

Git & Github

Guide

Contents

GIT	2
GIT STATUS	2
GIT INIT	3
GIT ADD	4
GIT COMMIT	5
GIT LOG	6
.GITIGNORE FILE	7
GIT BRANCH	8
WHAT IS HEAD?	10
GIT MERGE	13
GIT DIFF	16
GIT STASH	19
GIT CHECKOUT	21
GIT RESTORE	22
GIT RESET	23
GIT REVERT	24
GIT REBASE	25
Interactive Rebase	28
GIT TAG	34
GITHUB	38

What is GitHub?	38
GIT CLONE	39
Setting SSH Config for GitHub	40
Connecting Local Git Repository with GitHub	42
GIT REMOTE	43
GIT PUSH	45
GIT FETCH	48
GIT PULL	51
What is a Pull Request?	54
Creating and Managing Issues on GitHub	57
FORKING	63

GIT

GIT STATUS

The **git status** command is like a "health check" for your project. When you type **git status** in your terminal, it tells you what's going on in your Git repository. It shows you:

- Which files you've changed but haven't saved yet.
- Which files you've saved (or "staged") but haven't "committed" (or finalized) yet.
- Which branch you're currently working on.

It's a way to keep track of your work and see what you need to do next. Think of it as a "What's happening now?" button for your project.

```
rmian03@RazaXps03 MINGW64 ~/Documents/gitPractice (main)
$ git status
On branch main

No commits yet

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

GIT INIT

The **git init** command is like setting up a new diary for your project. When you type **git init** in your computer's terminal, it creates a new Git repository in the folder you're currently in. This means Git will start keeping track of all the changes you make to the files in that folder.

Example:

Let's say you have a folder named **MyProject** and you want Git to keep track of changes in this folder.

1. Open your terminal and navigate to the **MyProject** folder.
2. Type **git init**.

```
rmian03@RazaXps03 MINGW64 ~/Documents
$ cd MyProject

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject
$ git init
Initialized empty Git repository in C:/Users/rmian03/Documents/MyProject/.git/
```

Now, Git will watch this folder and you can use other Git commands to track changes, save versions, and collaborate with others.

So, in simple terms, **git init** is like saying, "Hey Git, please keep an eye on everything that happens in this folder from now on."

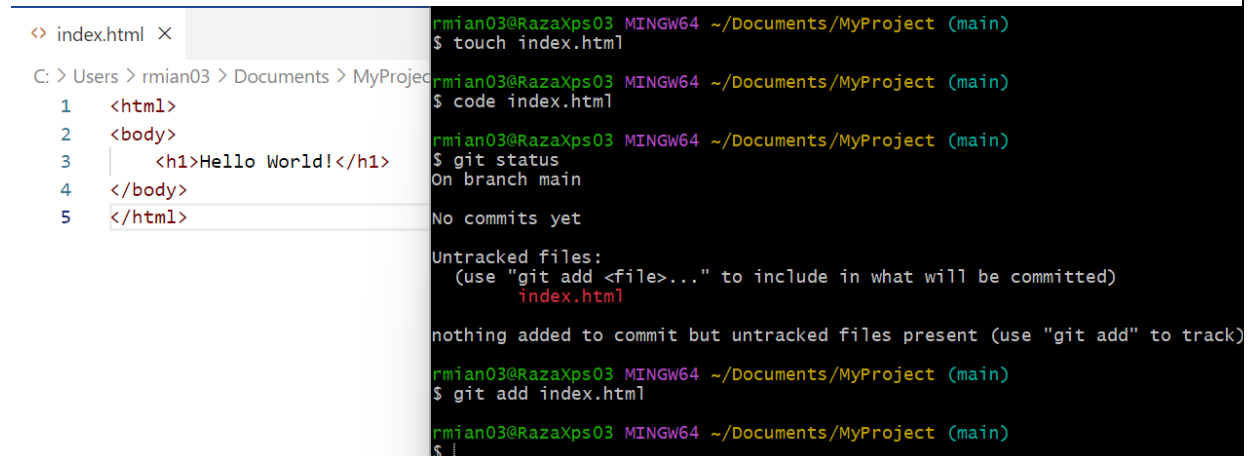
GIT ADD

The **git add** command is like putting your changes into a "save box" before you actually save them. Imagine you're working on a puzzle and you've placed a few pieces. You use **git add** to tell Git, "Hey, I want to keep these puzzle pieces as they are right now."

Example:

You've made changes to a file called **index.html** and you want to prepare it for saving (or "committing" in Git terms).

1. Open your terminal and make sure you're in the folder where your project is.
2. Type **git add index.html**.



The screenshot shows a code editor on the left with the following content for `index.html`:

```
<html>
<body>
  <h1>Hello World!</h1>
</body>
</html>
```

On the right, a terminal window shows the following commands and output:

```
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ touch index.html

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ code index.html

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git status
On branch main
No commits yet

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

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

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git add index.html

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$
```

No output means the command worked. Now, the changes in **index.html** are staged, meaning they're in the "save box" and ready to be committed.

You can also add all changes in all files at once by using:

```
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git add .
```

This tells Git to put all the changed files into the "save box."

So, **git add** is your way of telling Git which changes you want to keep before you actually save them with **git commit**.

GIT COMMIT

The **git commit** command is like hitting the "Save" button on your work. After you've used **git add** to put your changes into a "save box," you use **git commit** to actually save them. This creates a snapshot of your work that you can look back on or share with others.

Example:

You've made changes to a file called **index.html** and you've already used **git add** to stage these changes.

1. Open your terminal and make sure you're in the folder where your project is.
2. Type **git commit -m "Your message here"**.

```
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git commit -m "Added Hello message"
[main (root-commit) b5a8f6e] Added Hello message
1 file changed, 5 insertions(+)
create mode 100644 index.html
```

The **-m** allows you to add a message describing what you did. This is helpful for you and anyone else who looks at the project later.

So, in simple terms, **git commit** is your way of saying, "I want to save these changes and I want to describe what I did as 'Added Hello message'."

Important Notes:

- If you have a lot to say in your commit message and you'd like to write a detailed description, you can do so by simply running **git commit** without the **-m** option. This will open up a text editor where you can write a more detailed message.
- If you made a typo in the commit message or if the message wasn't descriptive enough, you can modify it. However, modifying commit messages should be done carefully, especially for commits that have already been pushed to a remote repository. Here's how you can change the last commit message: `$ git commit --amend`

GIT LOG

The **git log** command is like a history book for your project. It shows you a list of all the "Save" points (or "commits") you've made in your project. Each entry tells you who made the change, when they made it, and what message they left to describe it.

Example:

You've made a few commits and now you want to see the history of your project.

1. Open your terminal and make sure you're in the folder where your project is.
2. Type **git log**.

```
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git log
```

Output:

```
$ git log
commit 20ef84f05e2fc4a6a19fa8202363da2a845688e2 (HEAD -> main)
Author: rmian2 <rmian2@yahoo.com>
Date:   Fri Sep 8 22:16:19 2023 -0400

    Add styles for heading and paragraph elements

commit cf818e59cbf086177b2cbe9fdbad3a1f04f9a86b
Author: rmian2 <rmian2@yahoo.com>
Date:   Fri Sep 8 22:15:14 2023 -0400

    Added style.css file

commit b5a8f6e827b5fe9078a4a0e4d67d6e7397b835de
Author: rmian2 <rmian2@yahoo.com>
Date:   Fri Sep 8 22:02:34 2023 -0400

    Added Hello message
```

This shows you a list of all the commits made, starting with the most recent. You'll see the commit ID, the author, the date, and the message describing what was done.

So, **git log** is your way of looking back to see what changes have been made, who made them, and when they were made. It's like a timeline for your project.

Note: The **git log --oneline** command is a simplified version of **git log**. It shows each commit as a single line, making it easier to get a quick overview of your project's history. Instead of displaying the full commit hash, author, date,

and message, it shows just the short commit hash and the commit message. This is useful for quickly scanning through your commit history without getting bogged down in details. Example:

```
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git log --oneline
6b70182 (HEAD -> main) Add styles for heading and paragraph elements
cf818e5 Added style.css file
b5a8f6e Added Hello message
```

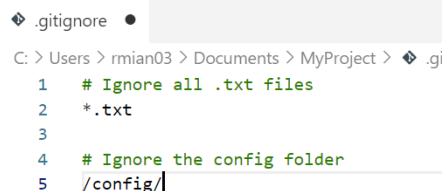
.GITIGNORE FILE

The **.gitignore** file is like a "Do Not Disturb" sign for Git. It tells Git which files or folders to ignore when you're saving (committing) your changes. This is useful for files that you don't want to share with others or keep track of, like temporary files, passwords, or configuration settings.

How to Create and Use .gitignore

1. In your project folder, create a new file and name it **.gitignore**.
2. Open **.gitignore** with a text editor.
3. Add the names of the files or folders you want to ignore, one per line.

For example:



```
.gitignore
1 # Ignore all .txt files
2 *.txt
3
4 # Ignore the config folder
5 /config/
```

4. Save and close the file.

What Happens Now?

- Any **.txt** files in your project will be ignored by Git.
- The **config** folder will also be ignored.

So when you run **git add .** or **git commit**, the files and folders listed in **.gitignore** won't be included.

Example:

Let's say you have the following files in your project:

- **index.html**
- **passwords.txt**
- **config/settings.yaml**

And your **.gitignore** file contains:

- ***.txt**
- **/config/**

When you run **git add .**, only **index.html** will be staged for commit. The **passwords.txt** and **config/settings.yaml** files will be ignored.

In simple terms, **.gitignore** helps you keep your project clean by telling Git what not to include.

GIT BRANCH

In Git, a "branch" is like a parallel universe for your project. It allows you to work on different features, fixes, or experiments without affecting the main (or "master") version of your project. Once you're happy with your changes in a branch, you can merge them back into the main branch to make them official.

Why Use Branches?

Imagine you're writing a book. The main branch is your published book, and branches are like drafts or alternate versions. You can try out different plot twists, add new characters, or change the ending—all without messing up your published book. Once you're happy with a draft, you can merge it into the published book.

Basic Branch Commands:

1. **List all Existing Branches:** To list existing branches in your Git repository, you can use the following command:

```
$ git branch
```

This will show you a list of all local branches. The branch you are currently on will be highlighted and marked with an asterisk (*). Example:


```
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git branch
feature-1
feature-2
* main
```

In this example, there are three branches: feature-1, feature-2 and main. The main branch is currently checked out, as indicated by the asterisk.

2. **Create a New Branch:** To create a new branch, you use the command:

```
$ git branch new-feature
```

This creates a new branch called **new-feature**.

3. **Switch to a Branch:** To start working in that branch, you switch to it using:

```
$ git checkout new-feature
```

or

```
$ git switch new-feature
```

4. **Make Changes:** Now you can make changes, commit them, and they'll only affect this branch.

5. **Switch Back to Main Branch:** To go back to the main branch:

```
$ git checkout main
```

or

```
$ git switch main
```

6. **Merge Changes:** If you're happy with the changes in **new-feature**, you can merge them into **main** with:

```
$ git merge new-feature
```

7. **Delete the Branch:** Once you've merged the changes, you can delete the branch if you want to:

```
$ git branch -d new-feature
```

Important Points:

- **Isolation:** Branches are a way to isolate your changes. You can work on a new feature without affecting the main code.
- **Collaboration:** They are also useful in a team setting. Each team member can work on their own branch without conflicting with others.

- **Flexibility:** You can have multiple branches and switch between them as needed.

In summary, branches make it easier to manage your code, whether you're working alone or with a team. They allow you to work on different tasks simultaneously without interfering with the main codebase.

Some More Branch Related Commands:

Creating and Switching Branch in one Command: Instead of creating a branch and then switching to it, you can also create a new branch and switch to it in one command:

```
$ git switch -c feature-3
```

Renaming a Branch: The **git branch -m** command is used to rename a local branch in a Git repository. The **-m** flag stands for "move" or "rename," and it allows you to change the name of the branch you're currently on or specify which branch to rename.

1. Rename the Current Branch: If you're already on the branch you want to rename, you can simply use:

```
$ git branch -m new_name
```

2. Rename a Different Branch: If you're not on the branch you want to rename, you can specify both the old and new branch names:

```
$ git branch -m oldBranchName newBranchName
```

WHAT IS HEAD?

In Git, the term "HEAD" refers to a reference or pointer to the most recent commit in the currently checked-out branch. In simpler terms, it's like a bookmark that keeps track of where you are in your project's history.

You often see HEAD in the output of commands like **git log**, indicating the most recent commit:

```
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git log
commit cf1c14f7686c1ead93808b937b7fb41230ddc1f8 (HEAD -> main)
Author: rmian2 <rmian2@yahoo.com>
Date: Sat Sep 9 00:34:31 2023 -0400

    Add gitignore file

commit 6b701824e4b285207591f1e2d9b4bb073ba9d876
Author: rmian2 <rmian2@yahoo.com>
Date: Fri Sep 8 22:16:19 2023 -0400

    Add styles for heading and paragraph elements
```

Key Points:

1. **Current Snapshot:** HEAD points to the latest commit in the branch you're working on, representing the most recent state of your project in that branch.
2. **Changing HEAD:** When you switch branches, make new commits, or even go back to an older commit, the HEAD changes to point to that new "current" commit.
3. **Detached HEAD:** Normally, HEAD points to the tip of a branch. However, if you check out a specific commit that is not the latest on a branch, you'll be in a "detached HEAD" state. This means you're not on any branch, and any changes you make won't be saved to a branch unless you explicitly create a new one.

Example:

Let's say you have a branch named **main** with four commits. HEAD will point to the most recent commit, which is commit **cf1c...**:

```

$ git log
commit cf1c14f7686c1ead93808b937b7fb41230ddc1f8 (HEAD -> main)
Author: rmian2 <rmian2@yahoo.com>
Date: Sat Sep 9 00:34:31 2023 -0400

    Add gitignore file

commit 6b701824e4b285207591f1e2d9b4bb073ba9d876
Author: rmian2 <rmian2@yahoo.com>
Date: Fri Sep 8 22:16:19 2023 -0400

    Add styles for heading and paragraph elements

commit cf818e59cbf086177b2cbe9fdbad3a1f04f9a86b
Author: rmian2 <rmian2@yahoo.com>
Date: Fri Sep 8 22:15:14 2023 -0400

    Added style.css file

commit b5a8f6e827b5fe9078a4a0e4d67d6e7397b835de
Author: rmian2 <rmian2@yahoo.com>
Date: Fri Sep 8 22:02:34 2023 -0400

    Added Hello message

```

If you create a new branch called **feature**, HEAD will still point to commit **cf1c...**, but now both **main** and **feature** point to **cf1c...**

```

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git branch feature

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git log
commit cf1c14f7686c1ead93808b937b7fb41230ddc1f8 (HEAD -> main, feature)
Author: rmian2 <rmian2@yahoo.com>
Date: Sat Sep 9 00:34:31 2023 -0400

    Add gitignore file

```

If you make a new commit on **feature**, HEAD moves to point to that commit.

```
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (feature)
$ git log
commit 0e4577509b556a7cfefb68b83dd2144bf6c56e24 (HEAD -> feature)
Author: rmian2 <rmian2@yahoo.com>
Date: Sat Sep 9 00:50:36 2023 -0400

    Add new heading in index.html

commit cf1c14f7686c1ead93808b937b7fb41230ddc1f8 (main)
Author: rmian2 <rmian2@yahoo.com>
Date: Sat Sep 9 00:34:31 2023 -0400

    Add gitignore file

commit 6b701824e4b285207591f1e2d9b4bb073ba9d876
Author: rmian2 <rmian2@yahoo.com>
Date: Fri Sep 8 22:16:19 2023 -0400

    Add styles for heading and paragraph elements

commit cf818e59cbf086177b2cbe9fdbad3a1f04f9a86b
Author: rmian2 <rmian2@yahoo.com>
Date: Fri Sep 8 22:15:14 2023 -0400
```

In summary, HEAD is a way for Git to know what your current working commit is, so it knows what base to use for changes you make next.

GIT MERGE

The **git merge** command is used to combine the changes from one branch into another. Merging is a way to integrate work from different branches, allowing multiple people to collaborate on a project without interfering with each other.

Basic Usage:

To merge changes from a branch named **feature** into the **main** branch, you would do the following:

1. Checkout or Switch to the branch you want to merge into (in this case, **main**):

```
$ git checkout main
```

or

```
$ git switch main
```

2. Run the **git merge** command:

```
$ git merge feature
```

```

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (feature)
$ git switch main
Switched to branch 'main'

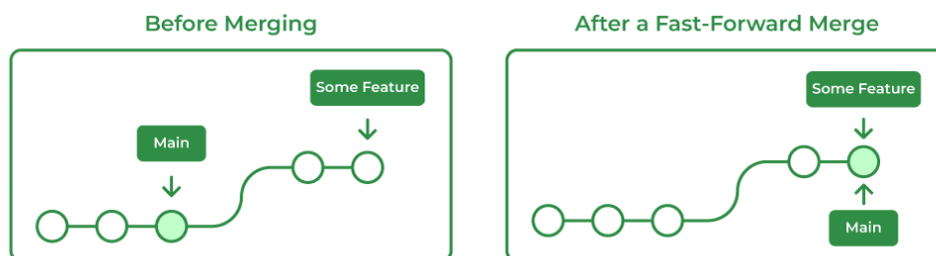
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git merge feature
Updating cf1c14f..0e45775
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git log --oneline
0e45775 (HEAD -> main, feature) Add new heading in index.html
cf1c14f Add gitignore file
6b70182 Add styles for heading and paragraph elements
cf818e5 Added style.css file
b5a8f6e Added Hello message

```

Types of Merges:

1. **Fast-Forward Merge:** If the branch you're merging into hasn't had any new commits since you branched off, Git will simply move the HEAD pointer up to the latest commit of the branch you're merging (See the previous image).



2. **3-Way Merge:** If both branches have new commits like this:

```

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (feature)
$ git log --oneline
81bd926 (HEAD -> feature) Add style for heading three
0e45775 Add new heading in index.html
cf1c14f Add gitignore file
6b70182 Add styles for heading and paragraph elements
cf818e5 Added style.css file
b5a8f6e Added Hello message

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (feature)
$ git switch main
Switched to branch 'main'

rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git log --oneline
097f9b8 (HEAD -> main) Add third heading
0e45775 Add new heading in index.html
cf1c14f Add gitignore file
6b70182 Add styles for heading and paragraph elements
cf818e5 Added style.css file
b5a8f6e Added Hello message

```

Then if we merge both these branches, Git will create a new commit that includes changes from both branches.

```
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git log --oneline
097f9b8 (HEAD -> main) Add third heading
0e45775 Add new heading in index.html
cf1c14f Add gitignore file
6b70182 Add styles for heading and paragraph elements
cf818e5 Added style.css file
b5a8f6e Added Hello message

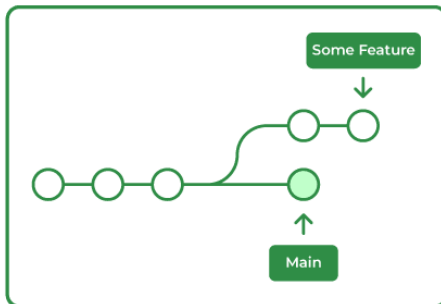
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git merge feature
hint: Waiting for your editor to close the file... unix2dos: converting file c:
users/rmian03/Documents/MyProject/.git/MERGE_MSG to DOS format...

MERGE_MSG
File Edit View

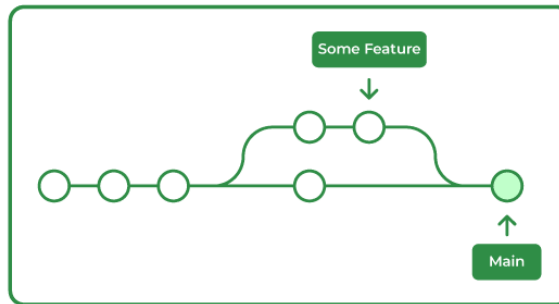
Merge branch 'feature'
# Please enter a commit message to explain
# especially if it merges an updated
#
# Lines starting with '#' will be ignored
# and will be placed below the line
# the commit.
```

```
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git log --oneline
669b799 (HEAD -> main) Merge branch 'feature'
81bd926 (feature) Add style for heading three
097f9b8 Add third heading
0e45775 Add new heading in index.html
cf1c14f Add gitignore file
6b70182 Add styles for heading and paragraph elements
cf818e5 Added style.css file
b5a8f6e Added Hello message
```

Before Merging

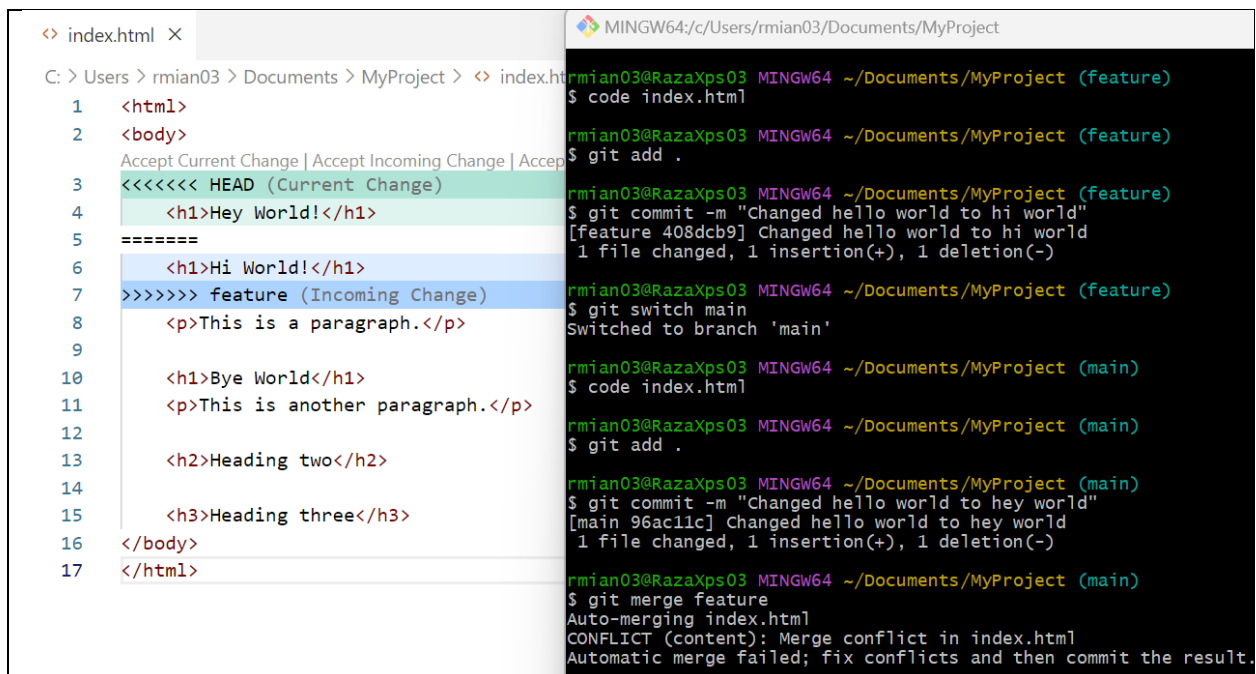


After a 3-way Merge



Merge Conflicts:

Sometimes, the same lines in the same files have been changed in both branches. Git can't automatically decide which changes to keep, leading to a merge conflict.



```
<> index.html X
C: > Users > rmian03 > Documents > MyProject > <> index.html
1 <html>
2 <body>
   Accept Current Change | Accept Incoming Change | Accept
3 <<<<<<< HEAD (Current Change)
4 <h1>Hey World!</h1>
5 =====
6 <h1>Hi World!</h1>
7 >>>>>>> feature (Incoming Change)
8 <p>This is a paragraph.</p>
9
10 <h1>Bye World!</h1>
11 <p>This is another paragraph.</p>
12
13 <h2>Heading two</h2>
14
15 <h3>Heading three</h3>
16 </body>
17 </html>

MINGW64/c:/Users/rmian03/Documents/MyProject
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (feature)
$ code index.html
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (feature)
$ git add .
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (feature)
$ git commit -m "Changed hello world to hi world"
[feature 408dcb9] Changed hello world to hi world
1 file changed, 1 insertion(+), 1 deletion(-)
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (feature)
$ git switch main
Switched to branch 'main'
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ code index.html
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git add .
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git commit -m "Changed hello world to hey world"
[main 96ac11c] Changed hello world to hey world
1 file changed, 1 insertion(+), 1 deletion(-)
rmian03@RazaXps03 MINGW64 ~/Documents/MyProject (main)
$ git merge feature
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

How to Resolve:

1. Open the file with the conflict. You'll see conflict markers (<<<<<<<, =====, >>>>>>>) indicating the conflicting changes.
2. Edit the file to keep the changes you want and remove the conflict markers.
3. Add the resolved file to staging
4. Commit the changes:

In summary, **git merge** is a powerful tool for integrating changes from different branches. It can perform fast-forward merges or 3-way merges, and you may occasionally need to resolve merge conflicts manually.

GIT DIFF

The **git diff** command is used to show differences between commits, branches, or even what's in your working directory compared to your staging area and/or last commit. It's a way to see what has changed in your code, which is especially useful before committing those changes.

Basic Usage:

1. **git diff**

- **What it shows:** This command shows the changes you've made but haven't told Git about yet (i.e., before using **git add**). These changes are in your working directory. In other words, it compares the differences between the working directory (unstaged) and the staging area (staged).
- **Special Case:** If you haven't staged any changes (meaning you haven't run **git add**), this command will show you the changes compared to your last commit.

```
$ git diff
```

2. **git diff HEAD**

- **What it shows:** This command shows all the changes you've made compared to your last commit, whether you've staged them or not.

```
$ git diff HEAD
```

3. **git diff --staged** or **git diff --cached**

- **What it shows:** This command shows what changes you've told Git about and are about to save (commit). In other words, it shows the differences between the staged files and the last commit.

```
$ git diff --staged
```

or

```
$ git diff --cached
```

More Options:

4. **Between Commits:** To see the differences between two specific commits:

```
$ git diff <commit_id_1> <commit_id_2>
```

5. **Between Branches:** To compare two branches:

```
$ git diff branch1..branch2
```

6. **File Specific:** If multiple files were changed, to see changes in a specific file, use:

```
$ git diff path/to/file
```

7. **Summary:** To see a summary of changes (like which files have changed but not the actual line-by-line differences), use:

```
$ git diff --name-only
```

What You'll See:

The output will show you lines that are added or removed. Lines that are added are prefixed with a **+** and are usually shown in green. Lines that are removed are prefixed with a **-** and are usually shown in red.

Example:

```
rmian03@RazaXps03 MINGW64 ~/documents/MyProject (main)
$ git diff
diff --git a/index.html b/index.html
index 95cdb4c..ba68bb7 100644
--- a/index.html
+++ b/index.html
@@ -3,11 +3,12 @@
     <h1>Hello World!</h1>
     <p>This is a paragraph.</p>

-    <h1>Bye World</h1>
     <p>This is another paragraph.</p>

     <h2>Heading two</h2>

     <h3>Heading three</h3>
+    <h4>Heading four</h4>
   </body>
 </html>
\ No newline at end of file
```

This output shows that line `<h1>Bye World</h1>` was removed, and line `<h4>Heading four</h4>` was added in the index.html file.

Understanding the Output:

- **a/index.html b/index.html**: Represents the file that has changed. **a** and **b** indicate the old and new versions respectively.
- **@@ -3,11 +3,12 @@**: This is called a "hunk header". It shows which lines in the old and new files this hunk of changes applies to.

Why is git diff Useful?

- **Review**: Before committing, you can review your changes to make sure you're only committing what you intend to.
- **Debugging**: If something broke, you can use **git diff** to see what changed recently.
- **Collaboration**: When working with others, you can see what changes they have made.

In summary, **git diff** is a powerful tool for viewing changes in your code. It can help you understand what has been modified, added, or deleted, making it easier to track your work and collaborate with others.

GIT STASH

What is git stash?

Imagine you're in the middle of working on a feature, and suddenly you need to switch to another task—maybe fix a bug or check out another branch. But you're not ready to commit your changes yet. What do you do? This is where **git stash** comes in handy. It takes your modified tracked files and staged changes and saves them away for later use, allowing you to switch branches with a clean working directory.

Basic Commands

1. **git stash save** or just **git stash**

- **What it does:** Takes your uncommitted changes (both staged and unstaged), saves them for later, and then reverts your working directory to the last commit.

```
git stash save "Your message here"
```

or simply

```
git stash
```

2. **git stash list**

- **What it does:** Lists all your stashed changes.

```
$ git stash list
```

3. **git stash apply**

- **What it does:** Re-applies the stashed changes, but keeps them in the stash. This is useful if you want to apply the same stashed changes to multiple branches.

```
$ git stash apply
```

4. **git stash pop**

- **What it does:** Re-applies the stashed changes and removes them from the stash.

```
$ git stash pop
```

5. **git stash drop**

- **What it does:** Deletes the latest stash.

```
$ git stash drop
```

6. **git stash clear**

- **What it does:** Deletes all stashes.

```
$ git stash clear
```

Advanced Commands

1. **git stash apply stash@{n}**

- **What it does:** Applies a specific stash from the stash list.

```
$ git stash apply stash@{2}
```

2. **git stash drop stash@{n}**

- **What it does:** Deletes a specific stash.

```
$ git stash drop stash@{2}
```

Summary

- **git stash:** Quickly saves your changes and cleans your working directory.
- **git stash list:** Shows you all your saved stashes.
- **git stash apply:** Puts the stashed changes back into your working directory but keeps them in the stash.
- **git stash pop:** Puts the stashed changes back and removes them from the stash list.
- **git stash drop:** Deletes the latest stash or a specific stash.
- **git stash clear:** Deletes all stashes.

git stash is like a pause button for your changes, allowing you to switch contexts easily. Once you're ready, you can come back and pick up exactly where you left off.

GIT CHECKOUT

The **git checkout** command is used for various tasks like switching branches, reverting changes, and even detaching the HEAD to a specific commit. Here's a breakdown:

1. **git checkout branch_name**

- **What it does:** Switches to a different branch.

```
$ git checkout feature-branch
```

2. **git checkout -b new_branch_name**

- **What it does:** Creates a new branch and switches to it.

```
$ git checkout -b new-feature
```

3. **git checkout commit_hash**

- **What it does:** Detaches the HEAD and moves it to a specific commit. This allows you to explore the project at that point in time.

```
$ git checkout abc1234
```

Note: You can even split off from this specific point in time into another branch and start working on the files as they looked at that point in time. Once you are in detached head and in the location you want to split off, use:

```
$ git switch -c new_branch or $ git checkout -b new_branch
```

4. **git checkout HEAD~n**

- **What it does:** An alternative to using the commit hash. It Moves the HEAD to n commits before the current commit. Useful for quickly navigating the commit history.

```
$ git checkout HEAD~1
```

5. **git checkout HEAD**

- **What it does:** This is essentially a no-op. It will keep you on the current commit and won't change anything.

6. **git checkout -- filename** or **git checkout HEAD filename**

- **What it does:** Reverts changes in an unstaged file to the last committed state.

```
$ git checkout -- myfile.txt
```

GIT RESTORE

The **git restore** command is a more user-friendly alternative to some functionalities of **git checkout**. It was introduced to make it easier to undo changes. The **git restore** command is like an "undo" button in Git. It helps you undo changes in your working directory or staging area without altering your commit history. This makes it a safer option compared to commands like **git reset** that can rewrite history.

Here's how you can use **git restore**:

1. **Unstaged Changes** (**git restore <filename>**): This will undo the changes you made to a file in your working directory, reverting it back to how it looks in the latest commit. It's useful when you've made a change to a file but haven't staged it yet and want to undo it.
2. **Staged Changes** (**git restore --staged <filename>**): This will unstage any changes you've staged, meaning it takes changes out of the staging area but keeps them in your working directory. You can then modify these changes or stage them again later.
3. **Specific Commit** (**git restore --source=<commit_hash> <filename>**): This allows you to restore a file to how it was at a specific commit. It doesn't affect your commit history, just changes the file in your working directory.
4. **Previous Commit** (**git restore --source=HEAD~1 <filename>**): This is similar to the above but uses a relative commit reference (**HEAD~1** means one commit before the latest commit) to restore the file.

Examples:

- **git restore myfile.txt**: This will undo any unstaged changes in **myfile.txt**, making it match the latest commit.

- **git restore --staged myfile.txt**: This will unstage any changes you've staged in **myfile.txt** but keep them in your working directory.
- **git restore --source=4a0fc01 myfile.txt**: This will restore **myfile.txt** to how it was in the commit with the hash **4a0fc01**.
- **git restore --source=HEAD~1 myfile.txt**: This will restore **myfile.txt** to how it was one commit before the latest commit.

Remember, **git restore** is a safe way to undo changes without affecting your commit history. It's particularly useful for local changes that you haven't committed yet.

GIT RESET

The **git reset** command is like a time machine for your Git project. It allows you to go back to a specific point in your commit history and make some changes from there. However, be cautious when using it, as it can rewrite commit history (it undoes previous commits).

Here are some common ways to use **git reset**:

1. **Soft Reset** (**git reset --soft <commit_hash>**): This moves the **HEAD** pointer back to a previous commit but keeps all your changes staged (in the staging area). You can then make more changes and commit them as a new commit.
2. **Mixed Reset** (**git reset --mixed <commit_hash>** or **git reset <commit_hash>**): This is the default option. It moves the **HEAD** back to a previous commit, unstages the changes, but keeps them in your working directory. You'll see the changes as uncommitted local modifications.
3. **Hard Reset** (**git reset --hard <commit_hash>**): This is the most dangerous option. It moves the **HEAD** back to a previous commit and deletes all the changes in the staging area and the working directory. Use this option only when you're sure you want to lose all those changes.

Example:

Let's say you have commits A -> B -> C, and you're currently at commit C.

- **git reset --soft B**: You'll go back to commit B, but the changes from commit C will still be staged.
- **git reset --mixed B** or **git reset B**: You'll go back to commit B, and the changes from commit C will be unstaged but still present in your working directory.
- **git reset --hard B**: You'll go back to commit B, and all changes from commit C will be lost.

Remember, **git reset** is a powerful command. Use it carefully, especially the **--hard** option.

GIT REVERT

The **git revert** command is like a time machine for your code, but with a twist. Instead of erasing history, it adds a new commit that undoes a previous one. This way, you keep a record of what happened, making it a safe operation that doesn't rewrite history.

Here's how it works:

1. **Undo a Single Commit** (**git revert <commit_hash>**): This will create a new commit that undoes the changes made in the commit specified by **<commit_hash>**. Your commit history remains intact, and a new commit is added to reverse the effects of the specified commit.
2. **Undo Multiple Commits** (**git revert OLDEST_COMMIT^..NEWEST_COMMIT**): This will undo a range of commits. Note the **^** after the oldest commit; it's important for including the oldest commit in the range.
3. **Interactive Revert** (**git revert -n <commit_hash>** or **git revert --no-commit <commit_hash>**): This will apply the changes to undo the commit but won't commit them. You can then make additional changes before committing.

Examples:

- **git revert 4a0fc01**: This will create a new commit that undoes the changes made in the commit with the hash **4a0fc01**.

- **git revert HEAD**: This will undo the most recent commit and create a new commit to record this undo action.
- **git revert 4a0fc01^..HEAD**: This will undo all commits from **4a0fc01** to **HEAD** (the latest commit).
- **git revert -n 4a0fc01**: This will apply the undo changes but won't commit them, allowing you to make additional changes before committing.

After running **git revert**, you'll need to push the new commit to your remote repository with **git push** if you want others to see the reverted changes.

Remember, **git revert** is a safe way to undo changes because it doesn't rewrite commit history. It's particularly useful when you've already pushed commits and want to undo them without causing problems for other people who have pulled the same commits.

GIT REBASE

The **git rebase** command is a powerful Git tool that allows you to move or combine commits from one branch to another. It's used to keep your commit history clean, linear, and easy to follow by reapplying changes from one branch on top of another. In contrast to **git merge**, which adds a merge commit to the history, **git rebase** rewrites the commit history to avoid unnecessary merge commits.

What is git rebase?

git rebase is a way to integrate changes from one branch into another by **reapplying** commits. It takes the commits from one branch and moves (or "rebases") them onto the tip of another branch, creating a linear history without merge commits.

In simple terms, **git rebase** allows you to "replay" your commits on top of the latest version of another branch. This results in a cleaner, more linear project history compared to **git merge**.

Basic Syntax

```
git rebase <upstream-branch>
```

- **<upstream-branch>**: The branch onto which you want to rebase your current branch.

Example:

```
git rebase main
```

This command takes the commits from your current branch and re-applies them on top of the latest version of the **main** branch.

What Happens with `git rebase main`?

- **Your feature branch**: Let's assume you are working in your feature branch, `feature/new-feature`, and you've made some commits there.
- **The main branch**: Other team members might have pushed new commits to the main branch, and you want to update your feature branch to include these new changes.

When you run:

```
git rebase main
```

What happens is:

1. **Fetch the latest main branch**: Make sure you have the latest changes from main by running `git fetch origin` and `git pull origin main`.
2. **Rebase**: Git will move your feature branch commits and reapply them on top of the latest version of the **main** branch.
 - Your commits are temporarily "removed," Git applies the changes from **main** first, and then your commits are "replayed" on top of that.

Example:

Let's say the commit history looks like this before rebasing:

```
main:
A---B---C
feature/new-feature:
A---B---C
      \
        D---E---F
```

After running **git rebase main**, it will look like this:

```
main:
A---B---C
feature/new-feature:
A---B---C---D'---E'---F'
```

Now, the D, E, and F commits from your feature branch have been "rebased" onto the tip of the latest main branch, creating a linear history. However, your feature branch has not yet been merged into main.

Important Points:

- After rebasing, you still need to **push** your rebased feature branch (**git push --force** if necessary).
- If you want to merge your changes into main, you still need to do that separately, for example using a **pull request** or a **git merge** command.

To Merge After Rebase:

After rebasing, you can merge your changes into main by:

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

```
git checkout main
```

2. **Merge your feature branch:**

```
git merge feature/new-feature
```

3. **Push to the remote main branch:**

```
git push origin main
```

So, **git rebase main** does not merge changes into main; it simply updates your feature branch by applying its changes on top of the latest main branch. If you want to merge your feature branch into main, you need to perform a separate merge operation.

When to Use git rebase?

1. **Keep a Clean History:** git rebase is often used to avoid merge commits, creating a linear history. This makes it easier to understand the project's progression by looking at the commit log.

2. **Integrate Latest Changes from main:** If other developers have pushed updates to the main branch, you can rebase your feature branch on top of the latest changes, keeping your branch up to date without adding merge commits.
3. **Interactive Rebase to Clean Up Commits:** If your feature branch has multiple commits, some of which are small fixes or tweaks, you can use interactive rebase to squash those commits into a single, meaningful commit. This keeps the commit history clean and makes it easier for others to review your work.

git rebase vs. git merge

- **git merge:** Combines two branches and creates a merge commit if necessary. This keeps both branch histories intact, but it can clutter the history with merge commits.
- **git rebase:** Reapplies commits on top of another branch, creating a linear, cleaner history. However, it rewrites commit history, so it requires careful use.

Interactive Rebase

What is Interactive Rebase?

Interactive rebase (**git rebase -i**) enables you to review and modify the commits you are rebasing. This process opens an editor where you can decide what to do with each commit. It's typically used to:

- Combine multiple small commits into one (squash).
- Edit commit messages.
- Reorder commits.
- Remove unnecessary or unwanted commits.
- Split a single commit into multiple smaller commits.

When Should You Use Interactive Rebase?

Interactive rebase is most useful when:

1. **You want to clean up your commit history** before merging your branch.

2. **You want to rewrite commit messages** to provide clearer context or more detailed explanations.
3. **You want to squash multiple commits** (for example, if you made several small fixes or tweaks) into a single, meaningful commit.
4. **You need to reorder or remove commits** in your branch to make the history clearer.

Interactive rebase helps in maintaining a **clean and linear** Git history, especially in team-based projects where code reviews happen frequently.

How to Use Interactive Rebase: Step-by-Step

1. Initiating Interactive Rebase

The basic syntax for interactive rebase is:

```
git rebase -i <base-branch>
```

- **<base-branch>**: This is the branch or commit where you want to rebase onto. Typically, it's main or master.

Example:

```
git rebase -i main
```

This command tells Git to interactively rebase your current branch onto the main branch.

2. The Rebase Editor

After running **git rebase -i <base-branch>**, Git will open a text editor that shows a list of your commits in reverse order (the oldest commit is at the bottom).

Example editor output:

```
pick 1234567 Add user authentication
pick 89abcde Add error handling
pick fedcba9 Fix typo in authentication module
```

Each line in the editor corresponds to a commit, and each commit is prefixed with the word **pick**. This is where you can tell Git what you want to do with each commit.

3. Editing the Rebase Plan

You can replace the word pick with any of the following commands:

- **pick**: Use the commit as it is (no changes).
- **reword**: Use the commit, but modify the commit message.
- **edit**: Pause the rebase process and allow changes to the commit (e.g., to edit the code in the commit).
- **squash** (or **s**): Combine this commit with the previous one. The commit message can be edited.
- **fixup** (or **f**): Like squash, but the commit message is discarded.
- **drop**: Remove the commit completely.
- **exec**: Run a shell command during the rebase.

4. Common Use Cases in Interactive Rebase

4.1 Squashing Commits

Squashing is useful when you want to combine multiple small commits into one larger, more meaningful commit. Let's say you made three commits that should be combined:

Before interactive rebase:

```
pick 1234567 Add user authentication
pick 89abcde Add error handling
pick fedcba9 Fix typo in authentication module
```

Change the pick for the second and third commits to squash:

```
pick 1234567 Add user authentication
squash 89abcde Add error handling
squash fedcba9 Fix typo in authentication module
```

When you save and exit the editor, Git will combine the three commits into one. You'll then have the option to edit the commit message.

4.2 Rewording a Commit Message

If you want to modify the commit message of a specific commit, change pick to reword:

Before:

```
pick 1234567 Add user authentication
```

```
pick 89abcde Add error handling
```

Change it to:

```
pick 1234567 Add user authentication
```

```
reword 89abcde Add error handling
```

When you save and exit, Git will prompt you to edit the commit message for the selected commit.

4.3 Editing a Commit

You can change the code in a commit during interactive rebase by selecting edit. Let's say you want to modify the code in the second commit:

Before:

```
pick 1234567 Add user authentication
```

```
pick 89abcde Add error handling
```

Change it to:

```
pick 1234567 Add user authentication
```

```
edit 89abcde Add error handling
```

After saving and exiting, Git will stop at that commit and allow you to modify it. Once you've made the changes, you can run:

```
git add .
```

```
git commit --amend
```

```
git rebase --continue
```

This tells Git to update the commit and continue the rebase.

4.4 Reordering Commits

You can reorder commits by simply moving their lines in the editor. For example:

Before:

```
pick 1234567 Add user authentication
```

```
pick 89abcde Add error handling
```

Change it to:

```
pick 89abcde Add error handling
```

```
pick 1234567 Add user authentication
```

When you save and exit, Git will apply the commits in the new order.

4.5 Dropping a Commit

If you want to remove a commit entirely, change pick to drop:

Before:

```
pick 1234567 Add user authentication
pick 89abcde Add error handling
pick fedcba9 Fix typo in authentication module
```

Change it to:

```
pick 1234567 Add user authentication
pick 89abcde Add error handling
drop fedcba9 Fix typo in authentication module
```

When you save and exit, the commit with the typo fix will be removed from the history.

Completing the Interactive Rebase

Once you've finished editing the rebase plan and saved the file, Git will start applying the commits based on your instructions. If conflicts arise during the rebase, Git will pause and allow you to resolve them. After resolving conflicts, you can continue the rebase with:

```
git rebase --continue
```

If you want to abandon the rebase process and go back to the original state, use:

```
git rebase --abort
```

Example Interactive Rebase Workflow

Here's an example of using interactive rebase to clean up a feature branch before merging it into main.

1. Create a new branch and make several commits:

```
git checkout -b feature/new-feature
git commit -m "Initial commit for new feature"
```



```
git commit -m "Fix bug in new feature"  
git commit -m "Improve error handling"  
git commit -m "Fix typo in error message"
```

2. Start an interactive rebase to clean up the commit history:

```
git rebase -i main
```

3. In the editor, change the rebase plan to squash and reword commits:

```
pick 1234567 Initial commit for new feature  
squash 89abcde Fix bug in new feature  
reword fedcba9 Improve error handling  
squash abcd123 Fix typo in error message
```

4. After saving, Git will combine the commits as specified. You'll be prompted to update the commit message for the squashed commits and reword the third commit.

5. Push the changes (force push is necessary after a rebase):

```
git push origin feature/new-feature --force
```

Caution: Rewriting History with Rebase

- **Rewriting commit history:** Rebasing rewrites your branch's commit history, so be careful when using it on branches that are shared with others. If you've already pushed the branch to a remote repository, make sure you understand the consequences of force-pushing.
- **Force pushing after rebase:** Since rebasing changes commit hashes, you'll need to force push (`git push --force`) your changes after the rebase. This overwrites the branch on the remote with your updated history.

Conclusion

Interactive rebase is a powerful tool for cleaning up your commit history and keeping your project organized. It's particularly useful in collaborative environments, where maintaining a clean and linear Git history helps with code reviews and debugging. However, it's important to use it carefully, especially if working with shared branches, as it rewrites history and requires force pushing.

By mastering interactive rebase, you can maintain a professional, readable Git history, making your project's development process smoother and more understandable for everyone involved.

GIT TAG

What Are Git Tags?

A **Git tag** is like a label that points to a specific commit in the repository's history. Unlike branches, which move as you continue working, tags are static. They're typically used to **mark a specific point** in time, such as a version release or a major milestone, like v1.0 or v2.0, and they make it easier to identify and reference important commits.

Types of Git Tags:

1. **Lightweight Tags:** Simple pointers to a specific commit. Think of them as bookmarks.
2. **Annotated Tags:** These contain additional metadata such as the tagger's name, date, and a message describing the tag. Annotated tags are recommended because they carry more information, which is especially useful when you're managing releases.

How to Create Git Tags

1. Creating a Lightweight Tag

A **lightweight tag** is just a pointer to a commit with no extra metadata. Here's how you create one:

```
git tag v1.0
```

This command creates a lightweight tag **v1.0** that points to the **latest commit**. However, lightweight tags aren't recommended for official releases because they lack a description or any identifying metadata.

2. Creating an Annotated Tag

An **annotated tag** includes extra information such as the tagger's name, the date, and a message. It is a much better option for marking official releases.

```
git tag -a v1.0 -m "Version 1.0 stable release"
```

- **-a v1.0**: This creates the tag v1.0.
- **-m "Version 1.0 stable release"**: This adds a message to the tag, similar to a commit message.

3. Tagging a Specific Commit

You can also create a tag for a specific commit, not just the latest one. To do this, you need to find the **commit hash**.

1. Use git log to find the commit hash:

```
git log --oneline
```

2. Tag that specific commit:

```
git tag v1.0 <commit-hash>
```

or

```
git tag -a v1.0 <commit-hash> -m "Version 1.0 stable  
release"
```

This command creates an annotated tag on the specified commit.

Working with Git Tags

1. Listing Tags

To view all tags in your repository, use:

```
git tag
```

This will give you a list of all the tags you've created. If you want to see more details about a specific tag (annotated tags), you can run:

```
git show v1.0
```

This will display details about the tag v1.0, including the commit it points to, the tagger's information, and the message.

2. Searching/Filtering a Tag

We can search for tags that match a particular pattern by using **git tag -l** and then passing a wildcard pattern, for example:

```
git tag -l "*beta*"
```

This will print a list of tags that include "beta" in their name.

3. Pushing Tags to Remote Repository

Tags are created locally by default, meaning they are not automatically pushed to a remote repository like GitHub. To push a specific tag to the remote:

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

If you want to push **all tags** to the remote:

```
git push origin --tags
```

4. Checking Out a Tag

Although tags are static, you can **check out** a tag to look at the state of the repository at that point in time. When you check out a tag, you enter a "detached HEAD" state, meaning any new commits you make won't be associated with a branch unless you create a new branch.

```
git checkout v1.0
```

If you want to make changes from a tag, you should create a new branch:

```
git checkout -b new-branch v1.0
```

This creates a new branch called new-branch from the v1.0 tag.

5. Comparing Tags with Git Diff:

We can compare the changes between different tags by using git diff, for example:

```
git diff v17.0.0 v17.0.1
```

This will show the changes that were made between v17.0.0 and v17.0.1

Managing Tags

1. Renaming Tags

Git doesn't have a direct command to rename a tag, but you can achieve this by deleting the old tag and creating a new one:

1. Delete the old tag locally:

```
git tag -d old-tag
```

2. Create the new tag:

```
git tag new-tag <commit-hash>
```

3. Delete the old tag on the remote:

```
git push origin --delete old-tag
```

4. Push the new tag to the remote:

```
git push origin new-tag
```

2. Deleting Tags

You may want to delete a tag locally or remotely if it was created by mistake.

- **Delete a tag locally:**

```
git tag -d v1.0
```

- **Delete a tag on the remote:**

```
git push origin --delete v1.0
```

3. Pushing and Deleting All Tags

- To **push all tags** to the remote:

```
git push origin --tags
```

- To **delete all tags** on the remote, you will have to do it manually for each tag:

```
git push origin --delete <tag-name>
```

Why Use Git Tags?

- **Versioning:** Tags are an excellent way to keep track of different versions of your software. You can easily refer to v1.0, v1.1, etc.
- **Releases:** Tags are often used to create official releases. On platforms like GitHub, you can use tags to create downloadable releases for your project.
- **Milestones:** Whether you're marking the completion of a major feature or a specific commit in your project's timeline, tags help you keep track of significant changes.

GITHUB

What is GitHub?

GitHub is a web-based platform that uses Git for version control. It allows multiple people to work on projects at once, making it easier to collaborate on projects. GitHub provides a graphical interface for managing repositories, issues, pull requests, and other code-related tasks, making it more user-friendly than Git's command-line interface.

Git vs GitHub?

1. Location:

- Git is a tool that you can use on your local machine.
- GitHub is an online service that hosts your repositories in the cloud.

2. User Interface:

- Git is primarily used through the command line.
- GitHub provides a graphical interface with additional features like pull requests, issues, and project management tools.

3. Collaboration:

- Git is great for version control but doesn't provide built-in features for multiple people to collaborate.
- GitHub makes it easy for teams to collaborate by providing tools for discussion, code review, and more.

4. Additional Features:

- Git focuses solely on version control.
- GitHub offers additional features like GitHub Actions for automation, GitHub Pages for website hosting, and more.

5. Dependency:

- Git doesn't depend on GitHub. You can use Git independently of any online platform.
- GitHub, however, relies on Git as the system for tracking changes in the projects it hosts.

In summary, Git is the tool, and GitHub is the service for projects that use Git. You can use Git without GitHub, but you can't use GitHub without Git.

GIT CLONE

The **git clone** command is used to create a copy of an existing Git repository from a particular source. This command downloads the repository along with all its files, history, and branches to your local machine, allowing you to work on the project without affecting the original.

How to Use git clone?

To clone a repository, you'll need the URL of the repository you want to clone. This URL is often found on the repository's page if it's hosted on a service like GitHub, GitLab, or Bitbucket.

Here's a simple example:

1. Open your command prompt or terminal.
2. Navigate to the folder where you want to clone the repository.
3. Type **git clone** followed by the repository URL.

```
$ git clone https://github.com/username/repository-name.git
```

What Happens When You Clone?

When you run **git clone**, several things happen:

1. A new folder with the repository's name is created in the current directory.
2. The repository is initialized in this new folder.
3. All the files, branches, and commits from the remote repository are downloaded into this folder.

Now, you have a complete local copy of the repository, and you can start making changes, creating new branches, or exploring the code!

Why Use git clone?

- **Ease of Setup:** It's a quick way to start working on an existing project.
- **Isolation:** You can work on your local copy without affecting the original project until you're ready to share your changes.
- **Collaboration:** It allows multiple people to work on the same project by cloning it to their local machines.

In summary, **git clone** is your go-to command for getting a project from a remote repository onto your local machine to start contributing or exploring.

Setting SSH Config for GitHub

SSH (Secure Shell) is a secure method to communicate with remote servers. When you use SSH with GitHub, you establish a secure and encrypted connection between your local machine and GitHub's servers. This way, you don't have to enter your username and password every time you interact with the repository.

Why Use SSH with GitHub?

- **Security:** SSH is encrypted and secure.
- **Convenience:** Once set up, you don't need to enter your username and password for every Git operation.

Steps to Set Up SSH for GitHub

Step 1: Check for Existing SSH Keys

First, check if you already have SSH keys on your computer. Open your terminal and run:

```
$ ls -al ~/.ssh
```

If you see files like **id_rsa** and **id_rsa.pub**, you already have SSH keys. If not, proceed to the next step.

Step 2: Generate a New SSH Key

To generate a new SSH key, run the following command in your terminal:

```
$ ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

When prompted to "Enter a file in which to save the key," press Enter to accept the default location.

Step 3: Start the SSH Agent

Run the following command to ensure the SSH agent is running:

```
$ eval "$(ssh-agent -s)"
```

Step 4: Add SSH Key to the SSH Agent

Add your SSH key to the SSH agent:

```
$ ssh-add ~/.ssh/id_rsa
```

Step 5: Add SSH Key to GitHub Account

1. Open the SSH key with a text editor:

```
$ cat ~/.ssh/id_rsa.pub
```

2. Copy the entire key starting with **ssh-rsa**.
3. Go to GitHub, navigate to Settings > SSH and GPG keys > New SSH key.
4. Paste your copied SSH key into the "Key" field and give it a title.
5. Click "Add SSH Key."

Step 6: Test the Connection

To make sure everything is set up correctly, you can test the SSH connection:

```
$ ssh -T git@github.com
```

If you see a message like "Hi username! You've successfully authenticated...", you're all set!

Summary

Setting up SSH for GitHub may seem like a lot of steps, but it's a one-time setup that makes your life easier and your connections more secure. Once it's done, you can interact with GitHub repositories without having to enter your credentials every time.

Connecting Local Git Repository with GitHub

Scenario 1: You Have a Local Repository and Want to Connect it to GitHub

1. **Initialize Local Git Repository:** If you haven't already, navigate to your project folder in the terminal and run:

```
$ git init
```

This initializes a new Git repository and begins tracking an existing directory.

2. **Add Files to the Repository:** Add the files in your new local repository. This stages them for the first commit.

```
$ git add .
```

The `.` adds all the files in the current directory. You can also add specific files like **git add <filename>**.

3. **Commit the Files:** Commit the files that you've staged in your local repository.

```
$ git commit -m "First commit"
```

This commits the tracked changes and prepares them to be pushed to a remote repository.

4. **Create a New Repository on GitHub:** Go to GitHub and create a new repository. Do not initialize it with a README, .gitignore, or License. You'll connect your local repository to this new remote repository.
5. **Add GitHub Repository as a Remote:** Back in your terminal, add the URL for the remote repository where your local repository will be pushed.

```
$ git remote add origin <remote_repository_URL>
```

The **<remote_repository_URL>** will be provided by GitHub when you create the new repository.

6. **Push Local to Remote:** Push the changes in your local repository to GitHub.

```
$ git push -u origin main
```

This pushes your code to the **main** branch of your GitHub repository.

Scenario 2: You Don't Have a Local Repository and Want to Start Afresh

1. **Create a New Repository on GitHub:** Go to GitHub and create a new repository. You can initialize it with a README, .gitignore, or License if you want.
2. **Clone the Repository:** Open your terminal and navigate to the folder where you want to download (clone) the repository. Then run:

```
$ git clone <remote_repository_URL>
```

The **<remote_repository_URL>** will be provided by GitHub when you create the new repository.

3. **Navigate to the Directory:** Change your current working directory to the new repository's directory that was created on your local machine when you ran the **git clone** command.

```
$ cd <repository_name>
```

4. **Make Changes:** Make some changes to the project files.
5. **Add and Commit Changes:** Stage the changes and commit them.

```
$ git add . git commit -m "Your commit message"
```

6. **Push Changes:** Push the changes to the remote repository on GitHub.

```
$ git push origin main
```

And there you go! You've successfully got your code onto GitHub using both methods.

GIT REMOTE

The **git remote** command is used to manage the set of remote repositories that your local repository is configured to interact with. Essentially, it allows you to connect your local repository to remote repositories hosted on platforms like GitHub, GitLab, etc. Here's a breakdown for beginners:

Basic Usage:

1. **List Remotes:** To see the list of remote repositories that your local repository knows about, you can run:

```
$ git remote -v
```

This will list the URLs of remote repositories along with their short names (usually **origin** for the default one).

2. **Add Remote:** To add a new remote repository, you use:

```
$ git remote add <name> <URL>
```

Here, **<name>** is a short name you give to the remote for easier reference (commonly **origin**), and **<URL>** is the URL of the remote repository (you get this from your GitHub repository page).

3. **Remove Remote:** To remove a remote, you can use:

```
$ git remote remove <name>
```

This will remove the remote repository associated with the name you provide.

4. **Rename Remote:** If you want to rename a remote, you can use:

```
$ git remote rename <old_name> <new_name>
```

This changes the short name of the remote from **<old_name>** to **<new_name>**.

5. **Show Information:** To see more information about a specific remote, you can use:

```
$ git remote show <name>
```

This will display information about the remote repository like fetch and push URLs, tracking branches, etc.

6. **Update Remote URL:** If you need to change the URL of an existing remote, you can use:

```
$ git remote set-url <name> <new_URL>
```

This changes the URL of the remote to **<new_URL>**.

Examples:

- Add a new remote named **origin**:

```
$ git remote add origin  
https://github.com/username/repo.git
```

- List existing remotes:

```
$ git remote -v
```

- Remove a remote named **origin**:

```
$ git remote remove origin
```

Understanding **git remote** is crucial for working collaboratively through a remote repository. It's the bridge between your local development and the collaborative network of repositories.

GIT PUSH

What does git push do?

When you make changes to files in a local repository and commit them using **git commit**, these changes remain in your local Git repository. The **git push** command takes these commits and uploads them to a remote repository, which is typically a server shared by a team. This allows others to access your changes, and it also backs up your work to the remote repository.

Basic Syntax

```
git push <remote> <branch>
```

- **<remote>**: The name of the remote repository. Typically, it's called origin by default.
- **<branch>**: The branch name where you want to push your changes. The default branch is often main or master.

Example:

```
git push origin main
```

This command pushes your local commits on the **main** branch to the **main** branch on the remote repository called origin.

Commonly Used Subcommands and Options

1. Pushing to a Specific Remote and Branch

- You can specify the remote (like GitHub) and the branch you want to push to.

```
git push origin feature-branch
```

This pushes your local **feature-branch** to the **feature-branch** on the remote repository **origin**.

2. Setting Upstream Tracking with --set-upstream

- If you're working on a new branch that doesn't exist on the remote yet, you can push it and set it up to track that remote branch with one command:

```
git push --set-upstream origin new-branch
```

- This tells Git to create the **new-branch** on the remote and also link your local **new-branch** to the remote one, so next time you can just run **git push** without specifying anything.

3. Using -u as a Shortcut for Setting Upstream

- Instead of typing **--set-upstream**, you can use the shorter **-u** option. It behaves the same way:

```
git push -u origin new-branch
```

4. Forcing a Push with --force

- Sometimes your remote branch has changes that conflict with your local branch. If you really need to overwrite the remote branch with your local changes (for example, after a git rebase), you can force push:

```
git push --force
```

Warning: This is a destructive operation as it overwrites the remote history, which can cause problems for others. Use with caution, especially in shared repositories.

5. Pushing All Branches at Once

- If you want to push changes from all your local branches to the remote, you can use:

```
git push --all origin
```

6. Deleting a Remote Branch with --delete

- If you want to delete a branch from the remote repository, you can use:

```
git push origin --delete old-branch
```

This deletes the **old-branch** from the remote repository.

7. Dry Run with --dry-run

- You can use this option to simulate a push and see what would happen without actually making any changes on the remote.

```
git push --dry-run origin main
```

8. Tagging and Pushing Tags

- Tags are often used to mark specific points in history, like a release. To push a tag to a remote:

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

This pushes the tag **v1.0** to the remote. If you want to push all your tags, you can use:

```
git push --tags
```

Typical Workflow in a Workplace

Here's a step-by-step example of how you might use git push in a real work environment:

1. **Create a new branch for a feature or bug fix:**

```
git checkout -b feature/new-feature
```

2. **Make changes and commit them:**

```
git add .
```

```
git commit -m "Implement new feature"
```

3. **Push the new branch to the remote repository:**

```
git push --set-upstream origin feature/new-feature
```

4. **Collaborate with team members:** After pushing, your team can review your changes, and you may make more commits and push them again by just using git push.

5. **After a code review, merge to the main branch and push:**

```
git checkout main
```

```
git merge feature/new-feature
```

```
git push origin main
```

Summary of Important Commands:

- **git push origin main:** Pushes the **main** branch to the **origin** remote.

- **git push --set-upstream origin new-branch**: Pushes the **new-branch** and sets up tracking.
- **git push --force**: Forces a push, overwriting changes on the remote branch.
- **git push --all origin**: Pushes all branches to the remote.
- **git push origin --delete old-branch**: Deletes a branch from the remote.
- **git push --tags**: Pushes all tags to the remote.

GIT FETCH

What does git fetch do?

git fetch retrieves (fetches) the latest commits, branches, and tags from the remote repository and brings them to your local repository. However, it doesn't integrate these updates into your current work or branch. You can think of it as checking for updates, without making any immediate changes to your local files.

It's typically used when you want to see what's changed on the remote repository (e.g., new commits, new branches) before deciding to merge or work with those changes.

Basic Syntax

```
git fetch <remote> <refspec>
```

- **<remote>**: The name of the remote repository (e.g., origin).
- **<refspec>**: Specifies what branches or commits to fetch. You can leave this out to fetch all branches.

Example:

```
git fetch origin
```

This command fetches updates from the remote repository named origin for all branches.

Commonly Used Subcommands and Options

1. Fetching All Branches

- The most common use of **git fetch** is to fetch updates for all branches from a remote repository. This downloads all new changes without merging them into your working branch:

```
git fetch origin
```

2. Fetching Specific Branches

- If you only want to fetch updates for a specific branch (e.g., the main branch), you can specify that branch:

```
git fetch origin main
```

3. Fetching and Pruning with --prune

- Over time, old branches that have been deleted on the remote might still appear in your local copy. Using the **--prune** option helps to clean up these stale references:

```
git fetch --prune
```

This removes references to branches that no longer exist on the remote.

4. Dry Run with --dry-run

- You can check what would be fetched from the remote without actually downloading anything using the **--dry-run** option:

```
git fetch --dry-run origin
```

This gives you a preview of what changes are available without making any changes to your local repository.

5. Fetching All Remotes

- If you have more than one remote repository (e.g., origin for GitHub and upstream for a central repository), you can fetch updates from all remotes at once:

```
git fetch --all
```

This fetches changes from all remote repositories you have set up.

What Happens After Fetching?

After running `git fetch`, the updates are stored in your local repository's tracking branches (e.g., `origin/main` or `origin/feature-branch`). These are branches that reflect the state of the remote branches without affecting your working branch.

You can inspect these changes by switching to the tracking branch or running **git log** to see what commits have been added.

For example, to check what was fetched on the main branch, you can run:

```
git log origin/main
```

If you're ready to incorporate the fetched changes into your local branch, you can do so with:

- **git merge origin/main**: To merge changes from the remote main branch into your current local branch.
- **git pull**: To fetch and merge in one step.

Summary of Important Commands:

- **git fetch origin**: Fetches all updates from the origin remote.
- **git fetch origin main**: Fetches updates only for the main branch.
- **git fetch --prune**: Fetches updates and removes references to branches that no longer exist on the remote.
- **git fetch --all**: Fetches updates from all remotes.
- **git fetch --tags**: Fetches new tags from the remote.
- **git fetch --depth=5**: Fetches only the latest 5 commits from the remote.

git fetch vs. git pull

It's important to remember that **git fetch** just downloads changes but doesn't integrate them into your working branch, whereas **git pull** combines fetching and merging in one step. Many developers prefer to fetch first, review changes, and then merge, which gives them more control over the process.

By mastering **git fetch**, you can stay updated with your team's work without disrupting your current work until you're ready!

GIT PULL

The **git pull** command is another essential command in Git that combines two actions: **git fetch** and **git merge**. This command is used to bring changes from a remote repository into your local branch, making it convenient to stay up to date with other team members' work.

Here's a detailed breakdown of the git pull command and its common sub-commands for a beginner:

What does git pull do?

When you run git pull, Git:

1. **Fetches**: Downloads the latest changes from the remote repository.
2. **Merges**: Automatically merges those changes into your current local branch.

This allows you to quickly update your local branch with any commits made by other developers or team members on the remote branch.

Basic Syntax

```
git pull <remote> <branch>
```

- **<remote>**: The name of the remote repository (typically origin).
- **<branch>**: The name of the branch you want to pull from (typically main, master, or a feature branch).

Example:

```
git pull origin main
```

This command fetches and merges changes from the main branch of the origin remote into your local main branch.

Commonly Used Subcommands and Options

1. Pulling from a Remote Repository

- The most basic use of git pull is to fetch and merge updates from a remote repository to your current local branch:

```
git pull origin main
```

This fetches changes from the main branch of the origin remote repository and merges them into your local main branch.

2. Automatic Merge vs. Rebase with --rebase

- By default, git pull performs a **merge**. However, this can sometimes result in unnecessary merge commits. If you prefer a cleaner history without merge commits, you can use the --rebase option:

```
git pull --rebase origin main
```

This fetches the latest changes from the remote and then rebases your local commits on top of those changes. Rebase rewrites your commit history, keeping it linear without merge commits.

3. Pulling All Branches at Once with --all

- If you need to pull changes from all branches on the remote repository (this is rare but possible in certain workflows), you can use:

```
git pull --all
```

4. Fast-Forward Only with --ff-only

- By default, **git pull** allows Git to create a merge commit if necessary. If you want to only allow fast-forward merges (i.e., the branch must be able to move forward without creating a merge commit), use:

```
git pull --ff-only origin main
```

This ensures that no merge commits are created and the pull is only applied if it can be fast-forwarded.

5. No Commit with --no-commit

- If you want to pull the changes but delay committing them (allowing you to review the changes before committing), you can use:

```
git pull --no-commit origin main
```

This fetches and merges the changes but leaves the working directory in a state where you can review and manually commit the changes later.

6. Dry Run with --dry-run

- To simulate a pull and see what would happen without actually making any changes, use:

```
git pull --dry-run origin main
```

This allows you to preview what would be fetched and merged without altering your local repository.

7. Verbose Output with --verbose

- If you want to see detailed output about what is being fetched and merged, you can use the **--verbose** option:

```
git pull --verbose origin main
```

What Happens During a git pull?

1. **Fetch Phase:** The **git pull** command first downloads (fetches) changes from the remote repository. This could include new commits, updated branches, or new tags.
2. **Merge Phase:** After fetching, Git tries to automatically merge the changes from the remote branch into your current local branch. If there are no conflicts, this process is automatic, and you'll have the latest code merged into your branch.
3. **Conflict Resolution:** If there are conflicting changes (e.g., you and a teammate modified the same line of code), Git will pause the merge process and ask you to resolve the conflicts manually. Once the conflicts are resolved, you can complete the merge with a commit.

Summary of Important Commands:

- **git pull origin main:** Fetches and merges changes from the main branch of the remote repository origin.
- **git pull --rebase origin main:** Fetches and rebases changes, avoiding merge commits and keeping a cleaner history.
- **git pull --ff-only origin main:** Pulls only if the changes can be fast-forwarded; no merge commits are allowed.
- **git pull --no-commit:** Fetches and merges without immediately committing the changes.
- **git pull --all:** Fetches and merges from all branches in the remote repository.

What is a Pull Request?

A **pull request** is a request to merge changes (commits) from one branch into another branch. In most cases, the pull request is created to merge changes from a feature branch or a fork into the main branch (or a branch that contains the production-ready code). A pull request triggers a code review process where other team members or collaborators can:

- Review the changes.
- Provide feedback.
- Discuss potential improvements.
- Approve or reject the proposed changes.

While the term "pull request" is used in GitHub, GitLab uses "merge request," but the concept is the same.

When Do You Use a Pull Request?

1. **When you've completed a feature or bug fix:** After working on a feature or bug fix in a separate branch, you create a pull request to propose merging it back into the main branch or another target branch.
2. **When you want feedback on your code:** If you're working on a complex feature and would like input from your team, you can create a pull request to start a conversation.
3. **For collaborative development:** A pull request allows others to review and contribute to your code before it gets merged into the main codebase, ensuring quality and preventing errors from getting into production.

Pull Request Workflow: Step-by-Step Guide

1. Create a New Branch:

- Before making changes, it's best practice to create a new branch for each feature or bug fix:

```
git checkout -b feature/new-feature
```

2. Make and Commit Your Changes:

- After working on the new feature or fixing the bug, stage your changes and commit them:

```
git add .  
git commit -m "Add new feature"
```

3. Push the Branch to the Remote Repository:

- Push your new branch to the remote repository:

```
git push origin feature/new-feature
```

4. Create a Pull Request:

- Go to the remote Git repository platform (e.g., GitHub) and navigate to the branch you just pushed.
- Click on "New Pull Request" or "Create Merge Request" and select the branch you want to merge your changes into (usually main or master).
- In the pull request form, you can add the following:
 - **Title:** A brief description of the pull request, such as "Add new feature for user profile."
 - **Description:** A detailed explanation of what the pull request does, why it's needed, and any special instructions for the reviewer. This is also where you might list what issues are resolved by the PR.

5. Review by Team Members:

- Team members can now review the changes, add comments, ask for clarification, and suggest improvements. Some common actions during review are:
 - **Commenting:** Reviewers can leave comments on specific lines of code, ask for clarification, or make suggestions for changes.
 - **Requesting Changes:** If the reviewer feels that some parts of the code need modification, they may request changes before approving the pull request.
 - **Approving:** Once the reviewer is satisfied, they can approve the pull request, giving it the green light for merging.

6. Make Requested Changes (If Necessary):

- If reviewers request changes, you can update your code locally and push additional commits to the same branch:

```
git add .  
git commit -m "Address review feedback"  
git push origin feature/new-feature
```

- These new commits will automatically be added to the pull request, and reviewers can re-review the updated code.

7. Merge the Pull Request:

- Once the pull request is approved, it can be merged into the target branch (often main). You have several options:
 - **Merge:** Combines all changes into the target branch with a merge commit.
 - **Squash and Merge:** Combines all the commits in the pull request into a single commit and merges that commit into the target branch. This is useful for keeping the commit history clean.
 - **Rebase and Merge:** Re-applies your branch's changes onto the tip of the target branch, creating a linear commit history without a merge commit.

8. Delete the Feature Branch (Optional but Recommended):

- Once the pull request is merged, it's a good practice to delete the feature branch to keep your repository clean:

```
git branch -d feature/new-feature  
git push origin --delete feature/new-feature
```

Important Options/Features in Pull Requests

1. Assignees and Reviewers:

- You can assign specific team members to review the pull request. This helps to direct the attention of reviewers to the PR.

2. Linked Issues:

- In the description, you can reference issues that are being fixed by this PR using keywords like Fixes #123, where 123 is the issue number. This helps automatically close the issue when the pull request is merged.

3. Draft Pull Requests:

- If your code isn't ready for a full review but you want to share it early for feedback, you can create a **draft pull request**. This indicates that the PR isn't ready to be merged.

4. Pull Request Templates:

- Some repositories have templates that provide a structured form for pull requests, helping ensure all the necessary information (e.g., testing instructions, related issues) is provided.

5. Code Reviews:

- As part of the review process, GitHub and similar platforms allow line-by-line comments, suggestions, and discussions to improve code quality. Reviewers can request changes or approve the pull request after reviewing the code.

Pull Request vs. git pull

It's important not to confuse **pull requests** with the git pull command:

- **Pull Request:** A request to review and merge changes from one branch to another on a Git platform.
- **git pull:** A Git command used to fetch changes from a remote repository and merge them into your current branch.

Pull requests are part of the GitHub, GitLab, or Bitbucket web-based workflow, while git pull is a command that you run locally in your terminal.

Creating and Managing Issues on GitHub

Issues in GitHub are a powerful tool for tracking work, bugs, feature requests, or any type of task related to your project. They allow team members, contributors, or even users to open discussions around specific problems or

ideas. Managing issues efficiently is a key part of project collaboration on GitHub.

Here's a detailed explanation of how you can **create**, **manage**, and make the most of issues within your GitHub repository:

1. Opening Issues for Bugs, Feature Requests, or Tasks

GitHub Issues serve as a central place to track and manage various aspects of a project, from identifying bugs to suggesting new features. Issues can also serve as general tasks or items in a to-do list.

Types of Issues:

- **Bugs:** Issues that describe problems or errors in the codebase.
- **Feature Requests:** Ideas for new features, enhancements, or improvements.
- **Tasks:** General tasks or to-dos that need to be completed, such as updating documentation or writing tests.

Steps to Create an Issue:

1. **Navigate to the "Issues" tab** of your repository:
 - This is where you can view all issues related to the project.
2. **Click "New Issue":**
 - This opens a form where you can describe the issue.
3. **Provide a Title and Description:**
 - The **title** should briefly summarize the issue.
 - The **description** is where you provide detailed information. This can include:
 - What the issue is (e.g., bug description or feature request).
 - Steps to reproduce the bug (if it's an issue).
 - The expected vs. actual behavior.
 - Screenshots, code snippets, or links to external references.
4. **Submit the Issue:**

- Once submitted, the issue will be assigned a unique number (e.g., #23) that can be referenced later in commits, pull requests, or other discussions.

Best Practices for Writing Issues:

- **Be Specific:** Write clear, detailed descriptions that help others understand the problem or request.
- **Provide Context:** Include steps to reproduce bugs, potential solutions, and examples.
- **Use Markdown:** GitHub supports **Markdown** in issues, so you can use formatting (e.g., headers, code blocks, lists) to organize the information clearly.

2. Assigning Issues to Team Members

Assigning issues helps define **ownership** and **responsibility** for different tasks. It also gives visibility into who is working on what, which improves collaboration and prevents duplication of work.

Steps to Assign an Issue:

1. **Open the issue** that you want to assign.
2. **Click the “Assignees” section** on the right-hand side of the issue page.
3. **Select a team member:**
 - You’ll see a list of collaborators in the repository.
 - You can select one or more people to assign to the issue.
 - If the person you want to assign isn’t listed, they need to be invited as a collaborator to the repository first.

Use Case of Assigning Issues:

- **Bug Fix:** If there’s a bug, you may assign it to the developer most familiar with the codebase related to the bug.
- **Feature Development:** Assign the issue to a team member who will be responsible for developing the requested feature.
- **Task Delegation:** Use issues to distribute general tasks (e.g., documentation updates, testing) to different team members.

3. Adding Labels, Milestones, and Projects to Issues

GitHub allows you to further categorize and organize your issues using **labels**, **milestones**, and **projects**. These features are invaluable for managing a project effectively, especially when working with teams or on larger codebases.

Labels:

Labels are used to categorize issues and pull requests. They are color-coded and can represent different aspects of the issue (e.g., bug, enhancement, priority). You can create your own labels or use GitHub's default ones.

- **Examples of Labels:**
 - **bug:** Identifies issues that represent bugs in the system.
 - **enhancement:** Marks feature requests or improvements.
 - **help wanted:** Indicates that contributors are needed to work on the issue.
 - **priority: high:** Marks issues that are critical and need to be addressed urgently.

Steps to Add Labels:

1. **Open the issue.**
2. **Click the "Labels" section** on the right-hand side of the issue page.
3. **Select the appropriate labels** from the dropdown list.
4. **Create custom labels:**
 - You can create custom labels by going to the "Labels" tab under the "Issues" section of your repository.

Use Cases for Labels:

- **Bug Triage:** Separate bugs from feature requests or tasks, and prioritize high-severity issues with priority labels.
- **Feature Requests:** Use labels like "enhancement" to track feature ideas or improvements.

Milestones:

Milestones allow you to group related issues and pull requests to track progress toward a **specific goal** (e.g., a release or a project phase).

- **Creating a Milestone:**

- Go to the "Milestones" tab in the Issues section of your repository and click **New Milestone**.
- Define a title and, optionally, a due date or description for the milestone (e.g., "Version 1.0 Release").

- **Adding Issues to a Milestone:**

- Open an issue, then select the "Milestone" section on the right and choose the appropriate milestone.

Use Cases for Milestones:

- **Release Management:** Create a milestone for each upcoming release (e.g., "v1.0") and add issues that need to be completed before the release.
- **Sprint Planning:** Milestones can be used to track progress in Agile development, grouping issues related to a specific sprint or project phase.

Projects:

GitHub **Projects** provide a Kanban-style board for managing tasks and organizing work. You can group issues, pull requests, and notes in columns like **To Do**, **In Progress**, and **Done**.

- **Creating a Project:**

- Go to the "Projects" tab and click **New Project** to create a board for organizing issues.

- **Adding Issues to a Project:**

- Open an issue, and under the "Projects" section on the right, select the appropriate project board.

- **Using Project Boards:**

- Use the project board to move issues from column to column as they progress (e.g., moving from "To Do" to "In Progress").

Use Cases for Projects:

- **Agile Workflow:** Use GitHub Projects to create sprints or manage tasks in an Agile development process.

- **Organizing Features or Modules:** Use project boards to organize issues related to specific features or sections of the codebase.

Example Workflow for Managing Issues:

1. **Create an Issue:** A team member identifies a bug or requests a new feature and creates a new issue with a clear title and detailed description.
2. **Assign the Issue:** The issue is assigned to the relevant developer or contributor who will work on it.
3. **Label the Issue:** The issue is labeled as a "bug" or "enhancement" and possibly given a priority (e.g., "high").
4. **Add to Milestone:** If the issue is related to a release or sprint, it is added to the appropriate milestone (e.g., "v1.0 Release").
5. **Track in Project:** The issue is added to the project board and placed in the "To Do" column. As the work progresses, the issue moves to "In Progress" and finally to "Done."

Why Use Issues Effectively?

- **Better Collaboration:** GitHub Issues provide a central place for all project-related discussions. Anyone can contribute to discussions, whether it's the development team, contributors, or users.
- **Project Management:** By using labels, milestones, and project boards, you can organize tasks, track progress, and ensure that work is completed in a structured and efficient manner.
- **Accountability:** Assigning issues ensures clear ownership and responsibility for tasks, leading to better productivity and accountability within a team.

Conclusion

GitHub Issues are much more than just a bug tracker. They're a flexible, powerful tool for managing all kinds of project work, from reporting bugs and requesting features to assigning tasks and tracking progress toward goals. By utilizing features like **labels**, **milestones**, and **projects**, you can turn GitHub Issues into an efficient project management system that keeps your team organized and productive.

FORKING

What is a Fork?

In simple terms, **forking** is like making a personal copy of someone else's repository to your own GitHub account. Once a repository is forked, you can freely modify it without impacting the original project.

Forks are commonly used when:

- **Contributing to Open Source:** You want to propose changes or fix bugs in someone else's project.
- **Customizing a Project:** You want to modify an existing repository for your own use while keeping the original project's history.
- **Experimenting Safely:** You can try out new features, refactor code, or make changes in your fork without affecting the original repository.

Key Points about Forks:

- A fork is a **full copy** of the original repository, including all files, branches, and commits.
- A fork exists under your **GitHub account**, and you have full control over it.
- You can keep your fork **in sync** with the original repository by pulling in updates if the original project changes over time.
- Forks are especially useful for **pull requests**, where you suggest changes to the original project.

How Does Forking Work?

1. Creating a Fork

When you fork a repository on GitHub, you create a copy of the original repository in your own account. Here's how you do it:

1. **Navigate to the Repository** you want to fork on GitHub.
2. **Click the "Fork" Button** on the top right of the repository page. This will create a copy of the repository in your own GitHub account.
3. Once the fork is created, it will appear under your own GitHub account, and you can now start making changes.

2. Cloning Your Fork to Work Locally

After forking a repository, if you want to make changes locally on your computer, you'll need to **clone** the fork:

1. **Go to Your Fork** on GitHub.
2. Click the **Code** button and copy the repository's URL.
3. In your terminal or command prompt, use the following command to clone your forked repository to your local machine:

```
git clone <your-forked-repo-url>
```

Example:

```
git clone https://github.com/your-username/repo-name.git
```

3. Making Changes to the Fork

Once you have the forked repository on your machine, you can create a new branch, make changes, and commit those changes:

1. Create a new branch:

```
git checkout -b feature-branch
```

2. Make changes to the code (for example, fix a bug or add a new feature).
3. Commit your changes:

```
git add .
```

```
git commit -m "Add new feature or fix bug"
```

4. Push the changes back to **your fork** on GitHub:

```
git push origin feature-branch
```

Contributing Changes from a Fork (Pull Request)

One of the main reasons to fork a repository is to contribute back to the original project. Once you've made changes in your fork, you can request that the original project owners pull those changes into their repository. This process is known as creating a **pull request**.

Steps to Create a Pull Request:

1. **Push your changes to your fork** (as described above).
2. Go to the **original repository** on GitHub (the one you forked from).

3. GitHub will automatically detect that you've made changes in a forked version and will display a message: **"Compare & pull request"**.
4. Click the **"Compare & pull request"** button.
5. Fill in the details:
 - Provide a **title** and **description** of the changes you've made.
 - The maintainers of the original repository will see your request and can review your code.
6. Click **Create Pull Request**.

Your pull request is now open, and the maintainers of the original repository can **review**, **comment**, and **merge** your changes if they're accepted.

Syncing Your Fork with the Original Repository

If the original repository you forked from is updated (e.g., with new features, bug fixes), your fork won't automatically be updated. You need to **sync your fork** to keep it up to date with the latest changes from the original repository.

Steps to Sync Your Fork:

1. Add the original repository as an **upstream remote**:

```
git remote add upstream https://github.com/original-owner/repository.git
```

2. **Fetch** the latest changes from the original repository:

```
git fetch upstream
```

3. **Merge** the changes into your local branch:

```
git checkout main
```

```
git merge upstream/main
```

4. **Push** the updated changes to your fork:

```
git push origin main
```

By keeping your fork in sync with the original repository, you ensure that you're working on the latest version of the project.

Why Use Forks?

Forking is a core part of GitHub's open-source collaboration workflow. Here are some common scenarios where forks are useful:

1. **Contributing to Open Source:**

- Fork the repository of an open-source project, make improvements or fix bugs, and create a pull request to contribute your changes back to the original project.

2. **Customizing a Project for Personal Use:**

- If you find an open-source project that's almost perfect for your needs but requires a few changes, you can fork it and customize it for your own use without affecting the original.

3. **Experimenting Safely:**

- Forking allows you to experiment with new features or ideas without affecting the original project. If your changes work well, you can submit them for consideration via a pull request.

Summary of Forking Workflow:

1. **Fork** a repository on GitHub.
2. **Clone** your fork to your local machine.
3. **Create a new branch**, make changes, and commit them to your fork.
4. **Push** your changes to your fork on GitHub.
5. **Create a pull request** to contribute your changes back to the original project.
6. **Sync your fork** periodically to keep up with updates from the original repository.

Conclusion

Forking is a powerful tool on GitHub, enabling you to create a copy of any repository under your own account, modify it, and potentially contribute back to the original project through pull requests. It's essential for open-source collaboration and gives you the flexibility to experiment, learn, and build on top of existing projects.

Whether you're contributing to a project or customizing code for your own use, understanding how to work with forks will enhance your productivity and help you engage with the broader GitHub community.