checkout master
```

Step 3: Merge “feature-branch” into “master”.

```bash
# Merge the changes from the feature-branch into the current branch (master)
git merge feature-branch
```

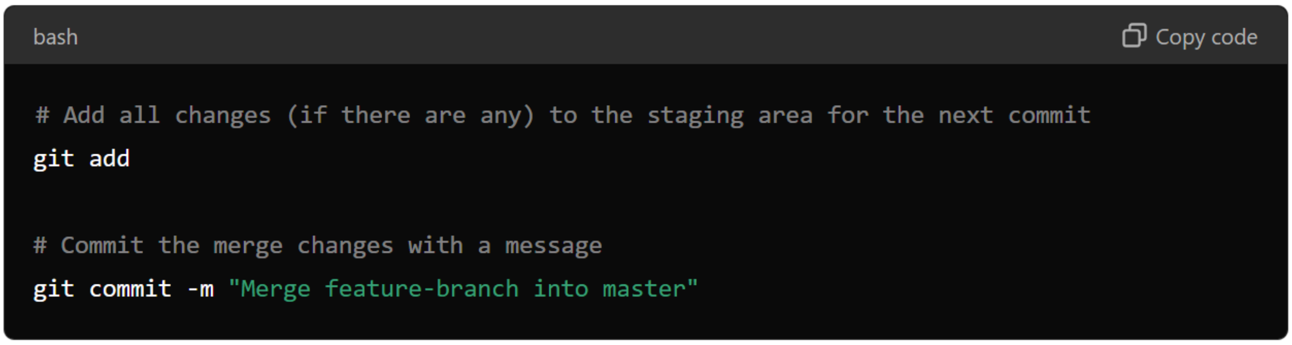
If there are no conflicts, Git will automatically perform a fast-forward merge. If there are conflicts, Git will prompt you to resolve them before completing the merge. If you used git checkout -b feature branch to create and switch to the new branch, you can switch back to master and merge in a single command.



```bash
# Switch to the master branch
git checkout master

# Merge the changes from feature-branch into the master branch
git merge feature-branch
```

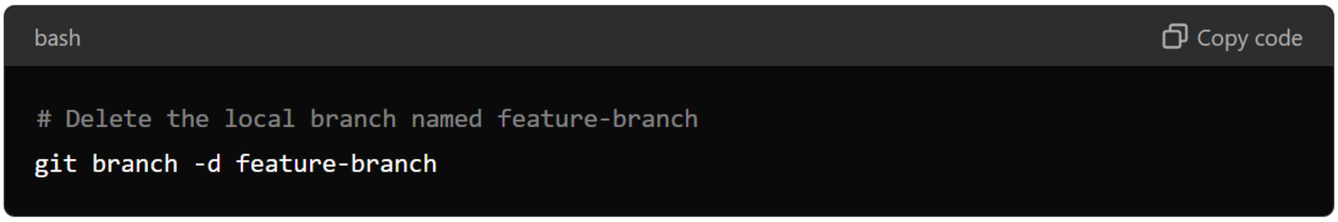
Step 4: Resolve any conflicts (if needed) and commit the merge:

If there are conflicts, Git will mark the conflicted files. Open each conflicted file, resolve the conflicts, and then:



```bash
# Add all changes (if there are any) to the staging area for the next commit
git add

# Commit the merge changes with a message
git commit -m "Merge feature-branch into master"
```

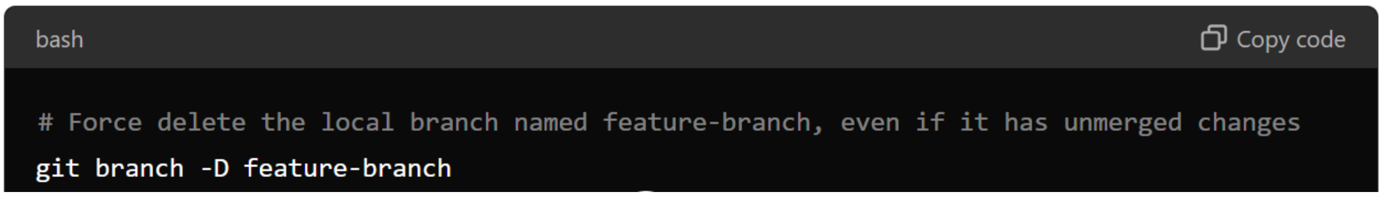
Now, the changes from “feature-branch” are merged into the “master” branch. If you no longer need the “feature-branch,” you can delete it



```bash
# Delete the local branch named feature-branch
git branch -d feature-branch
```

This assumes that the changes in “feature-branch” do not conflict with changes in the “master” branch. If conflicts arise during the merge, you’ll need to resolve them manually before completing the merge.

The `-d` option will delete the branch only if it has been fully merged into the current branch. If the branch contains unmerged changes, use `-D` to force delete it:



```bash
# Force delete the local branch named feature-branch, even if it has unmerged changes
git branch -D feature-branch
```

Lab 3. Creating and Managing Branches

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

Step 1. Stash your changes:

```bash
# Save the current changes in a stash with a descriptive message
git stash save "Your stash message"
```

This command will save your local changes in a temporary area, allowing you to switch branches without committing the changes.

Step 2. Switch to another branch:

```bash
# Switch to the branch named your-desired-branch
git checkout your-desired-branch
```

Step 3. Apply the stashed changes:

```bash
# Apply the most recent stash to the current working directory
git stash apply
```

If you have multiple stashes and want to apply a specific stash, you can use:

```bash
# Apply a specific stash by its identifier (e.g., stash@{1})
git stash apply stash@{1}
```

After applying the stash, your changes are reapplied to the working directory.

Step 4. Remove the applied stash (optional):

If you no longer need the stash after applying it, you can remove it:

```bash
# Remove the most recent stash from the stash list
git stash drop
```

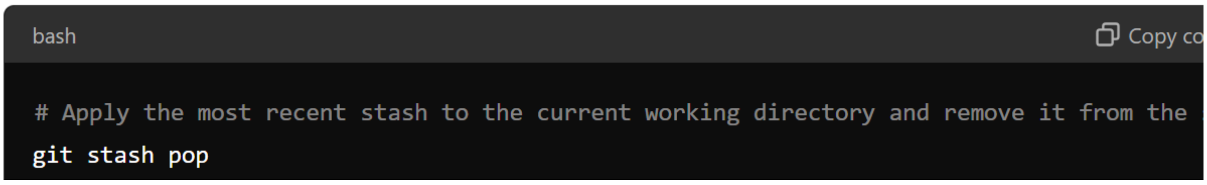
To remove a specific stash:

```bash
# Remove a specific stash by its identifier (e.g., stash@{1})
git stash drop stash@{1}
```

If you want to clear all the stashes, you can use:

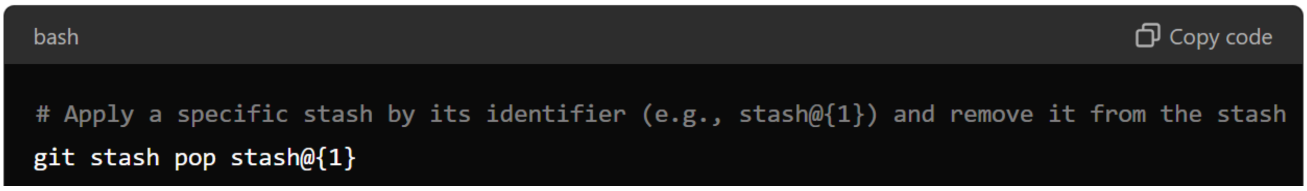
```bash
# Remove all stashes
git stash clear
```

If you want to apply and drop in one step, you can use git stash pop:



```bash
# Apply the most recent stash to the current working directory and remove it from the
git stash pop
```

The `git stash pop` command applies the latest stash and then deletes it from the stash list, combining the actions of `git stash apply` and `git stash drop`. If you need to pop a specific stash, you can provide its identifier:



```bash
# Apply a specific stash by its identifier (e.g., stash@{1}) and remove it from the stash
git stash pop stash@{1}
```

Now, you’ve successfully stashed your changes, switched branches, and applied the stashed changes.

Lab 4. Collaboration and Remote Repositories

Clone a remote Git repository to your local machine.

To clone a remote Git repository to your local machine, you can use the `git clone` command. Here's the general syntax:

```bash
# Clone a repository from the specified URL to a local directory
git clone <repository_url>
```

Replace <repository_url> with the actual URL of the Git repository you want to clone. For example:

```bash
# Clone a repository from GitHub
git clone https://github.com/user/repository.git
```

This command will create a new directory with the name of the repository and download all the files from the remote repository into that directory. If the repository is private and requires authentication, you might need to use the SSH URL or provide your credentials during the cloning process.

For SSH:

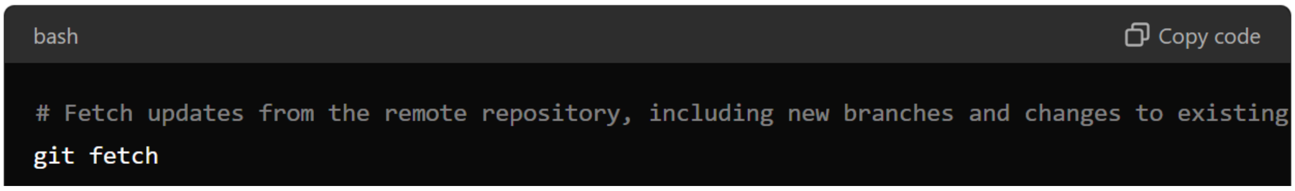
```bash
# Clone the repository from GitHub using SSH
git clone git@github.com:example/repo.git
```

After running the `git clone` command, you’ll have a local copy of the remote repository on your machine, and you can start working with the code.

Lab 5. Collaboration and Remote Repositories

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

Step 1. Fetch the latest changes from the remote repository:

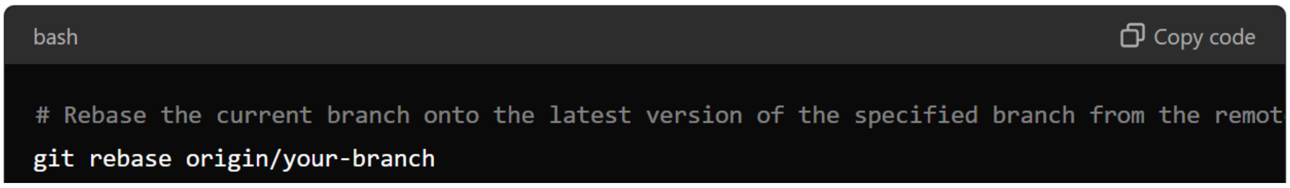


```bash
# Fetch updates from the remote repository, including new branches and changes to existing
git fetch
```

This command fetches the latest changes from the remote repository without automatically merging them into your local branches.

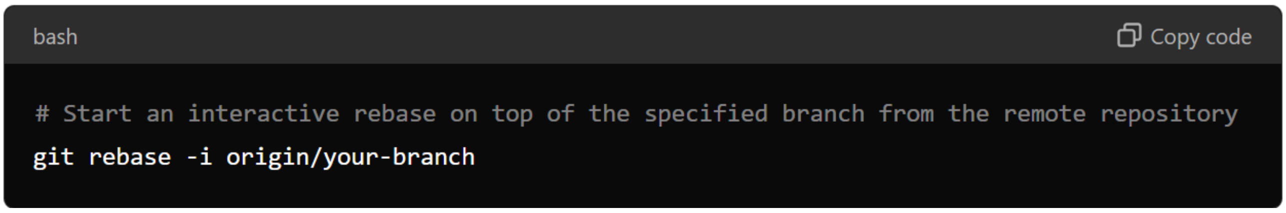
Step 2. Rebase your local branch onto the updated remote branch:

Assuming you are currently on the branch you want to update (replace your-branch with the actual name of your branch):



```bash
# Rebase the current branch onto the latest version of the specified branch from the remot
git rebase origin/your-branch
```

This command applies your local commits on top of the changes fetched from the remote branch. If conflicts arise, Git will pause the rebase process and ask you to resolve them. Alternatively, you can use the interactive rebase to review and modify commits during the rebase:



```bash
# Start an interactive rebase on top of the specified branch from the remote repository
git rebase -i origin/your-branch
```

This opens an editor where you can pick, squash, or edit individual commits.

Step 3. Continue the rebase or resolve conflicts:

If conflicts occurred during the rebase, Git will prompt you to resolve them. After resolving conflicts, you can continue the rebase with:

```bash
# After resolving conflicts, continue the rebase process
git rebase --continue
```

If you decide to abort the rebase at any point, you can use:

```bash
# Abort the rebase process and return to the state before rebase
git rebase --abort
```

Step 4. Push the rebased branch to the remote repository:

After successfully rebasing your local branch, you may need to force-push the changes to the remote repository:

```bash
# Force push the current branch to the remote repository, overwriting the remote branch
git push origin your-branch --force
```

Be cautious with force-pushing, especially if others are working with the same branch, as it rewrites the commit history. Now, your local branch is rebased onto the updated remote branch. Keep in mind that force-pushing should be done with caution, especially on shared branches, to avoid disrupting collaborative work.

A safer alternative to `--force` is `--force-with-lease`, which checks if the remote branch has been updated before force-pushing. This prevents accidental overwrites of changes made by others:

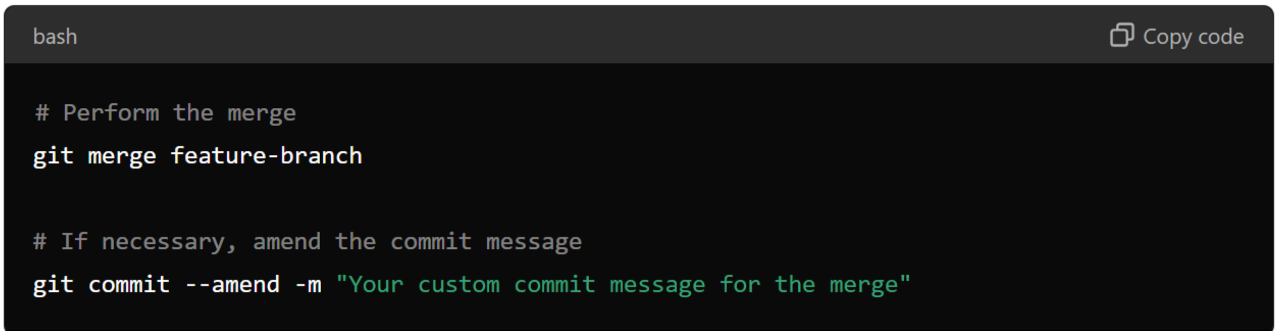
```bash
git push origin your-branch --force-with-lease
```

Lab 6. Collaboration and Remote Repositories

Write the command to merge “feature-branch” into “master” while providing a custom commit message for the merge.

The `git merge` command does not support the `-m` option for providing a custom commit message directly within the merge command. The `-m` option is used for creating commit messages in `git commit`, not `git merge`.

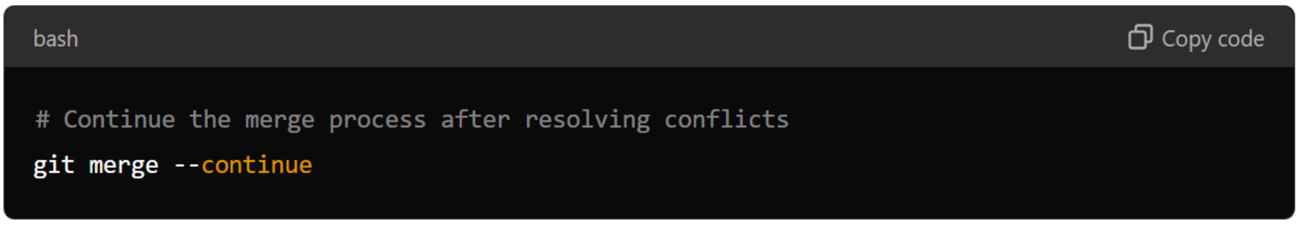
When you perform a merge, Git automatically generates a commit message by default. If you want to provide a custom commit message during a merge, you can use the `-m` option with `git commit` after the merge is completed.



```bash
# Perform the merge
git merge feature-branch

# If necessary, amend the commit message
git commit --amend -m "Your custom commit message for the merge"
```

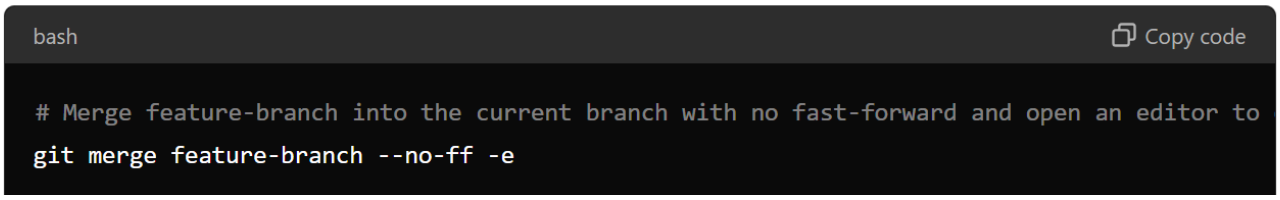
Replace “Your custom commit message” with the actual message you want to use for the merge commit. This command performs the merge and creates a new commit on the “master” branch with the specified message. If there are no conflicts, Git will complete the merge automatically.

If conflicts occur during the merge, Git will pause and prompt you to resolve the conflicts manually. After resolving conflicts, you can continue the merge process with:



```bash
# Continue the merge process after resolving conflicts
git merge --continue
```

Alternatively, you can use an interactive merge to modify the commit message before finalizing the merge:



```bash
# Merge feature-branch into the current branch with no fast-forward and open an editor to
git merge feature-branch --no-ff -e
```

`--no-ff`: This option forces the creation of a merge commit even if the merge could be performed as a fast-forward. This is useful for keeping a record of the merge in the commit history.

This opens an editor where you can edit the commit message before completing the merge. Again, replace “feature-branch” with the name of your actual feature branch.

Lab 7: Git Tags and Releases

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

To create a lightweight Git tag named “v1.0” for a specific commit in your local repository, you can use the following command:

```bash
# Create a new tag named v1.0 for a specific commit identified by <commit_hash>
git tag v1.0 <commit_hash>
```

`git tag`: This command is used to create, list, delete, or verify tags in Git.
`v1.0`: This is the name of the tag you want to create. Tags are often used to mark specific points in the commit history, such as releases or significant milestones.
`<commit_hash>`: This is the SHA-1 hash of the commit you want to tag. You can find this hash using commands like `git log`.

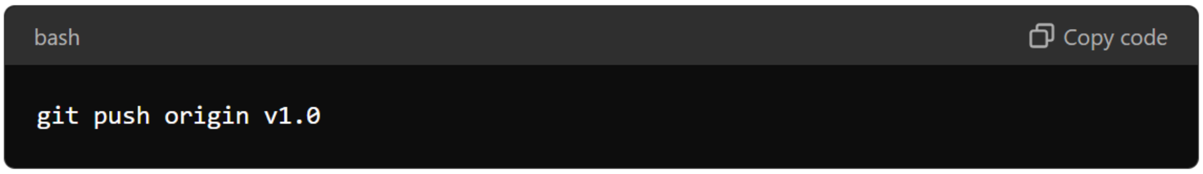
Replace <commit_hash> with the actual hash of the commit for which you want to create the tag.

For example, if you want to tag the latest commit, you can use the following:

```bash
# Create a new tag named v1.0 at the current commit pointed to by HEAD
git tag v1.0 HEAD
```

This creates a lightweight tag pointing to the specified commit. Lightweight tags are simply pointers to specific commits and contain only the commit checksum.

If you want to push the tag to a remote repository, you can use:



```bash
git push origin v1.0
```

This command pushes the tag named “v1.0” to the remote repository. Keep in mind that Git tags, by default, are not automatically pushed to remotes, so you need to explicitly push them if needed.

Lab 8. Advanced Git Operations

Write the command to cherry-pick a range of commits from “source-branch” to the current branch.

To cherry-pick a range of commits from “source-branch” to the current branch, you can use the following command:

```bash
# Cherry-pick a range of commits from <start-commit> to <end-commit> (inclusive) into the
git cherry-pick <start-commit>^..<end-commit>
```

`git cherry-pick`: This command applies the changes introduced by existing commits onto your current branch.
`<start-commit>`: This is the hash of the first commit in the range you want to cherry-pick.
`<end-commit>`: This is the hash of the last commit in the range you want to cherry-pick.
`^`: This notation refers to the commit immediately before `<start-commit>`. Including this ensures that `<start-commit>` itself is included in the range.

Replace <start-commit> and <end-commit> with the commit hashes or references that define the range of commits you want to cherry-pick. The ^ (caret) symbol is used to exclude the starting commit itself from the range.

For example, if you want to cherry-pick the commits from commit A to commit B (excluding A) from “source-branch” to the current branch, you would run:

```bash
# Cherry-pick all commits from commit A (exclusive) to commit B (inclusive) onto the curren
git cherry-pick A^..B
```

After running this command, Git will apply the specified range of commits onto your current branch. If there are any conflicts, Git will pause the cherry-pick process and ask you to resolve them. After resolving conflicts, you can continue the cherry-pick with:

```bash
git cherry-pick --continue
```

If you encounter issues and need to abort the cherry-pick operation, you can use:

```bash
git cherry-pick --abort
```

Remember that cherry-picking introduces new commits based on the changes from the source branch, so conflicts may arise, and manual intervention might be required.

Lab 9. Analyzing and Changing Git History

Given a commit ID, how would you use Git to view the details of that specific commit, including the author, date, and commit message?

To view the details of a specific commit, including the author, date, and commit message, you can use the following Git command:

```bash
git show <commit-id>
```

Replace <commit-id> with the actual commit hash or commit reference of the commit you want to inspect.

For example:

```bash
# Show details of the commit with hash abc123
git show abc123
```

This command will display detailed information about the specified commit, including the author, date, commit message, and the changes introduced by that commit.

If you only want a more concise summary of the commit information (without the changes), you can use:

```bash
# Show a single commit's log entry with a custom format for a specific commit identified b
git log -n 1 --pretty=format:"%h - %an, %ar : %s" <commit-id>
```

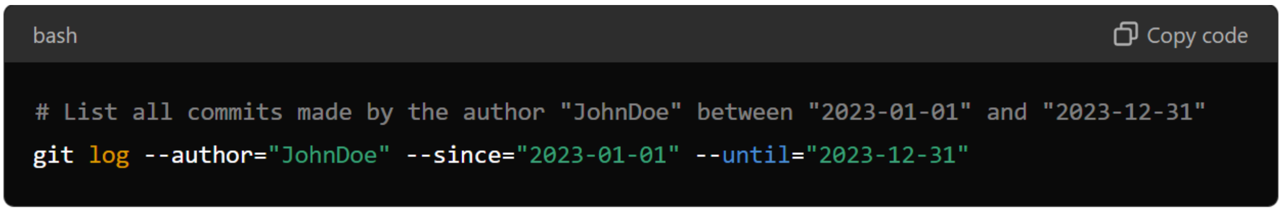
This command displays a one-line summary of the commit, showing the abbreviated commit hash (%h), author name (%an), relative author date (%ar), and commit message (%s).

Remember to replace <commit-id> with the actual commit hash or reference you want to inspect.

Lab 10. Analyzing and Changing Git History

Write the command to list all commits made by the author “JohnDoe” between “2023-01-01” and “2023 12-31.”

To list all commits made by the author “JohnDoe” between “2023-01-01” and “2023-12-31,” you can use the following git log command with the --author and --since / --until options:



```bash
# List all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31"
git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31"
```

This command will display the commit history that meets the specified criteria. Adjust the author name and date range according to your requirements. The --since and --until options accept a variety of date and time formats, providing flexibility in specifying the date range.

Lab 11. Analyzing and Changing Git History

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

To display the last five commits in the repository’s history, you can use the following git log command with the -n option:

```bash
# Display the most recent 5 commits in the current branch's history
git log -n 5
```

This command shows the latest five commits in the repository, with the most recent commit displayed at the top. Adjust the number after the -n option if you want to see a different number of commits.

If you want a more concise output, you can use the --oneline option:

```bash
git log -n 5 --oneline
```

This provides a one-line summary for each commit, including the abbreviated commit hash and the commit message.

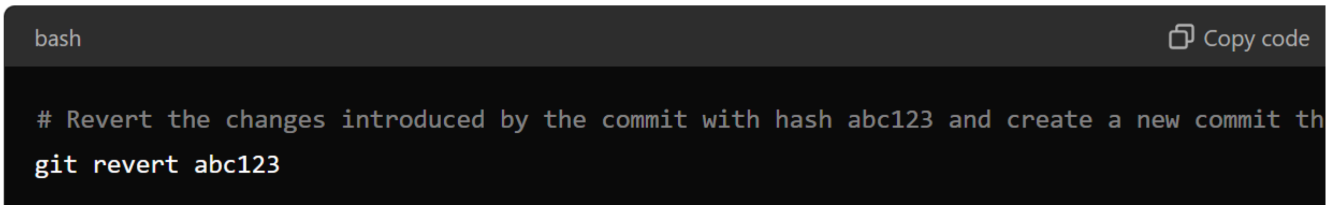
Lab 12. Analyzing and Changing Git History

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

To undo the changes introduced by a specific commit with the ID “abc123,” you can use the git revert command.

The git revert command creates a new commit that undoes the changes made in a previous commit. It does not alter the commit history but instead adds a new commit that reverses the changes.

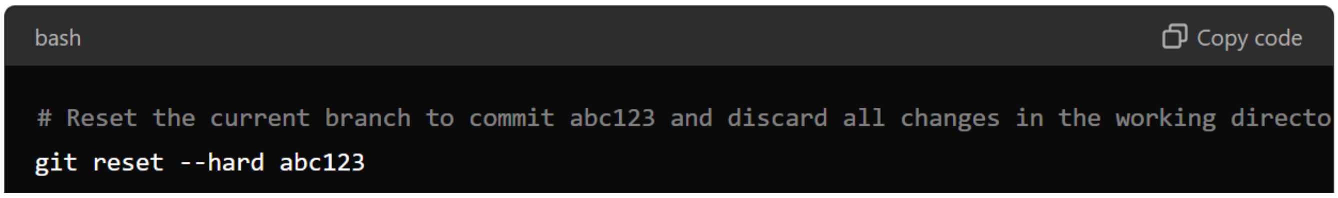
Here’s the command:



```bash
# Revert the changes introduced by the commit with hash abc123 and create a new commit th
git revert abc123
```

Replace “abc123” with the actual commit hash or commit reference of the commit you want to undo. After running this command, Git will open a text editor for you to provide a commit message for the new revert commit.

Alternatively, if you want to completely remove a commit and all of its changes from the commit history, you can use the git reset command. However, keep in mind that using git reset can rewrite history and should be used with caution, especially if the commit has been pushed to a remote repository.



```bash
# Reset the current branch to commit abc123 and discard all changes in the working directo
git reset --hard abc123
```

`git reset`: This command is used to reset the current branch's state to a specified commit.
`--hard`: This option discards all changes in the working directory and index (staging area), making the working directory match exactly the specified commit.

Again, replace “abc123” with the actual commit hash or commit reference.
After using git reset --hard, your working directory will be modified to match the specified commit, discarding all commits made after it. Be cautious when using --hard as it is a forceful operation and can lead to data loss.