Learn Git

This document provides a quick over view of Git and GitHub. A quick hands-on book for beginner.

Learn Git & GitHub

Contents

[**Learn Git** 3](#_Toc197860266)

[**Step-1:** 3](#_Toc197860267)

[**Step-2:** 3](#_Toc197860268)

[**Step-3:** 3](#_Toc197860269)

[**Step-4:** 3](#_Toc197860270)

[**Step-5:** 4](#_Toc197860271)

[**Step-6:** 4](#_Toc197860272)

[**Step-7:** 4](#_Toc197860273)

[**Step-8:** 4](#_Toc197860274)

[**Step-9:** 5](#_Toc197860275)

[**Step-10:** 5](#_Toc197860276)

[**Step-11:** 5](#_Toc197860277)

[**Step-12:** 5](#_Toc197860278)

[**Step-13:** 6](#_Toc197860279)

[**Step-14:** 6](#_Toc197860280)

[**Step-15:** 6](#_Toc197860281)

[**Step-16:** 6](#_Toc197860282)

[**Step-17:** 7](#_Toc197860283)

[**Step-18: (Git Branching by creating a new branch named feature-login)** 7](#_Toc197860284)

[**Step-19:** 7](#_Toc197860285)

[**Step-20: Change Branch from main to feature-login** 7](#_Toc197860286)

[**Step-21: Create a new file named “GitCommands.docx”** 8](#_Toc197860287)

[**Step-22: The file GitCommads.docx gets added to the remote directory under the newly created branch “feature-login”** 8](#_Toc197860288)

[**Step-23: Creating a Pull Request** 8](#_Toc197860289)

[**Step-24: On merging the feature-login branch with main. You will see the files reflecting on to main branch.** 8](#_Toc197860290)

[**Git Forking:** 9](#_Toc197860291)

[**Step-1: Fork a Repository** 9](#_Toc197860292)

[**Step-2: Clone the Forked Repository Locally** 9](#_Toc197860293)

[**Step-3: Connect to the Original Repository** 9](#_Toc197860294)

[**Step-4: Fetch & Merge Latest changes** 9](#_Toc197860295)

[**Step-5: Make Changes & Push** 10](#_Toc197860296)

[**Step-6: Create a Pull Request** 10](#_Toc197860297)

[**Difference between Git Forking and Git Cloning** 11](#_Toc197860298)

[**Git Forking vs Git Cloning** 11](#_Toc197860299)

[**Difference between Git Pull and Git Fetch** 12](#_Toc197860300)

[**Key Differences Between git pull and git fetch** 12](#_Toc197860301)

[**Git Stash** 13](#_Toc197860302)

[**Step-by-Step Guide to Git Stash** 13](#_Toc197860303)

[**Step-1: Check your Current Changes** 13](#_Toc197860304)

[**Step-2: Stash your changes** 13](#_Toc197860305)

[**Step-3: List Stashed Changes** 13](#_Toc197860306)

[**Step-4: Apply Stashed Changes** 13](#_Toc197860307)

[**Step-5: Remove a Stash** 13](#_Toc197860308)

[**Step-6: Create a Branch from a Stash** 14](#_Toc197860309)

[**Git reflog** 14](#_Toc197860310)

[**Step-by-Step Guide to Git Reflog** 14](#_Toc197860311)

[**Step-1: View Reflog History** 14](#_Toc197860312)

[**Step-2: Recover a Lost commit** 14](#_Toc197860313)

[**Step-3: Reset to a Previous State** 14](#_Toc197860314)

[**Step-4: Clean up Old Reflog Entries** 14](#_Toc197860315)

[**Git rebase** 15](#_Toc197860316)

[**Step-by-Step Guid to Get Rebase** 15](#_Toc197860317)

[**Step-1: Checkout Your Feature Branch** 15](#_Toc197860318)

[**Step-2: Fetch Latest changes from Remote** 15](#_Toc197860319)

[**Step-3: Start the Rebase Process** 15](#_Toc197860320)

[**Step-4: Resolve Merge Conflicts (If any)** 15](#_Toc197860321)

[**Step-5: Push the Updated Branch** 16](#_Toc197860322)

[**Hands-on Example:** 16](#_Toc197860323)

[**What is the difference between Git Rebase and Git Merge** 17](#_Toc197860324)

[**Key Difference between Git rebase and Git Merge** 17](#_Toc197860325)

[**Example Usage** 17](#_Toc197860326)

[**Hands-on Example** 18](#_Toc197860327)

[**GitHub Codespaces** 19](#_Toc197860328)

[**GitHub Codespaces** 19](#_Toc197860329)

[**Local Development** 19](#_Toc197860330)

[**What are the main features and benefits of GitHub Codespaces** 20](#_Toc197860331)

[**How do I set up my own GitHub Codespace** 21](#_Toc197860332)

[**Step-by-Step Installation of GitHub Codespaces** 21](#_Toc197860333)

[**Demo Example: Creating a Simple Web App in Codespaces** 22](#_Toc197860334)

# **Learn Git**

## **Step-1:**

Create a Repository in Git Hub

Let’s Create a Repo named LearnGit

The remote url for the LearnGit is: https://github.com/NaveenSilvester/LearnGit.git

## **Step-2:**

Create a folder in your local computer and name it as LearnGit

## **Step-3:**

Open Command Prompt and navigate into the newly created folder named LearnGit on your local computer

## **Step-4:**

Initiate Git by typing the following command

git init

You will get a message stating

"Initialized empty Git repository in E:/NaveenLearnings2025/LearnGit/.git/"

## **Step-5:**

Adding remote repo to local directory by typing this command on your local computer's command prompt

Note: You need to provide the git URL tha you obtained from Step-1

git remote add origin https://github.com/NaveenSilvester/LearnGit.git

## **Step-6:**

Make a pull request from Remote directory so that local directory is in sync with the latest files in Remote dirctory

git pull origin main

Note: You will get some message like the below:

remote: Enumerating objects: 3, done.

remote: Counting objects: 100% (3/3), done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)

Unpacking objects: 100% (3/3), 863 bytes | 3.00 KiB/s, done.

From https://github.com/NaveenSilvester/LearnGit

 \* branch            main       -> FETCH\_HEAD

 \* [new branch]      main       -> origin/main

## **Step-7:**

 View the Local Directory (LearnGit) and you will see the files from Remote directory copied into your local directory

## **Step-8:**

 Create a new file in your remote Directory and call the file as README.md

 Type some contents into this README.md file

## **Step-9:**

Check the Branch name of the Current Git folder

git branch

Note: You may see the branch as Master

\* master

## **Step-10:**

Rename the branch master to main (so that the names are consistent across Git and Github remote directory)

git branch -M master main

## **Step-11:**

Check the Branch name of the Current Git folder

git branch

Note: You will see the following message

\* main

## **Step-12:**

 Checking the Status of Git local directory

  git status

 You will see a new message:

 On branch master

Changes not staged for commit:

  (use "git add <file>..." to update what will be committed)

  (use "git restore <file>..." to discard changes in working directory)

        modified:   README.md

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

## **Step-13:**

Staging the files

git add \*

## **Step-14:**

 Checking the Status of Git local directory

 git status

 You will see a new message:

 On branch main

Changes to be committed:

  (use "git restore --staged <file>..." to unstage)

        modified:   README.md

## **Step-15:**

Committing the staged files

git commit -m "My First Commit"

Note: You will see a message something like the one below:

[main 3dd7301] My First Commit

 1 file changed, 48 insertions(+), 2 deletions(-)

## **Step-16:**

Pushing the Change to remote directory

git push origin main

Note: You will see some message similar to the below one:

Enumerating objects: 5, done.

Counting objects: 100% (5/5), done.

Delta compression using up to 4 threads

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 994 bytes | 994.00 KiB/s, done.

Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)

To https://github.com/NaveenSilvester/LearnGit.git

   17a9586..3dd7301  main -> main

## **Step-17:**

Login to Git but and get to the repository LearnGit

You will see the README.md file updated into remote directory

## **Step-18: (Git Branching by creating a new branch named feature-login)**

git branch feature-login

## **Step-19:**

Check the branches

git branch

Note: You will see this message on the screen

feature-login

\* main

## **Step-20: Change Branch from main to feature-login**

git switch main

git branch

Note: You will see this message on the screen

\* feature-login

Main

## **Step-21: Create a new file named “GitCommands.docx”**

Add some contents to the file

git add \*

git commit -m “Added to feature-login”

git push origin feature-login

## **Step-22: The file GitCommads.docx gets added to the remote directory under the newly created branch “feature-login”**

You will see on the GitHub that the file GitCommands.docx available under the branch feature-login and not under main

## **Step-23: Creating a Pull Request**

Go to GitHub and create a new pull request and then select the feature branch that you would need to merge and then submit it. A pull request will be created and the system will show if there are any conflicts. If there are conflicts one needs to review and approve the right changes and then click on merge request.

## **Step-24: On merging the feature-login branch with main. You will see the files reflecting on to main branch.**

# **Git Forking:**

Git forking is a way to create your own copy of a repository while keeping a connection to the original one. It allows you to experiment, develop new features, and contribute back to the original project without affecting the main repository.

## **Step-1: Fork a Repository**

1. Log into your GitHub account
2. Navigate to the repository you want to fork.
3. Click the Fork button at the top-right corner of the repository page.
4. GitHub will create a copy of the repository under your account,

## **Step-2: Clone the Forked Repository Locally**

1. Go to your newly forked repository on GitHub
2. Click the Code button and copy the URL
3. Open your terminal and run

git clone <https://github.com/your-username/forked-repo.git>

Replace “your-username” and “forked-repo” with your GitHub username and the repository name.

1. Change into the Cloned directory

cd forked-repo

## **Step-3: Connect to the Original Repository**

To keep your fork updated with the changes from the original repository:

1. Ad the original repository as an upstream remote:

git remote add upstream <https://github.com/original-author/original-repo.git>

1. Verify remotes

git remote -v

## **Step-4: Fetch & Merge Latest changes**

To keep your fork up to date:

1. Fetch latest updates

git fetch upstream

1. Merge the updates

git merge upstream/main

## **Step-5: Make Changes & Push**

1. Create a new branch for your changes

git checkout -b my-new-feature

1. Make your changes, add files

git add .

1. Commit the Changes

git commit -m “Added new feature”

1. Puch changes to your fork

git push origin my-new-feature

## **Step-6: Create a Pull Request**

1. Go to your forked repository on GitHub
2. Click Compare & Pull Request
3. Review your Changes and submit the Pull Request
4. Wait for approval from the original repository maintainer

# **Difference between Git Forking and Git Cloning**

## **Git Forking vs Git Cloning**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Git Forking** | **Git Cloning** |
| **Ownership** | Creates a copy under your GitHub account | Just a local copy without linking to GitHub |
| **Connection** | Maintains a connection with the original repository | No direct connection with the original repository |
| **Collaboration** | Used for contributing to the original repository | Used for personal development or local experimentation |
| **Updating** | Changes can be pulled from the original repository | Requires manual updates from the original repository |

**Use Case Example**

* **Forking:** If you want to contribute to a popular open-source project on GitHub, you fork it, make changes, and send a pull request.
* **Cloning:** If you just want to experiment with an open-source project locally without contributing back, you clone it.

# **Difference between Git Pull and Git Fetch**

Git pull and git fetch are both used to retrieve updates from a remote repository, but they work differently.

## **Key Differences Between git pull and git fetch**

|  |  |  |
| --- | --- | --- |
| **Feature** | **git pull** | **git fetch** |
| **Purpose** | Fetches changes and merges them into the current branch | Retrieves changes but does not merge them |
| **Effect on Local Branch** | Updates the working directory immediately | Does not modify the working directory |
| **Use Case** | When you want to update your local branch with remote changes | When you want to check for updates before merging |
| **Safety** | Can cause merge conflicts if changes are incompatible | Safer, allows reviewing changes before applying |

**Example Usage**

* **Using** git fetch:

git fetch origin

git diff origin/main

This lets you see what has changed before merging.

* **Using** git pull:

git pull origin main

This fetches and merges changes into your current branch.

# **Git Stash**

Git stash is a useful feature that allows you to temporarily save changes that you haven't committed yet. This is helpful when you need to switch branches or work on something else without losing your current progress.

## **Step-by-Step Guide to Git Stash**

## **Step-1: Check your Current Changes**

Before stashing, check your working directory for uncommitted changes:

git status

If you see modified files that you don’t want to commit yet, you can stash them.

## **Step-2: Stash your changes**

To stash your changes, run:

git stash

This will save your uncommitted changes and revert your working directory to the last committed state.

## **Step-3: List Stashed Changes**

You can view all stashed change using:

git stash list

This will show a list of stashed changes, each with an index like stash@{0}, stash@{1}, etc.,

## **Step-4: Apply Stashed Changes**

To bring back your stashed changes, you use:

git stash apply stash@{0}

This applies the latest stash but keep it in the stash list.

## **Step-5: Remove a Stash**

Once you have applied the stash, you can remove it using

git stash drop stash@{0}

Alternatively, if you want to remove all the stashed changes

git stash clear

## **Step-6: Create a Branch from a Stash**

If you want to create a new branch from stash

git stash branch new-feature stash@{0}

This creates a new branch and applies the stash to it.

# **Git reflog**

Git reflog is a powerful tool that records updates to branch tips and references in your local repository. It acts as an undo history, allowing you to recover lost commits or track changes that might not be visible in the regular Git log.

## **Step-by-Step Guide to Git Reflog**

## **Step-1: View Reflog History**

To see the history of HEAD movements:

git reflog

This will display a list of recent actions, including commits, checkouts, resets and merges

## **Step-2: Recover a Lost commit**

If you accidently reset or deleted a commit, find its reference in the reflog:

Identify the commit has and restore it.

git checkout <commit-hash>

## **Step-3: Reset to a Previous State**

To Undo a recent action and return to an earlier state

git reset – hard HEAD@{2}

This moves HEAD back to where it was two actions ago.

## **Step-4: Clean up Old Reflog Entries**

To remove outdated reflog entries

git reflog expire –expire=30.days.ago –all

git gc –prune=30.days.ago

This ensures your repository remains efficient

# **Git rebase**

Git rebase is a powerful command that helps keep a branch up to date with changes from another branch while maintaining a clean commit history. Instead of merging, which creates an additional merge commit, rebase rewrites the commit history by applying your changes on top of the latest updates.

## **Step-by-Step Guid to Get Rebase**

## **Step-1: Checkout Your Feature Branch**

Before rebasing, switch to the branch you want to update.

git checkout feature-branch

## **Step-2: Fetch Latest changes from Remote**

Ensure you have the latest updates from the main branch

git fetch origin

## **Step-3: Start the Rebase Process**

Rebase your feature branch onto the latest main branch

git rebase main

This applies your commits on top of the latest main branch commits.

## **Step-4: Resolve Merge Conflicts (If any)**

If there are conflicts, Git will pause the rebase and show the conflicting files. Fix the conflicts manually, then stage the resolved files

git add .

Continue the rebase process.

git rebase –continue

If you want to abort the rebase

git rebase –abort

## **Step-5: Push the Updated Branch**

Once the rebase is complete, push the updated branch to the remote repository

git push origin feature-branch –force

Since rebase rewrites history, --force is required to update the remote branch

## **Hands-on Example:**

Try rebasing in your LearnGit repository by following these steps:

1. Create a new branch: git checkout -b new-feature
2. Make some changes and commit them.
3. Fetch the latest updates: git fetch origin
4. Rebase onto the main branch: git rebase main
5. Resolve conflicts if needed and continue the rebase.
6. Push the rebased branch using git push origin new-feature --force.

# **What is the difference between Git Rebase and Git Merge**

Git rebase and merge both integrate changes from one branch into another, but they do so in different ways. Here’s a breakdown

## **Key Difference between Git rebase and Git Merge**

|  |  |  |
| --- | --- | --- |
| **Feature** | **git merge** | **git rebase** |
| **Purpose** | Combines changes from two branches, creating a merge commit | Moves commits from one branch onto another, rewriting history |
| **Commit History** | Preserves all commits, including merge commits | Creates a linear history by reapplying commits |
| **Conflict Handling** | Resolves conflicts once during the merge | Resolves conflicts at each commit during rebase |
| **Use Case** | Best for preserving history in collaborative projects | Best for keeping a clean, linear history |

## **Example Usage**

**Using git merge**

git checkout feature-branch

**Git merge main**

Creates a merge commit that ties both histories together

Maintain the original commit structure

**Using git rebase**

Git checkout feature-branch

git rebase main

Moves the feature branch commits on top of the latest main branch

Eliminates unnecessary merge commits for a cleaner history

## **Hands-on Example**

Try both approaches in your LearnGit repository by:

1. Creating a new branch: git checkout -b new-feature
2. Making some changes and committing them.
3. Merging with git merge main or rebasing with git rebase main.
4. Observing the difference in commit history using git log --oneline --graph.

# **GitHub Codespaces**

GitHub Codespaces is a cloud-based development environment that allows developers to set up fully configured coding spaces instantly. It provides a secure and collaborative workspace directly within GitHub, eliminating the need for complex local setups. Each codespace runs in a Docker container on a virtual machine, offering various configurations based on your needs. You can access it via a browser, Visual Studio Code, or GitHub CLI.

GitHub Codespaces and local development each have their strengths, depending on your workflow and needs.

## **GitHub Codespaces**

* **Instant Setup**: No need to install dependencies manually—everything is pre-configured.
* **Cloud-Based**: Runs in a containerized environment, meaning you can access it from anywhere.
* **Collaboration**: Easily share your development environment with teammates.
* **Scalability**: Works well for large projects without worrying about local machine limitations.

## **Local Development**

* **Full Control**: You configure your own environment, choosing your IDE, tools, and dependencies.
* **Performance**: Uses your system’s resources directly, often leading to better performance.
* **Offline Work**: No internet required, making it ideal for remote work or travel

## **What are the main features and benefits of GitHub Codespaces**

GitHub Codespaces offers a powerful cloud-based development environment with several key features and benefits:

**Main Features**

* **Preconfigured Development Environments**: Codespaces provides ready-to-use environments tailored for specific repositories, ensuring consistency across teams.
* **Cloud-Based Resources**: Work remotely on a machine with adequate processing power and storage, eliminating local hardware limitations.
* **Multi-Device Access**: You can access your codespace from any device with a web browser, making it easy to switch between machines.
* **Editor Flexibility**: Use VS Code in the browser, JupyterLab, or the desktop-based VS Code application.
* **Collaboration Tools**: Codespaces supports Live Share for real-time collaboration with teammates.
* **Web App Deployment**: Forward ports and share URLs to test applications before submitting changes.
* **Quickstart Templates**: Reduce setup time when learning new frameworks by using prebuilt templates.

**Benefits**

* **Faster Setup**: No need to install dependencies manually—everything is preconfigured.
* **Improved Collaboration**: Easily share development environments with teammates.
* **Scalability**: Works well for large projects without worrying about local machine limitations.
* **Security**: Built-in security features ensure a safe development environment.
* **Flexibility**: Work on multiple projects or different branches without affecting your local setup.

## **How do I set up my own GitHub Codespace**

Setting up your own GitHub Codespace is straightforward! Here’s how you can do it.

1. **Create a Codespace**

* Navigate to the repository you want to work on in GitHub
* Click the code button, then select Codespaces
* Click New Codespace to start the setup

1. **Configure Your Environment**

* GitHub will automatically set up a development container
* You can customize your environment using a “.devcontainer.json” file.

1. **Start Coding**

Once the Codespace is ready, you can start coding directly in the browser or open it in Visual Studio Code.

1. **Run & Test Your Application**

* Use the integrated terminal to install dependencies and run your application.
* Codespaces supports port forwarding, allowing you to preview web applications.

1. **Commit & Push Changes**

* Once you are done, commit your changes and push them to GitHub

## **Step-by-Step Installation of GitHub Codespaces**

1. **Navigate to Your Repository**

* Go to GitHub and open the repository where you want to create a Codespace.

1. **Open Codespaces**

* Click the Code button at the top of the repository
* Select the Codespaces tab
* Click New Codespace to start the setup

1. **Configure Your Environment**

* GitHub will automatically set up a development container
* You can customized your environment using “.devcontainer.json” file

1. **Start Coding**

* Once the Codespace is ready, you can start coding directly in the browser or open it in Visual Studio Code

1. **Run & Test Your Application**

* Use the integrated terminal to install dependencies and run application
* Codespaces supports port forwarding, allowing you to preview web applications.

1. **Commit & Push Changes**

* Once you are done, commit your changes and push them to GitHub

## **Demo Example: Creating a Simple Web App in Codespaces**

1. **Create a New Repository**

Go to GitHub and create a new repository named “codespaces-demo”

1. **Open Codespaces**

Click the code button and create a new Codespace for the repository

1. **Add a Simple HTML file**

Inside the Codespace, create a new file “index.html” with the following content.

<!DOCTYPE html>

<html>

<head>

<title>GitHub Codespaces Demo</title>

</head>

<body>

<h1>Hello from GitHub Codespaces!</h1>

</body>

</html>

1. **Preview the Web Page**

Use the terminal to start a simple web server

python -m http.server 8080

Click the ports tab in Codespaces and open the forwarded port in the browser.

1. **Commit & Push Changes**

Save the file and commit the changes

git add index.html

git commit -m “Added index.html”

git push origin main

Now, your simple web app is live in GitHub Codespaces