Introduction to Git and GitHub

---

1. Introduction to Version Control

What is Version Control?

Version control is a system that tracks changes to files over time. This is particularly important when working on projects where you need to keep a history of changes, collaborate with others, or revert to previous versions of your work.

Real-World Analogy: Think of version control like editing a document with "Track Changes" turned on. Every time you make a change, it's recorded, and you can review the history, compare versions, and revert if necessary.

Why Use Version Control?

- **Tracking Changes**: Keep a record of all changes made to a project.
- **Collaboration**: Work with others on the same project without overwriting each other's work.
- **Backups**: Have a backup of your project at every stage.

Types of Version Control Systems (VCS)

- **Centralized Version Control (CVC)**: There’s a single central repository that all users pull from and push to (e.g., SVN).
- **Distributed Version Control (DVC)**: Every user has a complete copy of the repository, including its full history (e.g., Git).

2.Understanding Git

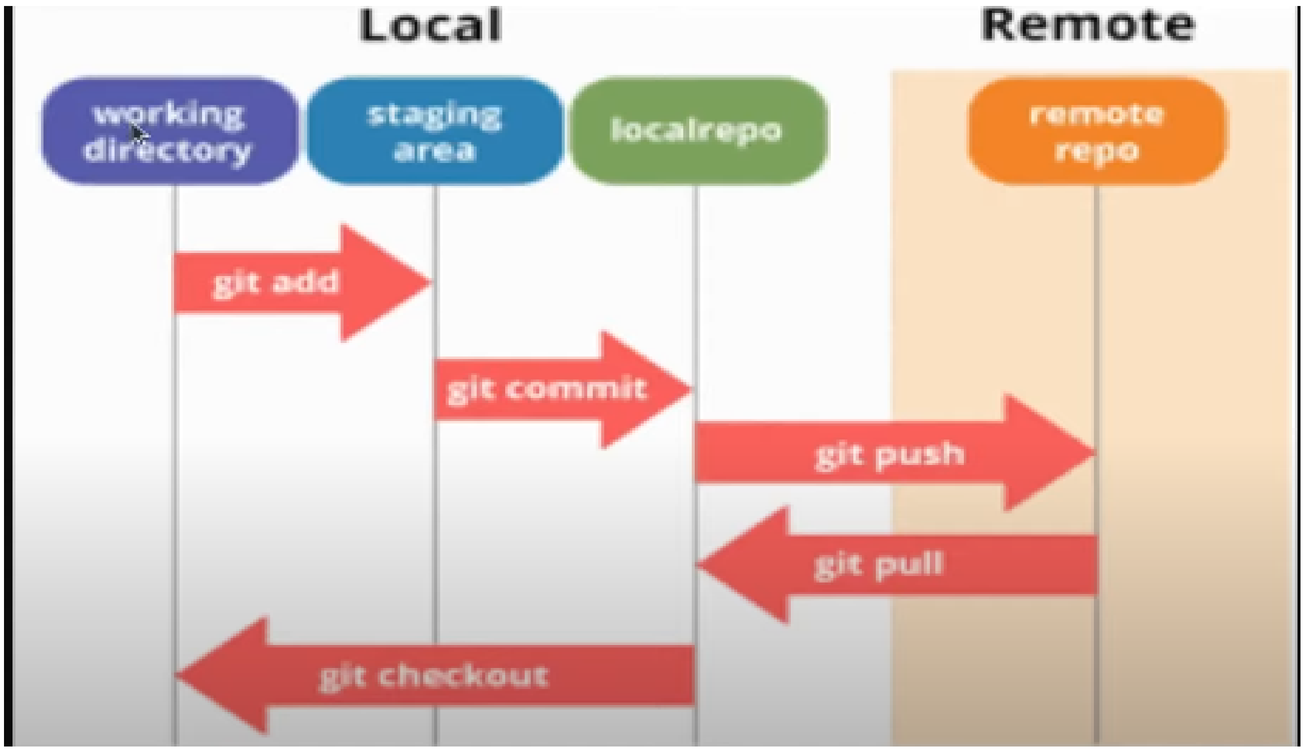
What is Git?

Git is a distributed version control system that allows multiple people to work on a project simultaneously without interfering with each other's work. It's widely used because of its efficiency and flexibility in managing project history.

How Git Works

Git tracks your project history through snapshots. Each time you save your project (commit), Git creates a snapshot of your files at that point. If you make changes, Git allows you to save another snapshot, enabling you to revert back to previous states if needed.

Understanding Key Git Concepts: Working Directory, Staging Area, Local Repository, and Remote Repository



1. Working Directory

The **Working Directory** is the place where you work on your project files. It's the directory on your computer that contains all the files and folders of your project.

Details:

- **What It Is**: The Working Directory is where you make changes to your files. This is the actual directory on your computer where you can open, edit, delete, or add files.
- **Analogy**: Think of the Working Directory as your office desk where all your documents and materials are laid out for you to work on.
- **How It Works**: Any changes you make to files in your project—whether you edit, create, or delete a file—happen in the Working Directory. These changes are not yet tracked by Git until you explicitly tell Git to do so.

Example:

- You create a new file called `index.html` in your project folder. This file is now part of your Working Directory. However, Git doesn't know about it yet. It's like writing on a new piece of paper on your desk; the piece of paper exists, but it's not yet stored in any filing system.

---

2. Staging Area

The **Staging Area** (also known as the Index) is a space where you gather changes that you want to include in your next commit. It acts as a buffer zone between the Working Directory and the Local Repository.

Details:

- **What It Is**: The Staging Area is an intermediary area where Git keeps track of changes that are ready to be committed. You decide which changes go into the Staging Area, which then prepares those changes to be saved in the next commit.
- **Analogy**: Imagine you're preparing a presentation. The Working Directory is where you draft your slides. The Staging Area is like a folder where you organize the finalized slides before presenting them.

- **How It Works**: When you use the `git add` command, you move changes from the Working Directory to the Staging Area. These changes are now ready to be committed, but they are not yet part of the repository history.

Example:

- After editing `index.html`, you want to save your changes. You add it to the Staging Area using `git add index.html`. Now, Git knows you want to include this file in the next commit. It's like placing your finished slide in the folder that you'll use for the presentation.

---

3. Local Repository

The **Local Repository** is the Git repository on your own computer. It stores the complete history of changes to your project, including all the commits.

Details:

- **What It Is**: The Local Repository is a database stored in your project directory (usually in a hidden `.git` folder) that records all your commits. Each commit is a snapshot of your project at a specific point in time.
- **Analogy**: Think of the Local Repository as a filing cabinet in your office where you store all your important documents. Each drawer (commit) contains a complete, organized set of documents (your project files) at a particular point in time.
- **How It Works**: When you use `git commit`, Git takes a snapshot of the files in the Staging Area and saves it in the Local Repository. This snapshot becomes a permanent part of the project's history.

Example:

- You've staged changes to `index.html` and now want to save this change as part of your project history. You use `git commit -m "Added index.html"`. This command moves the staged changes into the Local Repository as a new commit. It's like placing the final, approved version of your presentation into the filing cabinet.

---

4. Remote Repository

The **Remote Repository** is a Git repository that is hosted on a server other than your local machine, such as GitHub, GitLab, or Bitbucket. It allows you to collaborate with others by sharing your code online.

Details:

- **What It Is**: The Remote Repository is like a shared, online filing cabinet where you and your collaborators can store and access the project files. It mirrors the Local Repository but is accessible from anywhere.
- **Analogy**: Think of the Remote Repository as a cloud storage service where you can upload your filing cabinet's contents so that others can access it. It allows multiple people to work on the same project without needing to be in the same place.
- **How It Works**: When you use `git push`, you send your commits from the Local Repository to the Remote Repository. Conversely, `git pull` or `git fetch` allows you to download changes from the Remote Repository to your Local Repository.

Example:

- After committing your changes to `index.html`, you want to share this with your team. You use `git push origin main` to push your changes to the Remote Repository on GitHub. Now, your team members can access the latest version of your project. It's like uploading your finished presentation to a shared drive where everyone on the team can see it and make further changes if needed.

How These Concepts Work Together

1. **Working Directory**: You make changes to files in your Working Directory (e.g., editing `index.html`).
2. **Staging Area**: Once you're satisfied with your changes, you add them to the Staging Area using `git add`.
3. **Local Repository**: You commit the staged changes to the Local Repository with `git commit`, creating a new snapshot of your project.
4. **Remote Repository**: Finally, you push your commits to a Remote Repository with `git push`, making your changes available to collaborators.

Visualizing the Git Workflow

Here's a simple workflow that illustrates these concepts:

1. **Editing Files**:
   - Start by making changes to files in your **Working Directory**.
2. **Staging Changes**:
   - Add the changes you want to commit to the **Staging Area** using `git add`.
3. **Committing Changes**:
   - Commit the staged changes to your **Local Repository** with `git commit`.
4. **Pushing to Remote**:
   - Push your commits to the **Remote Repository** using `git push`, making your changes available to others.

Practical Exercises

Exercise 1: Working Directory

- Open your project folder (Working Directory).
- Create a new file named `hello.txt` and add some text to it.
- Check the status of your files with `git status` to see how Git recognizes this new file.

Exercise 2: Staging Area

- Add `hello.txt` to the Staging Area using `git add hello.txt`.
- Use `git status` again to see how the file is now staged for commit.

Exercise 3: Local Repository

- Commit the staged changes to your Local Repository with `git commit -m "Added hello.txt"`.
- View the commit history using `git log` to see your new commit.

Exercise 4: Remote Repository

- If you have a GitHub account, create a new repository online.
- Link your Local Repository to the Remote Repository using `git remote add origin <repo_URL>`.
- Push your commits to the Remote Repository with `git push origin main`.

3. Introduction to GitHub

What is GitHub?

GitHub is a web-based platform that hosts Git repositories. It provides a collaborative environment where developers can work together on projects, review each other's code, and contribute to open-source projects.

GitHub vs. Alternatives

- **GitHub**: Popular for open-source projects and community collaboration.
- **GitLab**: Offers built-in CI/CD pipelines and is popular for self-hosting.
- **Bitbucket**: Integrated with Atlassian tools like Jira and often used by teams already using those tools.

Why Use GitHub?

- **Collaboration**: Work with others on the same codebase.
- **Open-Source Projects**: Contribute to or maintain open-source software.
- **Portfolio Building**: Showcase your projects and code for potential employers or collaborators.

4. Setting Up Git

Installing Git

- **Windows**:
  1. Download Git from the official website.
  2. Run the installer and follow the instructions.
  3. Open Git Bash to start using Git.
- **macOS**:

Install Git via Homebrew:

```
brew install git
```

Alternatively, download Git from the official website.

- **Linux**:

Install Git using the package manager:

```
sudo apt-get install git  # For Debian/Ubuntu

sudo yum install git  # For Fedora
```

Configuring Git

After installing Git, configure your username and email, which will be associated with your commits:

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

git config --global user.email "your.email@example.com"
```

The --global flag in these commands specifies that the configuration settings should apply to your entire system. This means that the username and email address will be used for all Git repositories on your machine. This is useful if you typically use the same GitHub account for all your projects.
However, if you are working on a specific project that is a one-time thing or a project that isn't associated with your personal GitHub account, you might want to configure Git with different credentials for that particular repository. In such cases, you should omit the --global flag and run the commands within the repository directory:

```
git config user.name "PROJECT_USERNAME"

git config user.email "PROJECT_EMAIL"
```

This way, the username and email address will be set only for the current project and will not affect y

---

To check the currently configured username and email in Git, you can use the following commands:

To check the username:

```
git config --global user.name
```

To check the email:

```
git config --global user.email
```

These commands will display the username and email that are set globally for your Git configuration. If you want to check the username and email for a specific repository (local configuration), you can omit the `--global` flag:

For a specific repository (local configuration):

```
git config user.name

git config user.email
```

Using SSH Keys and Personal Access Tokens (PATs) with GitHub

1. Using SSH Keys

SSH keys provide a secure and convenient way to authenticate with GitHub. Here's how to set up and use SSH keys:

Generating a New SSH Key

1. **Open Terminal:** Use Git Bash (Windows), Terminal (macOS), or your preferred terminal on Linux.

Generate Key Pair:

```
ssh-keygen -t rsa -b 4096 -C "your.email@example.com"
```

2.
   - `-t rsa` specifies the type of key to create.
   - `-b 4096` specifies the number of bits in the key.
   - `-C "your.email@example.com"` adds a label with your email address.

3. **Specify the Path:**

When prompted to enter a file in which to save the key, press Enter to accept the default path or specify a new path:

```
Enter file in which to save the key
(/c/Users/YourUsername/.ssh/id_rsa):
/c/Users/YourUsername/.ssh/my_new_key
```

   -

4. **Enter Passphrase (Optional):**
   - You can add a passphrase for extra security. Press Enter if you prefer not to use one.

Adding the SSH Key to the SSH Agent

Start the SSH Agent:

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

Add Your Key:

```
ssh-add ~/.ssh/my_new_key
```

Adding the SSH Key to Your GitHub Account

Copy the SSH Key to Your Clipboard:

```
cat ~/.ssh/my_new_key.pub
```

Add the Key to GitHub:

- Go to **Settings** > **SSH and GPG keys** > **New SSH key**.
- Paste the copied key and give it a descriptive title.
- Click **Add SSH key**.

Configuring Git to Use the New SSH Key

Open (or Create) the SSH Config File:

```
vim ~/.ssh/config
```

Add Configuration for GitHub:

Type i to get into INSERT Mode in vim

```
  Host github.com

  HostName github.com

  User git

  IdentityFile ~/.ssh/my_new_key
```

Save and Exit:

- Type :wq to save and Exit.

Testing Your SSH Connection

Run the Test Command:

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

Check the Response:

- Successful connection will return a message like: `Hi <username>! You've successfully authenticated.`

2. Using Personal Access Tokens (PATs)

Personal Access Tokens (PATs) are used as a password replacement for Git operations over HTTPS. Here’s how to set up and use PATs:

Generating a Personal Access Token

1. **Go to GitHub:**
   - Log in to your GitHub account.
2. **Navigate to Settings:**
   - Go to **Settings** > **Developer settings** > **Personal access tokens**.
3. **Generate New Token:**
   - Click **Generate new token**.
   - Set a descriptive name and select the scopes (permissions) you need.
   - Click **Generate token**.
4. **Copy the Token:**
   - Copy the generated token immediately. You won’t be able to view it again.

Using the Personal Access Token

Clone a Repository Using PAT:

```
git clone https://github.com/username/repo.git
```

1.
    - When prompted for a password, use your PAT instead of your GitHub password.
2. **Push or Pull Using PAT:**
    - When performing operations like `git push` or `git pull`, use your PAT if prompted for a password.

3. Comparing SSH Keys and Personal Access Tokens (PATs)

Differences

- **Authentication Method:**
    - **SSH Key:** Uses a pair of cryptographic keys (public and private) for authentication.
    - **PAT:** Acts as a password replacement when using HTTPS.
- **Ease of Use:**
    - **SSH Key:** Requires initial setup but provides a passwordless experience once configured.
    - **PAT:** Simpler to set up initially but requires entering the token or using a credential helper for repeated use.
- **Security:**
    - **SSH Key:** Generally more secure and recommended for frequent use.
    - **PAT:** Secure but needs to be managed carefully to avoid exposure.
- **Operation Mode:**
    - **SSH Key:** Used with SSH.

    - **PAT:** Used with HTTPS.

When to Use Each

- **Use SSH Keys:**
    - When you want a passwordless, secure method for frequent interactions with GitHub.
    - Ideal for users who prefer SSH over HTTPS.
- **Use PATs:**
    - When you prefer working over HTTPS or are setting up automated processes and scripts.
    - Useful if you cannot or do not want to configure SSH.

4. Initial Setup for New Users

1. **Configure Git:**

First, set up your Git configuration:

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

git config --global user.email "your.email@example.com"
```

2. **Set Up Authentication:**
    - **If using SSH:** Generate and configure your SSH key as described above.
    - **If using PAT:** Generate a PAT and use it for Git operations over HTTPS.
3. **Choose Based on Workflow:**
    - If you need a secure, passwordless method, set up SSH.
    - If you prefer HTTPS or need to automate tasks, generate and use a PAT.

Summary

- **SSH Keys:** For a secure and convenient passwordless experience, set up SSH keys and add them to your GitHub account.
- **Personal Access Tokens (PATs):** Use PATs as a replacement for passwords when working over HTTPS.
- **Initial Steps:** Configure Git first, then choose and set up your preferred authentication method (SSH or PAT) based on your needs.

This guide should help beginners understand and set up both SSH keys and Personal Access Tokens for working with GitHub.

5. Git Repositories

Creating a Repository

Initialize a Repository:

```
git init
```

1. This command creates a new Git repository in your current directory.

Cloning a Repository:

```
git clone <repository-url>
```

2. This command copies an existing Git repository from a remote server to your local machine.

Understanding Local and Remote Repositories

- **Local Repository**: The repository on your local machine.
- **Remote Repository**: A version of your repository hosted on a server (e.g., on GitHub).

6. Understanding Revert vs. Reset

📝 **Note:** To use `git revert` or `git reset`, you need to know the commit ID (also known as the commit hash) of the commit you want to revert or reset to.

What is `git revert`?

- **Definition**: `git revert` creates a new commit that undoes the changes made by a previous commit. This is a safe way to undo changes because it doesn't alter the commit history.
- **Scenario**: Suppose you made a change in a commit that introduced a bug, and you want to undo this change without rewriting the commit history.
- **Example**:
  ```
  git revert <commit_id>
  ```
- This command creates a new commit that undoes the changes from the specified commit while leaving all other commits intact.
- **Behavior**:
  - **Revert leaves the commit history unchanged**: `git revert` does not remove any commits from the history. Instead, it adds a new commit that effectively undoes the changes made in the specified commit. This means that all commits before and after the commit being reverted remain visible and accessible.

What is `git reset`?

- **Definition**: `git reset` can undo commits by moving the branch pointer backwards and optionally modifying the working directory and staging area.
- **Types of Reset**:
  - `--soft`: Moves the HEAD pointer to the specified commit but leaves the working directory and staging area unchanged.
  - `--mixed` (default): Moves the HEAD pointer to the specified commit and resets the staging area but leaves the working directory unchanged.

    - `--hard`: Moves the HEAD pointer, resets the staging area, and resets the working directory to match the specified commit. This is a destructive operation and can lead to loss of data.
- **Scenario**: You accidentally committed changes too early, and you want to go back and make some changes before committing again.
- **Example**:

```
git reset --soft <commit_id>
```

- This command moves the HEAD pointer to the specified commit and leaves the files as they are, allowing you to amend the changes.
- **Behavior**:
    - **Reset removes commits after the specified commit**: When you reset to a previous commit, `git reset` moves the branch pointer back to that commit, effectively removing the visibility of any commits that occurred after it. However, these "removed" commits are not completely deleted. They are still in the Git history and can be accessed again if needed.
    - **Not Completely Deleted**: The commits that are no longer visible after a `git reset` are still in the repository until they are garbage collected. You can "reset" back to any of those commits if you change your mind or want to recover them. For example, if you reset to commit `A`, and then later decide you want to go back to the last commit before the reset, you can simply reset back to that last commit.

- **Difference between Reset and Revert**:
    - **Reset**: It rolls back the branch pointer to a previous commit, effectively removing the visibility of all subsequent commits from that branch. However, those commits are not permanently deleted and can be recovered by resetting to them.
    - **Revert**: It creates a new commit that undoes the changes made by a specific commit while keeping the commit history intact. Unlike `reset`, revert leaves all other commits visible and does not alter the commit history.

How to Find the Commit ID

Before using `git revert` or `git reset`, you need to know the commit ID of the commit you want to target. To view the commit history and find the commit ID, use:

```
git log
```

This command displays a list of commits, with each commit showing:

- The commit ID (a long alphanumeric string).
- The author of the commit.
- The date of the commit.
- The commit message.

For example:

```
commit d4f8f8d9c15d1c65a9f7f7c9ab1f67c6b5b8d8f9 (HEAD -> main)

Author: Your Name <your.email@example.com>

Date:   Thu Aug 18 12:34:56 2024 +0000


    Your commit message here
```

In this example, the commit ID is `d4f8f8d9c15d1c65a9f7f7c9ab1f67c6b5b8d8f9`. You would use this ID with `git revert` or `git reset` to specify the commit you want to target.

Working with Branches

What Are Branches?

Branches in Git allow you to create a separate line of development within your repository. This enables you to work on features, fix bugs, or experiment with new ideas without affecting the main codebase. Branches help in keeping your development organized and prevent unfinished or experimental code from being included in the main project.

Common Branching Commands

- **`git branch <branch-name>`**: Creates a new branch with the specified name.

Example:

```
git branch feature-branch
```

  - This command creates a new branch called `feature-branch`.

- **`git checkout <branch-name>`**: Switches to the specified branch.

Example:

```
git checkout feature-branch
```

  - This command switches your working directory to the `feature-branch`.

- **`git merge <branch-name>`**: Merges the specified branch into the current branch.

Example:

```
git merge feature-branch
```

  - This command merges the changes from `feature-branch` into the branch you are currently on (e.g., `main`).

- **`git branch -d <branch-name>`**: Deletes the specified branch.

Example:

```
git branch -d feature-branch
```

- This command deletes the `feature-branch` from your local repository.

`git switch`: A Modern Command for Switching Branches

`git switch` is a newer command introduced in Git 2.23 as a more intuitive way to switch between branches or create and switch to new branches. While `git checkout` is still widely used, `git switch` offers a more focused approach for branch switching and creation.

- **Basic Usage**:

Switch to an Existing Branch:

```
git switch <branch-name>
```

Example:

```
git switch feature-branch
```

- This command switches to the `feature-branch`.

Create and Switch to a New Branch:

```
git switch -c <new-branch-name>
```

Example:

```
git switch -c new-feature-branch
```

- This command creates a new branch called `new-feature-branch` and switches to it immediately.

Options and Flags for `git switch`:

- **`-c` or `--create`**: Creates a new branch and switches to it.

Example:

```
git switch -c new-branch
```

  - This creates a new branch named `new-branch` and switches to it.

- **`-`**: Switches back to the previous branch you were on.

Example:

```
git switch -
```

  - This command switches to the branch you were previously on.

- **`--detach`**: Switches to a branch in a "detached HEAD" state, meaning you can explore the branch without creating a new commit on it.

Example:

```
git switch --detach feature-branch
```

  - This command switches to `feature-branch` in detached HEAD mode.

- **`-f` or `--force`**: Forces the switch to another branch, discarding any local changes that would be lost due to the switch.

Example:

```
git switch -f feature-branch
```

  - This command forces the switch to `feature-branch`, discarding any unsaved changes.

When to Use `git switch` vs. `git checkout`

- **Use `git switch`** when your primary goal is to switch branches or create and switch to a new branch. It's more intuitive for these tasks and avoids the complexity of `git checkout`.
- **Use `git checkout`** for other tasks, such as checking out specific files or commits. `git checkout` is more versatile but also more complex.

Example Workflow with `git switch`:

Create and Switch to a New Branch:

```
git switch -c feature-branch
```

1. This creates a new branch called `feature-branch` and switches to it.

Make Changes and Commit:

```
# Make your changes

git add .

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

2.

Switch Back to the Main Branch:

```
git switch main
```

3.

Merge the Feature Branch:

```
git merge feature-branch
```

4.

Delete the Feature Branch:

```
git branch -d feature-branch
```

5.

By using `git switch`, you can streamline your branch management tasks, making the process more intuitive for beginners and experienced users alike.

Understanding Git Merge, Rebase, and Squash: Differences, Usage, and How-To

In Git, merging, rebasing, and squashing are essential techniques for managing changes across different branches of your project. Each method has its use cases, advantages, and potential drawbacks. Below, we'll explore what each of these techniques does, how they differ, when to use them, and how to execute them.

---

1. Git Merge

What Is Git Merge?

- **Merging** combines the changes from one branch into another, usually integrating changes from a feature branch into the main branch (e.g., `main` or `master`). This process creates a new commit called a "merge commit" that ties the histories of the two branches together.

How It Works:

- When you merge, Git takes the commits from the source branch (e.g., a feature branch) and combines them into the target branch (e.g., `main`). The histories of both branches remain intact, and the merge commit has two parent commits—one from each branch.

Why and When to Use Git Merge:

- **Why**: Use merging when you want to integrate changes from one branch into another without altering the history of either branch. Merging is straightforward and preserves the complete history of all branches.
- **When**: It's best used when you want a clear and complete history, such as in collaborative projects where tracking the origin of changes is important.

How to Use Git Merge:

Checkout the branch you want to merge into:

```
git checkout main
```

Merge the feature branch into the main branch:

```
git merge feature-branch
```

Resolve any merge conflicts (if they arise), then commit the merge.

Example:

- You have a branch called `feature-login` and want to merge it into `main`. After running `git merge feature-login`, a new merge commit appears in `main`, combining all the changes from `feature-login`.

---

2. Git Rebase

What Is Git Rebase?

- **Rebasing** re-applies the commits from one branch onto another branch, effectively "replaying" the changes. This process rewrites the project history, creating a linear sequence of commits.

How It Works:

- When you rebase, Git takes the commits from your current branch and applies them one by one on top of another branch (typically `main`). This results in a linear history without any merge commits, making the commit history cleaner.

Why and When to Use Git Rebase:

- **Why**: Use rebasing to create a cleaner, more linear project history. It avoids the clutter of merge commits, making it easier to follow the history of changes.
- **When**: It's ideal when you want to integrate changes from `main` into your feature branch before merging it back, or when working on a solo project where you want a neat history.

How to Use Git Rebase:

Checkout the branch you want to rebase:

```
git checkout feature-branch
```

Rebase your branch onto another branch (e.g., `main`):

```
git rebase main
```

Resolve any rebase conflicts (if they arise), then continue the rebase:

```
git rebase --continue
```

Example:

- You have a branch called `feature-login` and want to rebase it onto `main` to integrate the latest changes. After running `git rebase main`, your branch will contain all the commits from `main`, followed by your `feature-login` commits in a linear sequence.

Important: Be cautious when rebasing, especially on shared branches, as it rewrites commit history. Avoid rebasing commits that have already been pushed to a shared repository.

---

3. Git Squash

What Is Git Squash?

- **Squashing** combines multiple commits into a single commit. This is often done during a rebase to clean up a branch's history before merging it into another branch.

How It Works:

- When you squash commits, Git consolidates a series of commits into one, which can be useful to reduce noise in the commit history. This results in a single commit that contains all the changes from the squashed commits.

Why and When to Use Git Squash:

- **Why**: Use squashing to clean up your commit history, especially if you've made many small or experimental commits that you want to group into one logical change.
- **When**: It's best used before merging a feature branch into `main` to present a concise history, or during a rebase to tidy up a branch.

How to Use Git Squash:

Start an interactive rebase:

```
git rebase -i HEAD~n
```

1. Replace `n` with the number of commits you want to squash.
2. **In the editor that opens, change `pick` to `squash`** (or `s`) for the commits you want to squash.
3. **Save and close the editor** to proceed with the rebase.

Example:

- You have 5 commits on your `feature-login` branch, but 3 of them are small fixes. You can squash the 3 small commits into the main commit, resulting in just 3 commits instead of 5. This is done via `git rebase -i HEAD~5`, then marking the 3 small commits with `squash`.

Differences Between Merge, Rebase, and Squash

- **Merge**: Combines changes from different branches and preserves the history with a merge commit.
  - **Advantage**: Full history is maintained, including merge points.
  - **Disadvantage**: Can create a more complex history with many branches and merge commits.
- **Rebase**: Moves or re-applies commits from one branch onto another, creating a linear history.
  - **Advantage**: Results in a cleaner, linear history.
  - **Disadvantage**: Rewrites history, which can be risky if done on shared branches.
- **Squash**: Combines multiple commits into a single commit, often used during a rebase.
  - **Advantage**: Simplifies the commit history by reducing the number of commits.
  - **Disadvantage**: You lose the granularity of individual commits.

Practical Scenarios

Scenario 1: Using Merge

- **When**: You're working on a team project, and you want to bring in the changes from `main` into your feature branch without altering the history.
- **How**: Use `git merge main` to combine the changes.

Scenario 2: Using Rebase

- **When**: You're working on a solo project, and you want to keep the project history clean and linear before merging your feature branch.
- **How**: Use `git rebase main` to apply your changes on top of the latest `main` branch.

Scenario 3: Using Squash

- **When**: You've made several small commits while working on a feature, and you want to combine them into one commit before merging into `main`.
- **How**: Use `git rebase -i HEAD~n` and mark the commits to be squashed.

Understanding Git Stash: What It Is, How to Use It, and When to Use It

In Git, the `git stash` command is a powerful tool that allows you to temporarily save your changes in a "stash" without committing them to your branch. This can be particularly useful when you need to switch branches or pull updates without losing your current work. Below, we'll explore what Git stash is, how it works, when to use it, and how to execute the various stash commands.

---

1. What Is Git Stash?

Git Stash is a way to save the current state of your working directory and staging area without committing the changes. This saved state, called a "stash," allows you to temporarily set aside your uncommitted work and return to a clean working directory, so you can work on something else.

- **Stash vs. Commit**: Unlike a commit, which records changes permanently, a stash is a temporary save that you can apply or discard later.

---

2. How Git Stash Works

When you run `git stash`, Git takes the changes in your working directory and staging area and saves them in a special area called the "stash list." It then resets your working directory and staging area to match the latest commit, allowing you to switch branches or perform other tasks without your uncommitted changes getting in the way.

---

3. Why and When to Use Git Stash

Why:

- **Context Switching**: When you're in the middle of working on something but need to switch branches or work on a different task.
- **Clean Workspace**: When you need to pull the latest changes from a remote repository but don't want to commit your current work-in-progress.
- **Experimentation**: When you want to try something out without making permanent changes to your branch.

When:

- **Switching Branches**: You're working on a feature branch, but you need to quickly fix a bug on another branch.
- **Updating Your Branch**: You want to pull the latest changes from `main` but have uncommitted changes in your working directory.
- **Prototyping**: You're experimenting with some code and want to temporarily save your progress without committing.

---

4. How to Use Git Stash

Basic Stashing

Stash Your Changes:

```
git stash
```

1. This command stashes all changes in your working directory and staging area.

View Your Stashes:

```
git stash list
```

2. This command lists all the stashes you've saved. Each stash is assigned an index, starting from `stash@{0}`.

Apply a Stash:

```
git stash apply
```

3. This command applies the most recent stash to your working directory without removing it from the stash list.

Apply and Remove a Stash:

```
git stash pop
```

4. This command applies the most recent stash and removes it from the stash list.

Drop a Stash:

```
git stash drop
```

5. This command removes the most recent stash from the stash list without applying it.

Advanced Stashing

Stash with a Message:

```
git stash push -m "Work in progress on feature X"
```

1. This command saves the stash with a descriptive message, making it easier to identify later.

Stash Specific Files:

```
git stash push <file1> <file2>
```

2. This command stashes only the specified files, leaving other changes in your working directory.

Stash and Keep Index:

```
git stash push --keep-index
```

3. This command stashes only the changes in your working directory, keeping the changes in the staging area intact.

Apply a Specific Stash:

```
git stash apply stash@{1}
```

4. This command applies a specific stash by its index, such as `stash@{1}`.

---

5. Practical Scenarios for Using Git Stash

Scenario 1: Switching Branches

- You're working on a new feature but get an urgent request to fix a bug on `main`. You can stash your changes, switch to the `main` branch, fix the bug, then switch back and apply your stash.

Commands:

```
git stash

git checkout main

# Fix the bug

git commit -m "Fix bug"

git checkout feature-branch

git stash pop
```

Scenario 2: Pulling Changes from Remote

- You're halfway through a task but need to pull the latest changes from `main`. Instead of committing your half-done work, you can stash it, pull the changes, and then reapply your stash.

Commands:

```
git stash

git pull origin main

git stash pop
```

Scenario 3: Experimenting with Code

- You want to try out some experimental changes but aren't ready to commit them. You can stash your current work, make your experimental changes, and then either drop the stash or apply it if you decide to keep the experiment.

Commands:

```
git stash

# Try out new code

git stash drop  # If you want to discard the stash
```

Collaboration with GitHub

Forking and Cloning

- **Forking**: Copying someone else's repository to your own GitHub account. This is useful for contributing to open-source projects.
- **Cloning**: Downloading a repository from GitHub to your local machine.

Pull Requests (PRs)

- **What Are They?**: A way to propose changes to a repository.
- **When to Use**: When you've made changes in a forked repository and want to merge them back into the original repository.
- **How to Use**:
    1. Make changes in your forked repository.
    2. Push the changes to your GitHub fork.
    3. Create a pull request on the original repository.

Reviewing and Merging PRs

- **Code Review**: Review the changes proposed in a pull request.
- **Merge**: Once approved, merge the pull request to bring the changes into the main codebase.

Managing Conflicts

What Are Conflicts?

Conflicts occur when changes in two branches interfere with each other, and Git doesn't know which version to keep.

How to Resolve Conflicts

1. **Identify the Conflict**: Git will mark the files with conflicts.
2. **Edit the File**: Manually edit the conflicted file to resolve the conflict.

Add the Resolved File:

```
git add <file>
```

3.

Commit the Resolution:

```
git commit -m "Resolved merge conflict"
```

4.

GitHub Issues and Projects

Issues

- **What Are They?**: GitHub's way of tracking tasks, bugs, and feature requests.
- **How to Use**: Create an issue to describe a problem or request. Collaborators can discuss and track progress on the issue.

Projects

- **What Are They?**: Kanban-style boards for organizing and tracking work.
- **How to Use**: Create a project board, add issues and pull requests as cards, and move them across columns (e.g., To Do, In Progress, Done).

GitHub Actions

What Are GitHub Actions?

GitHub Actions is a CI/CD (Continuous Integration/Continuous Deployment) tool that allows you to automate workflows directly within your GitHub repository.

Common Use Cases

- **Automated Testing**: Run tests automatically when code is pushed.
- **Deployment**: Automatically deploy code to production after passing tests.
- **Code Linting**: Check code style and standards automatically.

Setting Up GitHub Actions

1. **Create a Workflow File**: Place it in the `.github/workflows/` directory of your repository.
2. **Define the Workflow**: Specify triggers, jobs, and steps in the workflow file (YAML format).
3. **Commit the Workflow File**: Once committed, the workflow will automatically run when triggered.

Example:

```
name: CI

on: [push, pull_request]

jobs:

  build:

    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run tests
        run: npm test
```

Best Practices

1. Commit Often

Make frequent commits with descriptive messages. This makes it easier to track changes and understand the history of your project.

2. Use Branches

Create branches for features, fixes, or experiments. This keeps the main branch clean and stable.

3. Write Meaningful Commit Messages

Your commit messages should describe what changes were made and why. This is crucial for collaboration and understanding the project history.

4. Regularly Pull from the Remote Repository

Before starting work, pull the latest changes from the remote repository to ensure you’re working with the most up-to-date code.

5. Keep Your Repository Organized

Use a consistent structure for your files and directories. This helps in navigating the project, especially when it grows in size.

This guide covers the foundational concepts of Git and GitHub. By understanding and practicing these concepts, you'll be well-equipped to manage your code, collaborate with others, and contribute to projects with confidence. Happy coding! — **EMMANUEL NMAJU**