Git and Github

A Distributed Version Controlled

Overview of Git

- What is Git ? Installation and Setup
- Introduction to Version Control and workflow of Git
- Understanding of Git init, status, add, commit, push
- Branching, Merging and Rebasing
- Working with upstream, remote with Github and CodeCommit

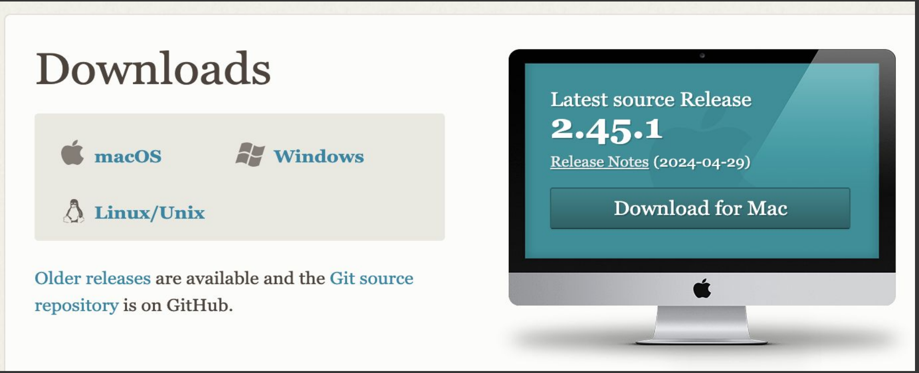
What is Git ?

A distributed version control system

- ➔ Everyone can act as the server
- ➔ Everyone mirrors the entire repository instead of simply checking out the latest version of the code.
- ➔ Cheap to create new branch, merge, etc
- ➔ Git and GitHub are different. Git is a software and Github is a service

Installation of Git Tool

https://git-scm.com/downloads



Downloads

macOS Windows

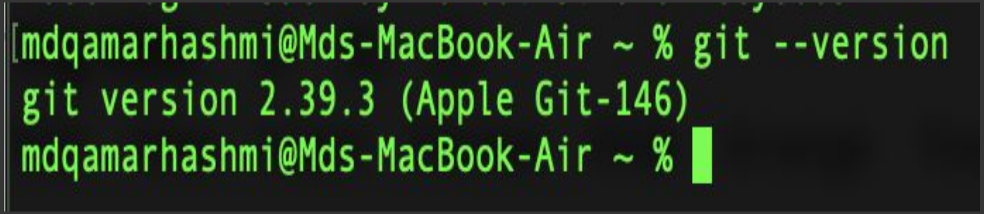
Linux/Unix

Older releases are available and the Git source repository is on GitHub.

Latest source Release
2.45.1
Release Notes (2024-04-29)

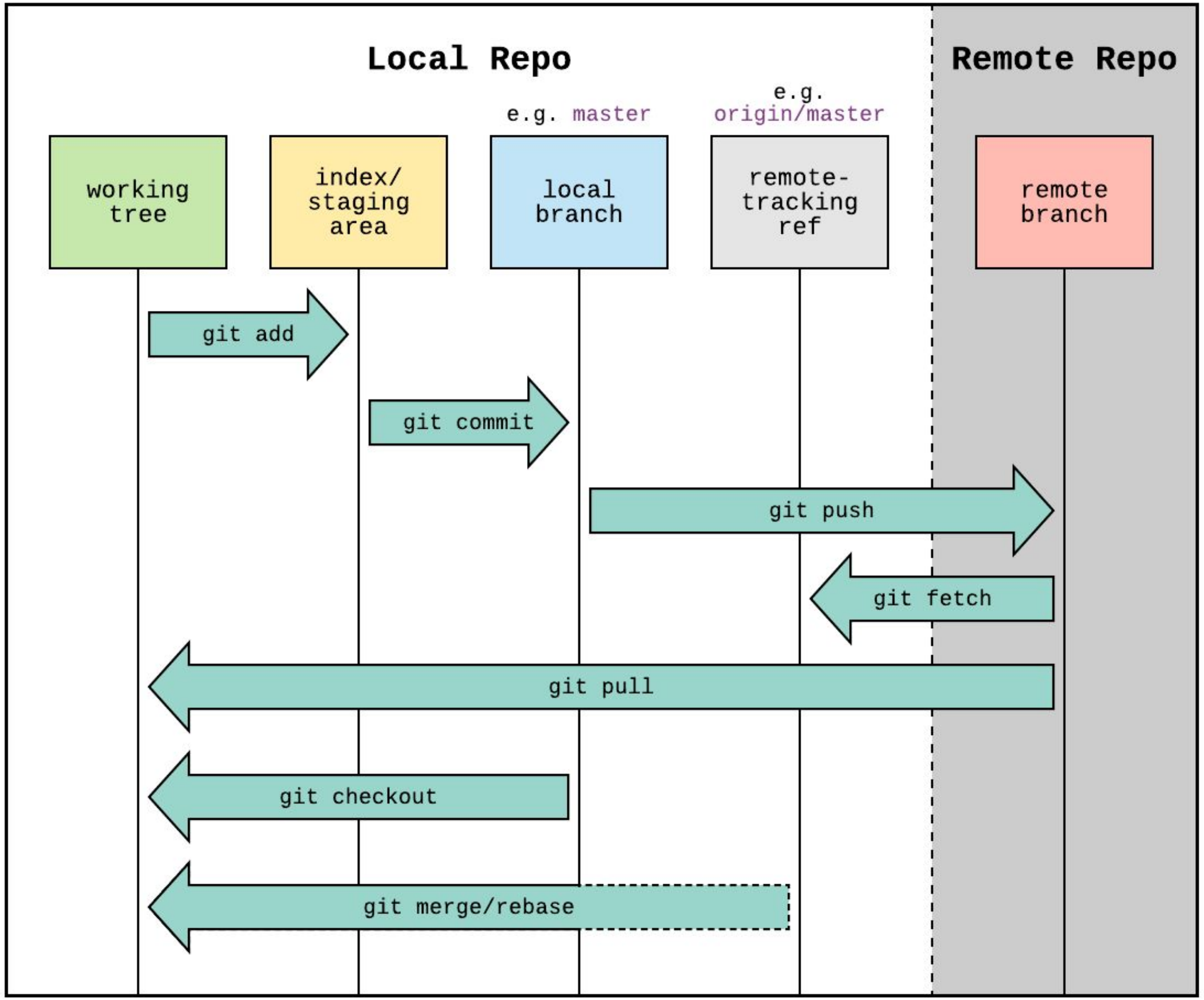
Download for Mac

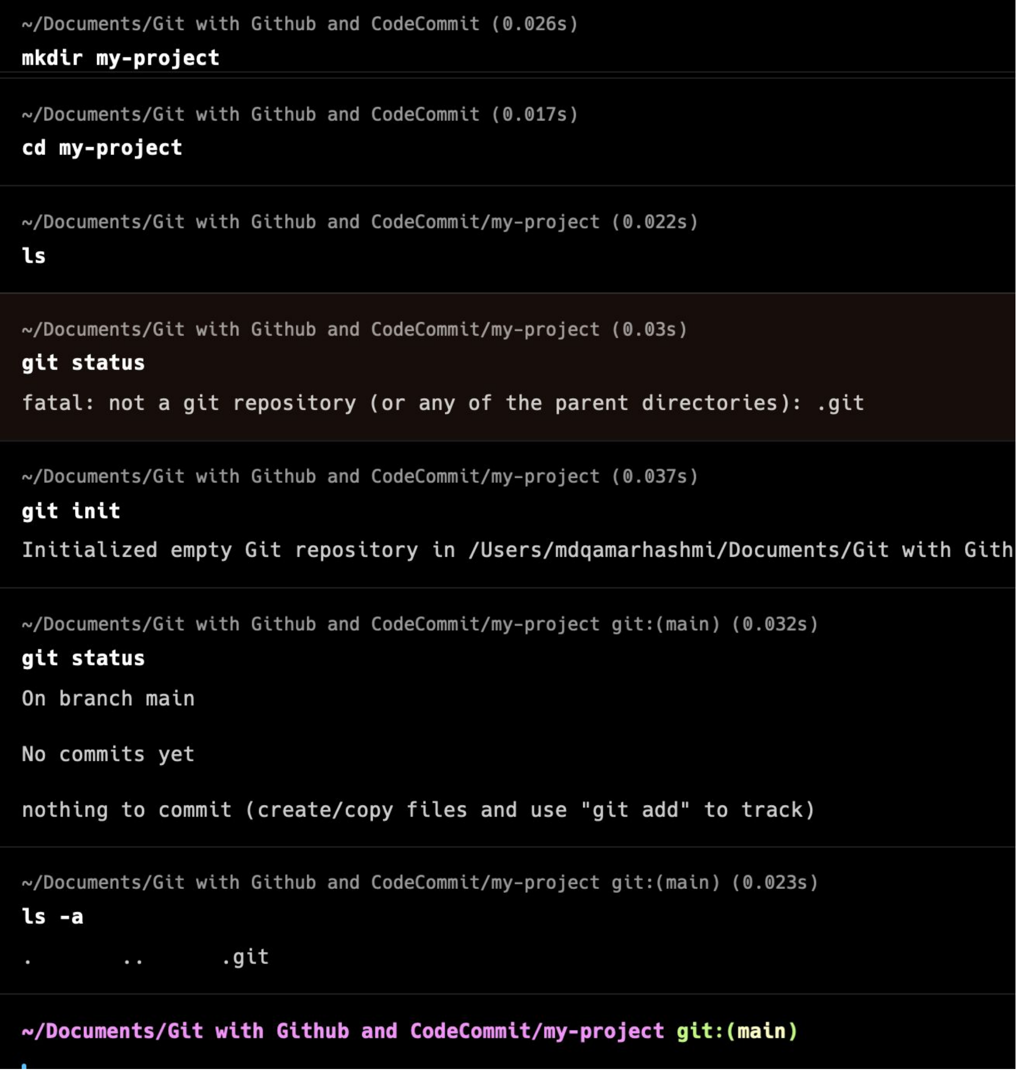
- Post installation check your git version using command "git --version"



```
mdqamarhashmi@Mds-MacBook-Air ~ % git --version
git version 2.39.3 (Apple Git-146)
mdqamarhashmi@Mds-MacBook-Air ~ %
```

Basic Workflow





```
~/Documents/Git with Github and CodeCommit (0.026s)
mkdir my-project

~/Documents/Git with Github and CodeCommit (0.017s)
cd my-project

~/Documents/Git with Github and CodeCommit/my-project (0.022s)
ls

~/Documents/Git with Github and CodeCommit/my-project (0.03s)
git status
fatal: not a git repository (or any of the parent directories): .git

~/Documents/Git with Github and CodeCommit/my-project (0.037s)
git init
Initialized empty Git repository in /Users/mdqamarhashmi/Documents/Git with Gith

~/Documents/Git with Github and CodeCommit/my-project git:(main) (0.032s)
git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)

~/Documents/Git with Github and CodeCommit/my-project git:(main) (0.023s)
ls -a
.       ..      .git

~/Documents/Git with Github and CodeCommit/my-project git:(main)
```

git "status" and "init"

Command : git status

- Shows current branch information. In my it shows "main"
- What changes to be committed
- Changes that are staged for commit
- Shows branch tracking information and etc

Command : git init

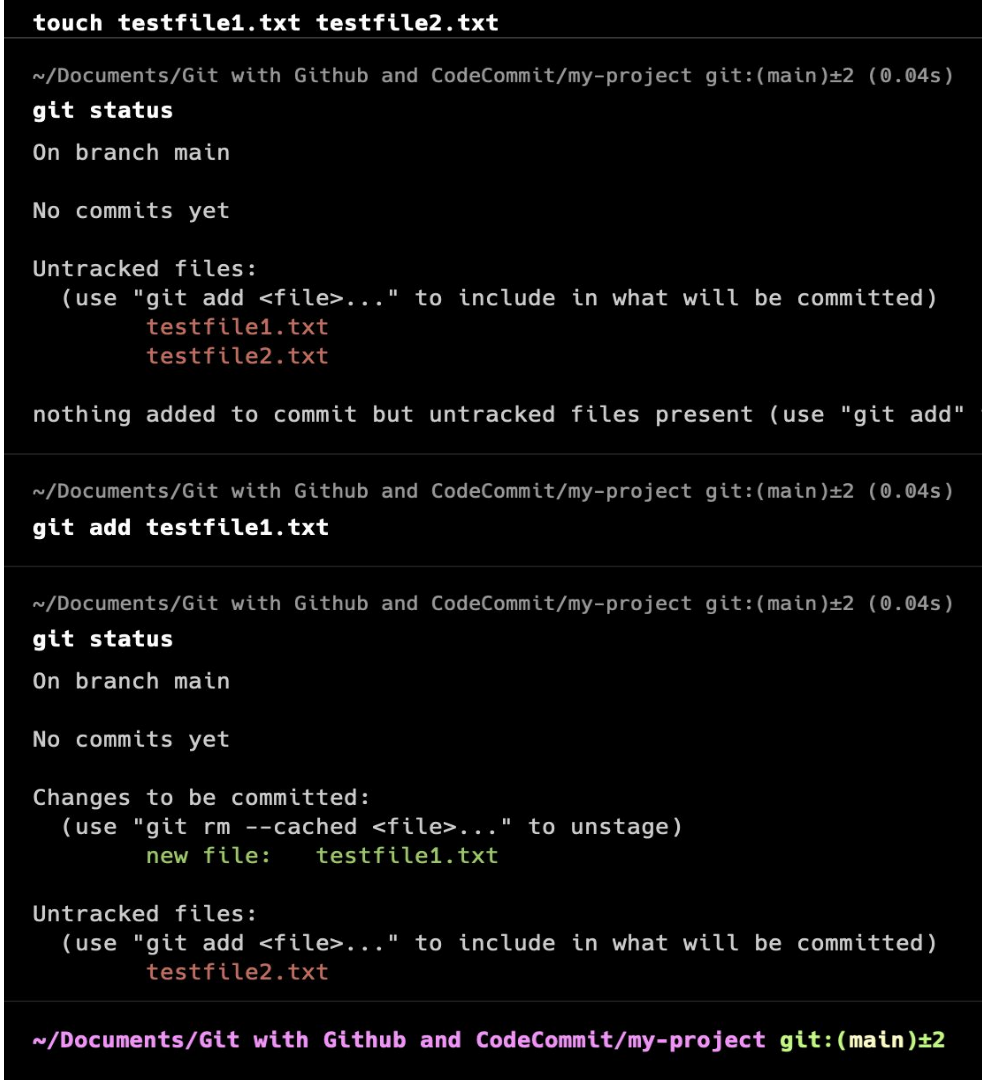
- Initializes a new Git repository in the current directory or in the specified directory.
- Creates a ".git" directory: This directory contains all the necessary Git metadata and configuration for the repository(like object database, configuration files, and the index.)
- No remote tracking
- Initialized only once per project.
- Undoing initialization : Remove .git folder. However, be cautious, as this action permanently removes all version control history and configuration.

git "add"

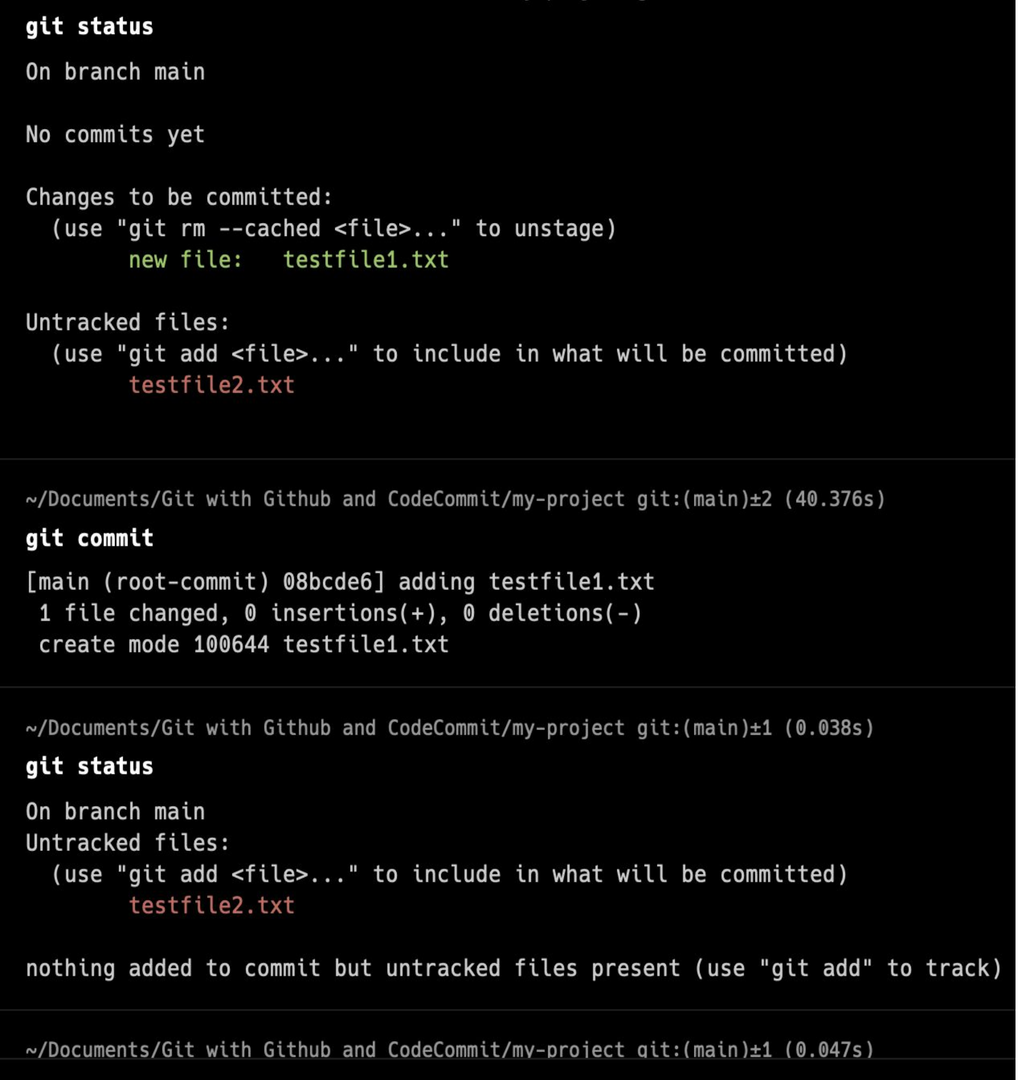
Command : git add <filename>

- Add files to be committed with "git add <filename>" or "git add ." Puts the file in the "staging area"
- Let's create two files named "testfile1.txt" and "testfile2.txt".
- Run "git status" to check the status and it will show two untracked files.
- Run "git add testfile1.txt" and perform "git status" to check the status.
- Status would show "testfile1.txt" as tracked and ready to commit. However, "testfile2.txt" shows as untracked.
- Just in case if you want to make any file as untracked run below command

Command : git rm --cached <file-name>



```
touch testfile1.txt testfile2.txt

~/Documents/Git with Github and CodeCommit/my-project git:(main)±2 (0.04s)
git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        testfile1.txt
        testfile2.txt

nothing added to commit but untracked files present (use "git add"

~/Documents/Git with Github and CodeCommit/my-project git:(main)±2 (0.04s)
git add testfile1.txt

~/Documents/Git with Github and CodeCommit/my-project git:(main)±2 (0.04s)
git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   testfile1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        testfile2.txt

~/Documents/Git with Github and CodeCommit/my-project git:(main)±2
```



```
git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   testfile1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        testfile2.txt


~/Documents/Git with Github and CodeCommit/my-project git:(main)±2 (40.376s)
git commit
[main (root-commit) 08bcde6] adding testfile1.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 testfile1.txt

~/Documents/Git with Github and CodeCommit/my-project git:(main)±1 (0.038s)
git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        testfile2.txt

nothing added to commit but untracked files present (use "git add" to track)

~/Documents/Git with Github and CodeCommit/my-project git:(main)±1 (0.047s)
```

git “commit”

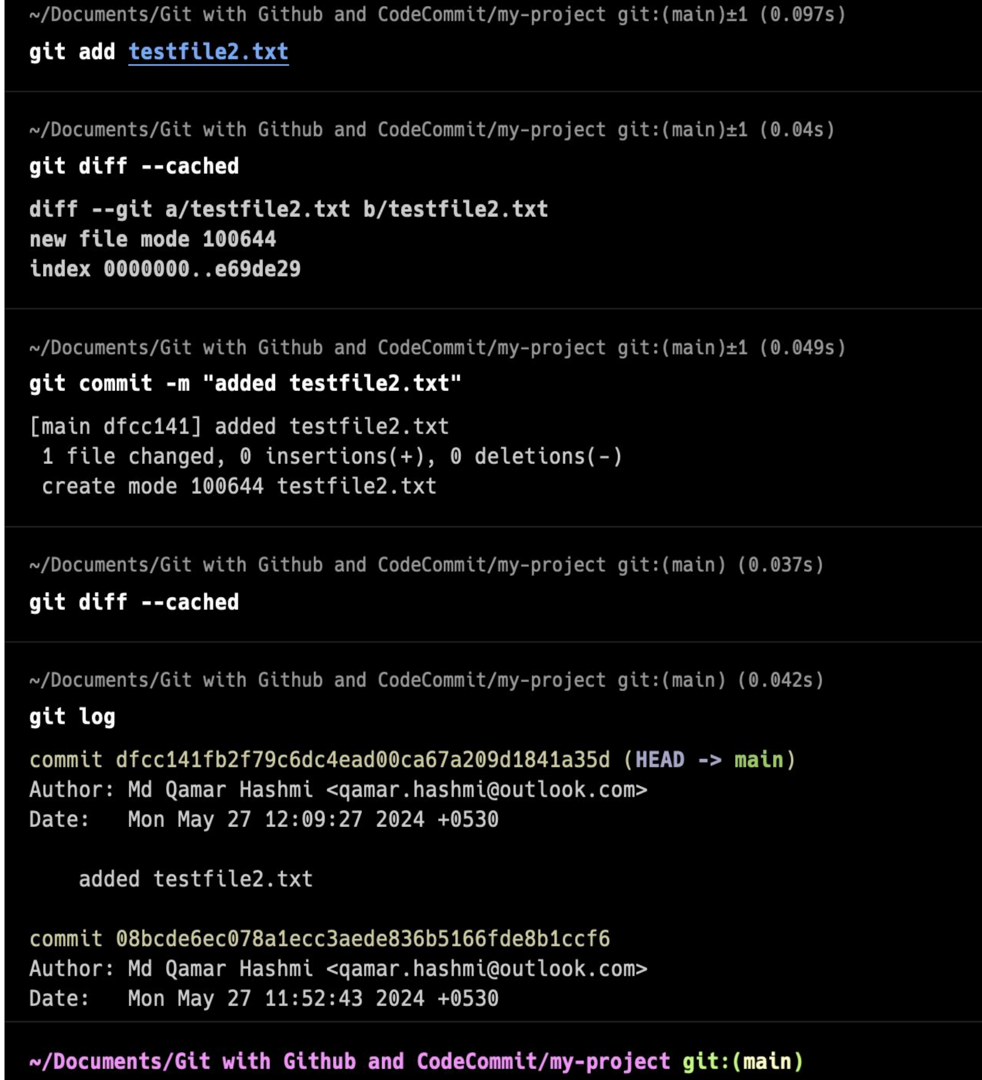
Command : “git commit” / git commit -m”added <file-name>”

- Push the code to local repository branch
- You can pass a message with commit by adding “-m”. If you don’t pass a message a terminal will open and ask you enter the message for that commit.
- The commit message should be descriptive and concise.
- Atomic Commits: Each commit should represent a single logical change. Avoid bundling unrelated changes into a single commit.
- Only staged changes will be included in the commit
- Review staged changes with “git diff --cached” before committing

git "log"

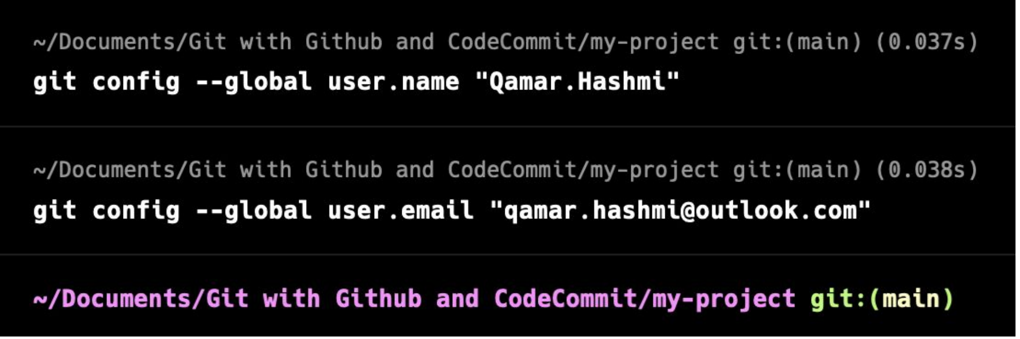
Command : git log

- "git log" displays a chronological list of commits in the repository, starting from the most recent. Each commit is represented by a unique hash, author, date, and commit message.
- You can filter the commit history using various options such as --author, --since, --until, --grep, etc. For example, git log --author="qamar" will show only commits authored by Qamar.
- Use options like "--max-count" or "-n" to limit the number of commits displayed. For example, "git log -n 5" will display the last 5 commits.
- "git log" can be combined with other Git commands such as grep, diff, show, etc., to perform more advanced operations on the commit history.
- By mastering git log, you gain insights into the evolution of your Git repository, enabling you to track changes, understand contributions, and troubleshoot issues effectively.

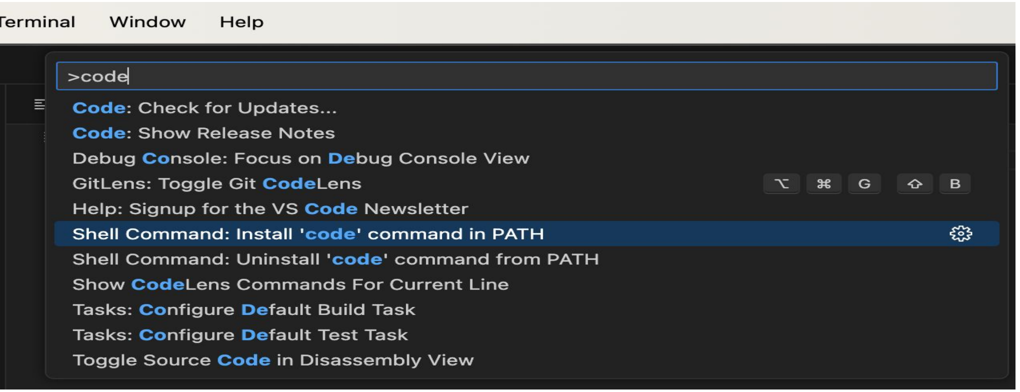


```
~/Documents/Git with Github and CodeCommit/my-project git:(main)±1 (0.097s)
git add testfile2.txt

~/Documents/Git with Github and CodeCommit/my-project git:(main)±1 (0.04s)
git diff --cached
diff --git a/testfile2.txt b/testfile2.txt
new file mode 100644
index 0000000..e69de29

~/Documents/Git with Github and CodeCommit/my-project git:(main)±1 (0.049s)
git commit -m "added testfile2.txt"
[main dfcc141] added testfile2.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 testfile2.txt

~/Documents/Git with Github and CodeCommit/my-project git:(main) (0.037s)
git diff --cached

~/Documents/Git with Github and CodeCommit/my-project git:(main) (0.042s)
git log
commit dfcc141fb2f79c6dc4ead00ca67a209d1841a35d (HEAD -> main)
Author: Md Qamar Hashmi <qamar.hashmi@outlook.com>
Date:   Mon May 27 12:09:27 2024 +0530

    added testfile2.txt

commit 08bcde6ec078a1ecc3aede836b5166fde8b1ccf6
Author: Md Qamar Hashmi <qamar.hashmi@outlook.com>
Date:   Mon May 27 11:52:43 2024 +0530

~/Documents/Git with Github and CodeCommit/my-project git:(main)
```

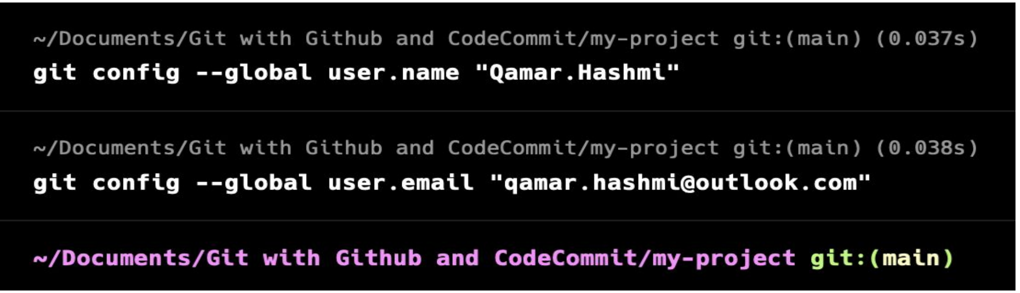
git “config”



```
~/Documents/Git with Github and CodeCommit/my-project git:(main) (0.037s)
git config --global user.name "Qamar.Hashmi"

~/Documents/Git with Github and CodeCommit/my-project git:(main) (0.038s)
git config --global user.email "qamar.hashmi@outlook.com"

~/Documents/Git with Github and CodeCommit/my-project git:(main)
```



```
Terminal    Window    Help

>code
Code: Check for Updates...
Code: Show Release Notes
Debug Console: Focus on Debug Console View
GitLens: Toggle Git CodeLens                          ⌥ ⌘ G  ⇧ B
Help: Signup for the VS Code Newsletter
Shell Command: Install 'code' command in PATH
Shell Command: Uninstall 'code' command from PATH
Show CodeLens Commands For Current Line
Tasks: Configure Default Build Task
Tasks: Configure Default Test Task
Toggle Source Code in Disassembly View
```



```
~/Documents/Git with Github and CodeCommit/my-project git:(main) (0.037s)
git config --global user.name "Qamar.Hashmi"

~/Documents/Git with Github and CodeCommit/my-project git:(main) (0.038s)
git config --global user.email "qamar.hashmi@outlook.com"

~/Documents/Git with Github and CodeCommit/my-project git:(main)
```

Command : “git config <key> <value>

- git config is a versatile command used to set or get configuration options for Git.
- Git configuration can be set globally for the user or locally for a specific repository. Global configuration affects all repositories on the system, while local configuration is specific to the current repository.
- Use git config --global to set global configuration options for the current user. Gets stored in “~/.gitconfig”
- Use git config without --global to set local configuration options for the current repository. Gets stored in “.git/config”.
- Let’s set user name. Use “user.email” to set email

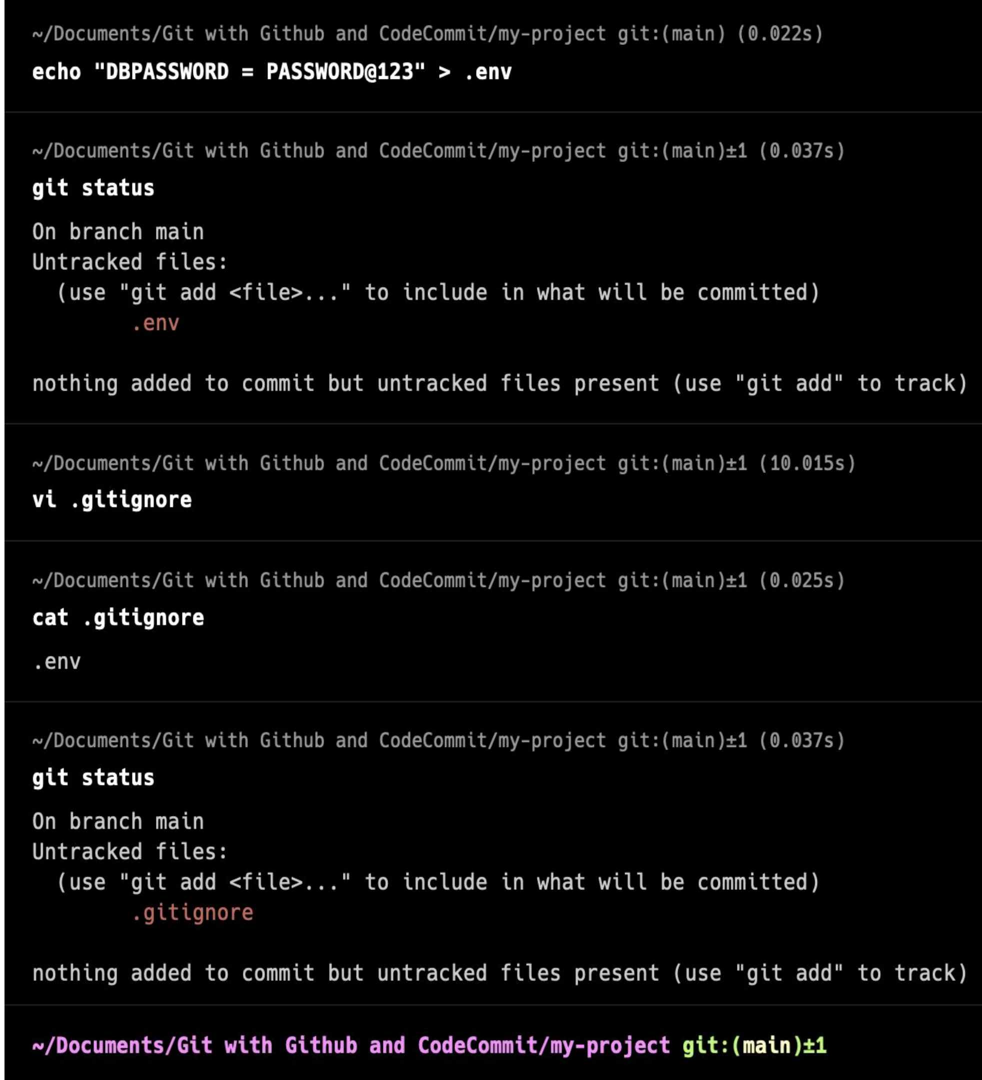
Command : git config --global user.name "Qamar Hashmi"

- Set “VS Code” as your core editor. Open VS code and perform below step
- For mac run “**command+shift+P**”,type “**code**” and set PATH. For Windows use “**control+shift+P**”

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

“.gitignore” working

- The .gitignore file tells Git which files or patterns it should ignore when tracking changes in a repository. For example, example.txt in the .gitignore file will ignore any file named example.txt.
- Ignored files will not be staged or committed unless explicitly added using “**git add -f**”.
- To ignore an entire directory and its contents, specify the directory name in the .gitignore file. For example, /logs/ in the .gitignore file will ignore the logs directory and all files within it.
- Use wildcard patterns to ignore files based on their extensions. For example, *.log in the .gitignore file will ignore all files with the .log extension.
- You can create a global .gitignore file that applies to all Git repositories on your system. Specify the global .gitignore file's path in your Git configuration using “**git config --global core.excludesfile**”.
- .gitignore generator : https://www.toptal.com/developers/gitignore



```
~/Documents/Git with Github and CodeCommit/my-project git:(main) (0.022s)
echo "DBPASSWORD = PASSWORD@123" > .env

~/Documents/Git with Github and CodeCommit/my-project git:(main)±1 (0.037s)
git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .env

nothing added to commit but untracked files present (use "git add" to track)

~/Documents/Git with Github and CodeCommit/my-project git:(main)±1 (10.015s)
vi .gitignore

~/Documents/Git with Github and CodeCommit/my-project git:(main)±1 (0.025s)
cat .gitignore
.env

~/Documents/Git with Github and CodeCommit/my-project git:(main)±1 (0.037s)
git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

nothing added to commit but untracked files present (use "git add" to track)

~/Documents/Git with Github and CodeCommit/my-project git:(main)±1
```

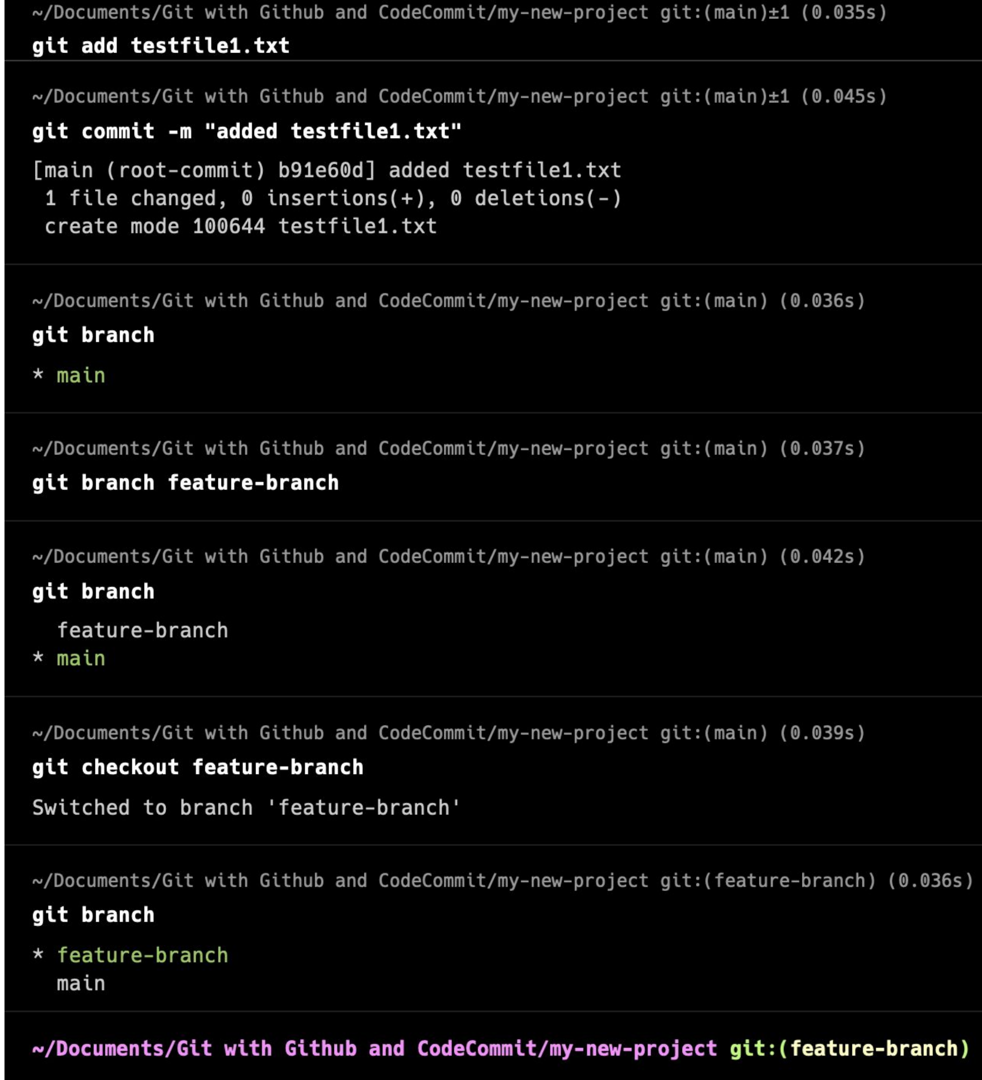
Working with Git Branch !

Branching in Git is a powerful feature that allows developers to work on separate lines of development without affecting the main codebase.

- The default branch in Git is usually called master or main.
- Git branching allows for various workflows, such as feature branching, release branching, and GitFlow.
- Regularly merge changes from the main branch into feature branches to keep them up-to-date.
- When you create a new branch, Git creates a new pointer that initially points to the same commit as the branch you were on.
- Keep branches focused on a single task or feature to simplify review and merging.

git "branch"

- Use the **git branch <branch-name>** command to create a new branch. For example, **git branch feature-branch** creates a new branch named feature-branch.
- Use the **git branch** command to list all existing branches in the repository.
- Use the **git checkout/switch <branch-name>** command to switch to a different branch. For example, **git checkout feature-branch** switches to the feature-branch branch.
- Before performing **git checkout <branch-name>** ref was referring to **ref: refs/heads/main**
- After performing **git checkout feature-branch** refer is pointing to **ref: refs/heads/feature-branch.**
- Use the **git merge <branch-name>** command to merge changes from one branch into another. For example, git merge feature-branch merges changes from feature-branch into the current branch.
- Use **git branch -d <branch-name>** command to delete the branch name



```
~/Documents/Git with Github and CodeCommit/my-new-project git:(main)±1 (0.035s)
git add testfile1.txt

~/Documents/Git with Github and CodeCommit/my-new-project git:(main)±1 (0.045s)
git commit -m "added testfile1.txt"
[main (root-commit) b91e60d] added testfile1.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 testfile1.txt

~/Documents/Git with Github and CodeCommit/my-new-project git:(main) (0.036s)
git branch
* main

~/Documents/Git with Github and CodeCommit/my-new-project git:(main) (0.037s)
git branch feature-branch

~/Documents/Git with Github and CodeCommit/my-new-project git:(main) (0.042s)
git branch
  feature-branch
* main

~/Documents/Git with Github and CodeCommit/my-new-project git:(main) (0.039s)
git checkout feature-branch
Switched to branch 'feature-branch'

~/Documents/Git with Github and CodeCommit/my-new-project git:(feature-branch) (0.036s)
git branch
* feature-branch
  main

~/Documents/Git with Github and CodeCommit/my-new-project git:(feature-branch)
```

Working with git merge

Merge Basics:

Merging is the process of combining changes from one branch into another. It's commonly used to integrate feature branches into the main development branch (e.g., master or main).

Fast-Forward Merge:

If the target branch (e.g., master) has not diverged from the source branch (e.g., feature-branch), Git performs a fast-forward merge. In a fast-forward merge, Git simply moves the target branch pointer to the latest commit of the source branch.

3-Way Merge:

When the branches have diverged, Git performs a 3-way merge. A 3-way merge compares the common ancestor commit of the branches with the latest commits of each branch to reconcile the changes.

Fast-Forward Merge

- Fast-forward merges are common in workflows where feature branches are frequently created and merged back into the main branch, especially when the feature branches have a linear history.
- Fast-forward merges are fast and efficient because they do not involve creating new commit objects or resolving conflicts. Git can simply move the branch pointer to the new commit.
- Let's understand with an example shown on the right side.

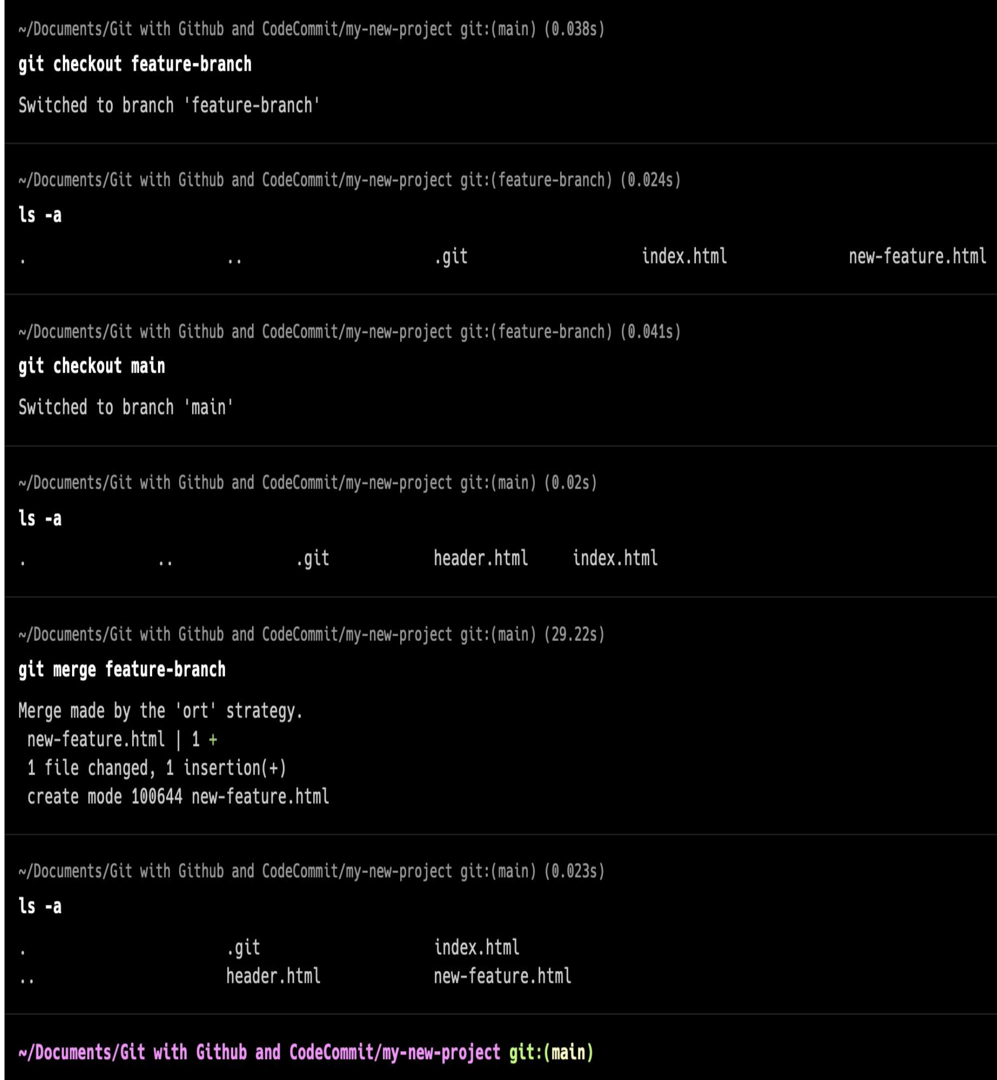
We have created two branch "main" and "feature-branch" with initially having only "index.html"

Next we added "new-feature.html" in feature-branch and "header.html" in main branch.

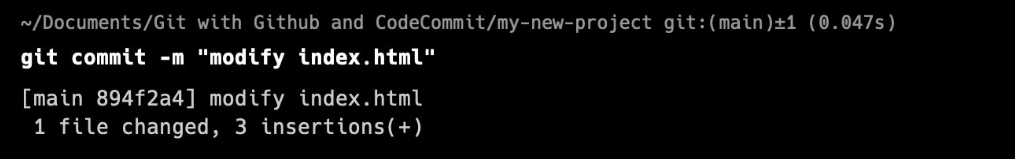
Please note, no changes has been made in index.html which was part of initial commit.

Now, we can simply perform a merge from the main branch using below command and there will be no conflicts between the two branches.

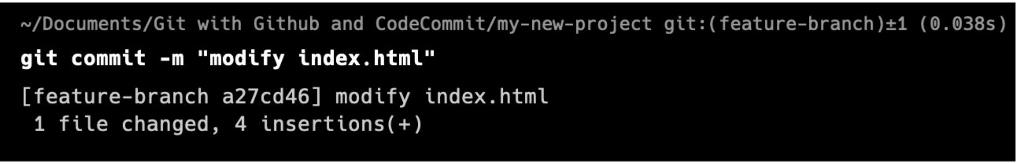
Command : git merge feature-branch -m "You_message"



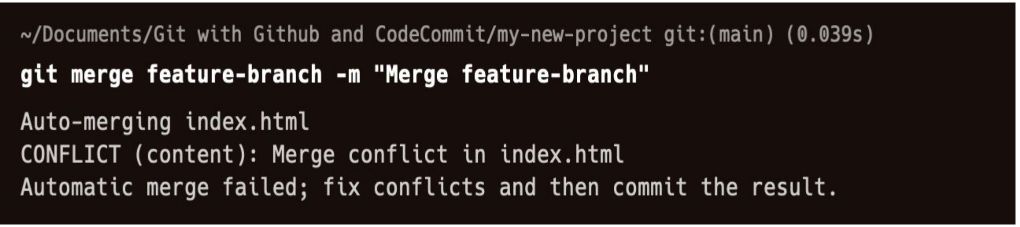
```
~/Documents/Git with Github and CodeCommit/my-new-project git:(main) (0.038s)
git checkout feature-branch
Switched to branch 'feature-branch'

~/Documents/Git with Github and CodeCommit/my-new-project git:(feature-branch) (0.024s)
ls -a
.                       ..                      .git                    index.html              new-feature.html

~/Documents/Git with Github and CodeCommit/my-new-project git:(feature-branch) (0.041s)
git checkout main
Switched to branch 'main'

~/Documents/Git with Github and CodeCommit/my-new-project git:(main) (0.02s)
ls -a
.               ..              .git            header.html     index.html

~/Documents/Git with Github and CodeCommit/my-new-project git:(main) (29.22s)
git merge feature-branch
Merge made by the 'ort' strategy.
 new-feature.html | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 new-feature.html

~/Documents/Git with Github and CodeCommit/my-new-project git:(main) (0.023s)
ls -a
.                       .git                    index.html
..                      header.html             new-feature.html

~/Documents/Git with Github and CodeCommit/my-new-project git:(main)
```



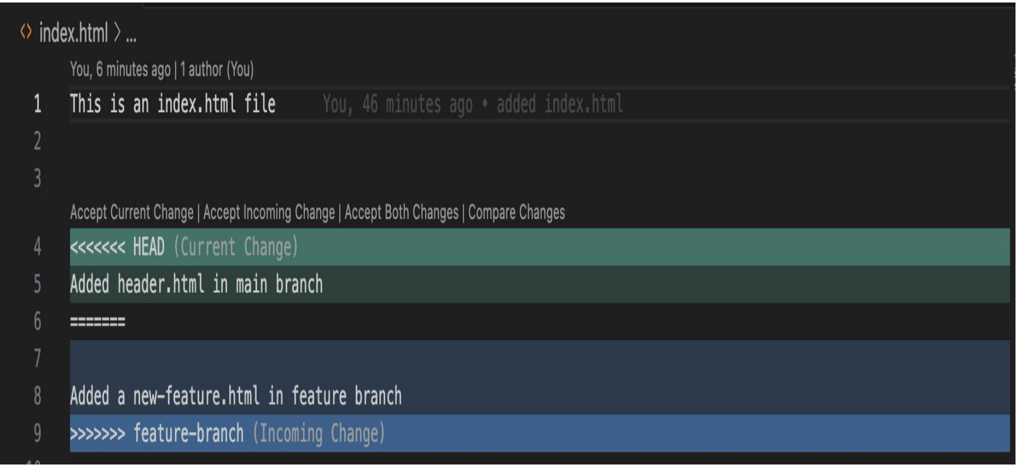
```
~/Documents/Git with Github and CodeCommit/my-new-project git:(main)±1 (0.047s)
git commit -m "modify index.html"
[main 894f2a4] modify index.html
 1 file changed, 3 insertions(+)
```



```
~/Documents/Git with Github and CodeCommit/my-new-project git:(feature-branch)±1 (0.038s)
git commit -m "modify index.html"
[feature-branch a27cd46] modify index.html
 1 file changed, 4 insertions(+)
```



```
~/Documents/Git with Github and CodeCommit/my-new-project git:(main) (0.039s)
git merge feature-branch -m "Merge feature-branch"
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```



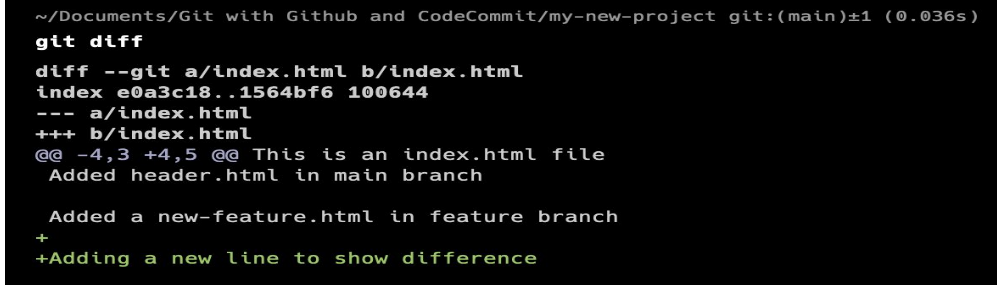
```
index.html > ...
You, 6 minutes ago | 1 author (You)
This is an index.html file      You, 46 minutes ago • added index.html


Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<<< HEAD (Current Change)
Added header.html in main branch
=======

Added a new-feature.html in feature branch
>>>>>>> feature-branch (Incoming Change)
```

3-Way Merge

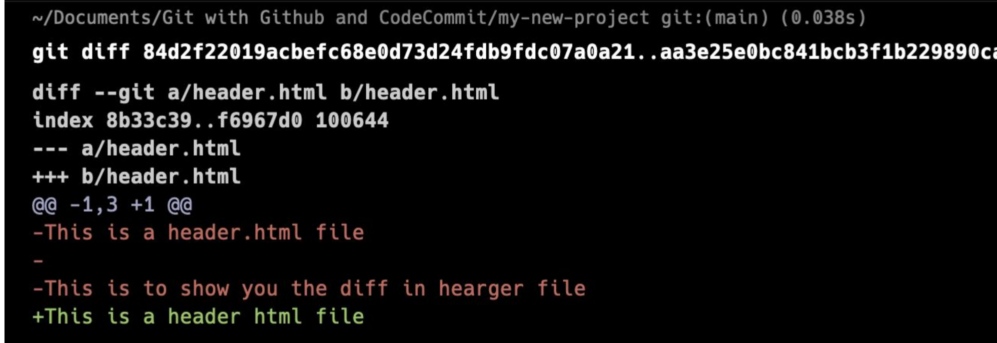
- A 3-way merge is performed when merging two branches with divergent histories. Git identifies a common ancestor commit and uses it as a reference point to integrate changes from both branches.
- In a 3-way merge, Git considers three versions of the code: the common ancestor (base), the current branch being merged into (target), and the branch being merged in (source).
- Git automatically integrates changes from both branches by comparing the changes made in the source and target branches relative to the common ancestor. It then applies these changes to create a new merge commit.
- If Git encounters conflicting changes between the source and target branches, it stops the merge process and asks the user to resolve the conflicts manually. Conflicts occur when changes made in one branch conflict with changes made in the other branch.
- Once all conflicts are resolved, Git creates a new merge commit that incorporates the changes from both branches. This merge commit has two parent commits, representing the branches that were merged.

git “diff”

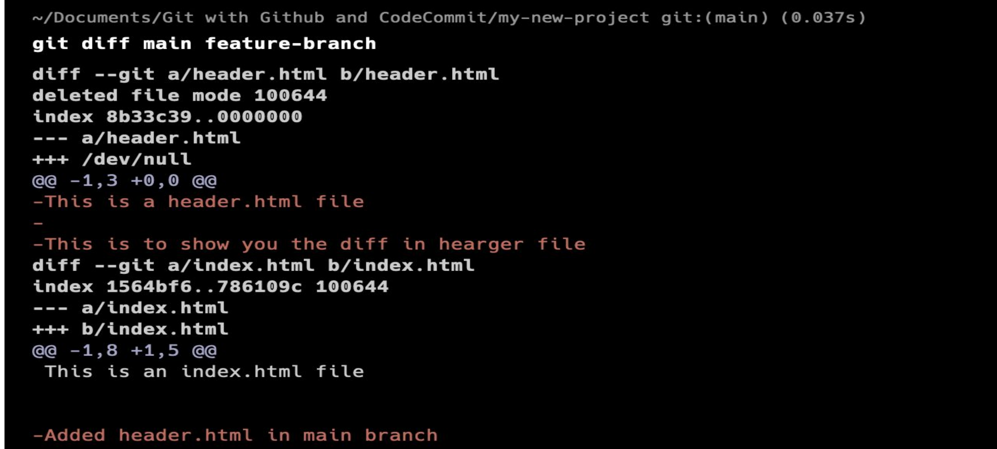
- git diff is a Git command used to show changes between commits, branches, or any two arbitrary points in the Git history. It compares the content of files in different commits or branches and displays the differences.
- When used without any arguments “git diff”, it compares the working directory (the current state of the files) with the index (the staged changes) and shows the differences.
- Adding the --cached flag to git diff (or --staged for older versions of Git) compares the changes that are staged (added using git add) with the last commit. This is useful for reviewing changes before committing.
- You can use git diff followed by two commit hashes to compare the content of files between those two commits. For example, “git diff commit1 commit2” will show the differences between commit1 and commit2.



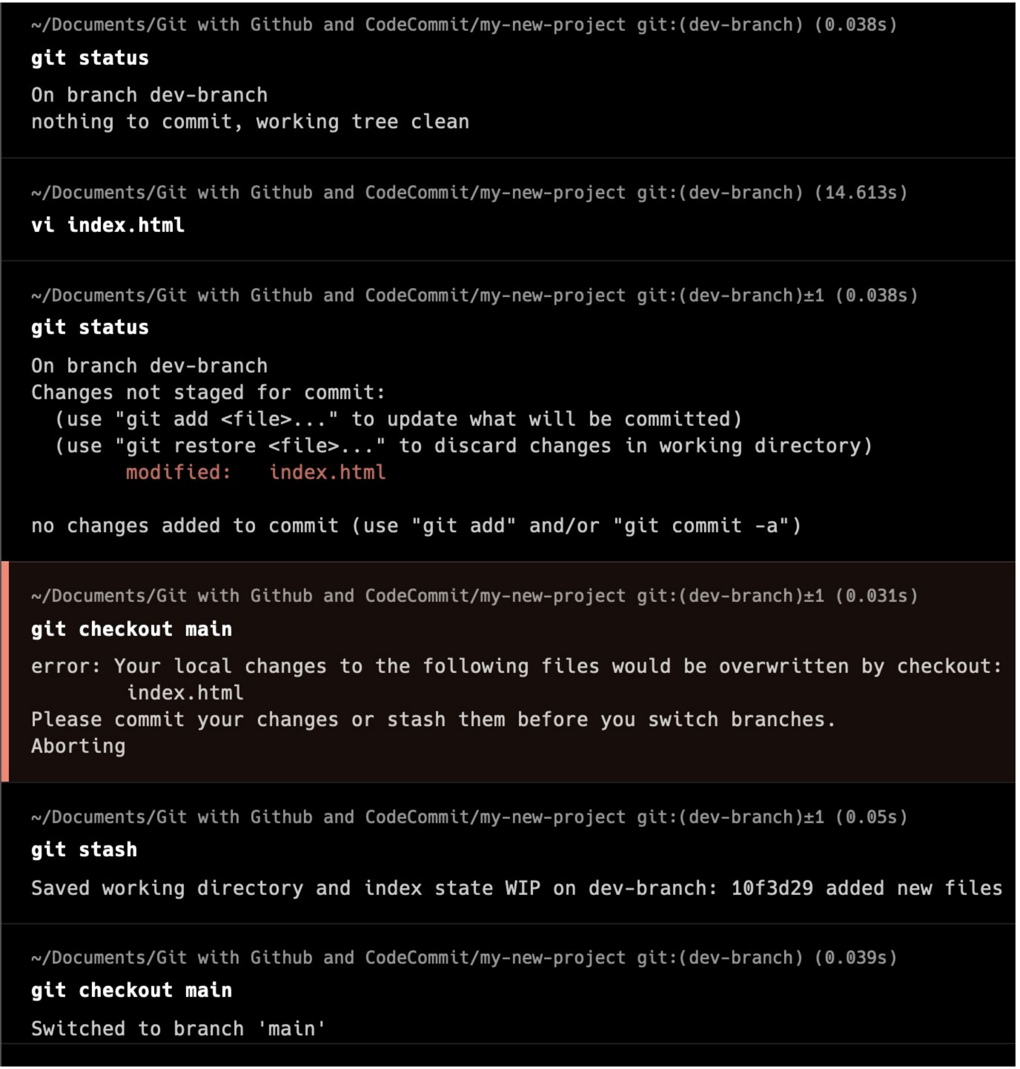
```
~/Documents/Git with Github and CodeCommit/my-new-project git:(main)±1 (0.036s)
git diff
diff --git a/index.html b/index.html
index e0a3c18..1564bf6 100644
--- a/index.html
+++ b/index.html
@@ -4,3 +4,5 @@ This is an index.html file
 Added header.html in main branch

 Added a new-feature.html in feature branch
+
+Adding a new line to show difference
```



```
~/Documents/Git with Github and CodeCommit/my-new-project git:(main) (0.038s)
git diff 84d2f22019acbefc68e0d73d24fdb9fdc07a0a21..aa3e25e0bc841bcb3f1b229890ca
diff --git a/header.html b/header.html
index 8b33c39..f6967d0 100644
--- a/header.html
+++ b/header.html
@@ -1,3 +1 @@
-This is a header.html file
-
-This is to show you the diff in hearger file
+This is a header html file
```



```
~/Documents/Git with Github and CodeCommit/my-new-project git:(main) (0.037s)
git diff main feature-branch
diff --git a/header.html b/header.html
deleted file mode 100644
index 8b33c39..0000000
--- a/header.html
+++ /dev/null
@@ -1,3 +0,0 @@
-This is a header.html file
-
-This is to show you the diff in hearger file
diff --git a/index.html b/index.html
index 1564bf6..786109c 100644
--- a/index.html
+++ b/index.html
@@ -1,8 +1,5 @@
 This is an index.html file


-Added header.html in main branch
```



```
~/Documents/Git with Github and CodeCommit/my-new-project git:(dev-branch) (0.038s)
git status
On branch dev-branch
nothing to commit, working tree clean

~/Documents/Git with Github and CodeCommit/my-new-project git:(dev-branch) (14.613s)
vi index.html

~/Documents/Git with Github and CodeCommit/my-new-project git:(dev-branch)±1 (0.038s)
git status
On branch dev-branch
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")

~/Documents/Git with Github and CodeCommit/my-new-project git:(dev-branch)±1 (0.031s)
git checkout main
error: Your local changes to the following files would be overwritten by checkout:
        index.html
Please commit your changes or stash them before you switch branches.
Aborting

~/Documents/Git with Github and CodeCommit/my-new-project git:(dev-branch)±1 (0.05s)
git stash
Saved working directory and index state WIP on dev-branch: 10f3d29 added new files

~/Documents/Git with Github and CodeCommit/my-new-project git:(dev-branch) (0.039s)
git checkout main
Switched to branch 'main'
```

git “stash”

- “git stash” is a command in Git that allows you to temporarily store changes that are not ready to be committed. It's useful when you need to switch to another branch or work on a different task without committing your current changes.
- “git stash pop” applies the changes from the most recent stash and removes it from the stash stack. This is a convenient way to apply changes and clean up the stash stack in one step.
- If you want to remove specific stashes from the stash stack, you can use “git stash drop” followed by the stash identifier. Alternatively, you can clear all stashes at once using “git stash clear”.
- You can have multiple stashes in Git. Each stash is assigned a unique identifier (stash@{n}, where n is the index of the stash, starting from 0), allowing you to stash changes multiple times and retrieve them individually.
- You can use git stash list to see a list of all stashes in the repository. This command shows the stash ID, the branch from which the stash was created, and a message if provided.
- To apply the changes from a stash, you can use “git stash apply”. By default, it applies the changes from the most recent stash, but you can specify a different stash using its identifier.

git “rebase”

- Rebase vs. Merge: While merging combines the changes of two branches, rebasing reapplies the commits of one branch onto another. This results in a linear history without merge commits, which can make the project history cleaner and easier to understand.
- Similar to merging, rebasing can lead to conflicts when Git tries to apply commits onto a different base. You'll need to resolve these conflicts manually by editing the affected files, marking them as resolved with git add, and continuing the rebase with git rebase --continue.
- One should never run rebase command on the main/master branch. It should be only applied to side branches like dev, feature.
- Before merging a feature branch into the main branch (e.g., master), it's common practice to rebase the feature branch onto the latest commits from the main branch. This ensures that the feature branch incorporates the latest changes and resolves any conflicts before integration.

GitHub

What is GitHub ?

➔ GitHub is primarily used for version control, allowing developers to manage and keep track of changes made to their codebase over time.

➔ Projects on GitHub are stored in repositories. Each repository contains all the files and folders associated with a project, along with the version history.

➔ GitHub facilitates collaboration among developers. Multiple contributors can work on the same project simultaneously, making changes, proposing modifications, and reviewing each other's code.

➔ Developers can create branches within a repository to work on new features or bug fixes without affecting the main codebase. Branches can be merged back into the main branch once changes are complete and reviewed.

➔ GitHub integrates with a wide range of third-party services and tools, including continuous integration (CI) systems, project management platforms, and code quality analysis tools, enhancing the development workflow.

Different Ways of Authenticating GitHub

Basic Authentication:

This method involves providing your GitHub username and password directly through the CLI. While simple, this approach is less secure and not recommended for long-term use. However, it may be suitable for quick, one-off operations.

```
$ git clone
https://username@github.com/user/repo.git
```

Personal Access Tokens:

Personal access tokens are a secure way to authenticate with GitHub via the CLI. You can generate a token with specific permissions (e.g., repo access, user access) and use it instead of your password.

```
$ git clone
https://<username>:<token>@github.com/user/repo.git
```

SSH Keys:

When the branches have diverged, Git performs a 3-way merge. A 3-way merge compares the common ancestor commit of the branches with the latest commits of each branch to reconcile the changes.

```
$ git clone
git@github.com:user/repo.git
```

GitHub CLI (gh):

GitHub CLI is an official command-line tool provided by GitHub. It allows you to perform various GitHub-related tasks from the command line, including authentication.

```
$ gh auth login
```

git “push”

- The primary purpose of git push is to upload local commits to a remote repository, making them accessible to others who have access to that remote repository.
- If a branch is specified in the command (e.g., git push origin main), Git will push the commits from that branch to the corresponding branch on the remote repository.
- When you use git push without specifying the remote and branch names, Git will attempt to push the current branch to its upstream branch (if it's set up). This is called a "tracking branch."
- When you execute git push, Git will prompt you for authentication credentials if necessary, depending on the remote repository's access settings. This could involve entering a username and password or providing an authentication token.

Command : git push <remote> <branch>

Here, <remote> refers to the name of the remote repository (e.g., origin), and <branch> is the name of the branch you want to push changes from.

git “fetch”

- The primary purpose of git fetch is to retrieve new commits and updates from a remote repository to your local repository. However, it doesn't integrate these changes into your working branch.
- Unlike git pull, which automatically merges fetched changes into your local branch, git fetch only downloads the changes to your local repository. This allows you to review the changes before merging them, providing more control over the process.
- git fetch is useful for keeping your local repository synchronized with the remote repository, allowing you to see what work has been done by others without affecting your local branches.
- After fetching changes, you can review the updates using commands like git log or git diff and then merge the changes into your local branch using git merge or git rebase.

Command : git fetch <remote>

Here, <remote> refers to the name of the remote repository from which you want to fetch changes (e.g., origin).

git “pull”

- git pull is a convenient way to update your local repository with changes from a remote repository. It first performs a git fetch to retrieve new commits and updates from the remote, and then it automatically merges those changes into the current branch.
- If you're on a local branch that is tracking a remote branch, you can simply execute `git pull` without specifying the remote and branch names. Git will automatically fetch changes from the remote branch it's tracking and merge them into your local branch.
- Before merging changes with git pull, it's crucial to fetch the latest changes from the remote repository to ensure you're working with up-to-date information.

Command : git pull <remote> <branch>

Here, <remote> refers to the name of the remote repository (e.g., origin), and <branch> is the branch from which you want to pull changes.

git “submodule”

4. Closing

Build confidence around your product or idea by including at least one of the these slides:

➔ **Milestones**
What has been accomplished and what might be left to tackle?

➔ **Testimonials**
Who supports your idea (or doesn't)?

➔ **What's next?**
How can the audience get involved or find out more?

Good luck!

We hope you'll use these tips to go out and deliver a memorable pitch for your product or service!

For more (free) presentation tips relevant to other types of messages, go to heathbrothers.com/presentations

For more about making your ideas stick with others, check out our book!