**Git topics:**

Git init

Git checkout

Git add

Git commit

Git push

Git pull

Git rebase

Git branch

Git merge

Merge conflicts

**What is Git?**

Git is a distributed version control system designed for tracking changes in source code and coordinating collaborative software development projects. It allows multiple developers to work on the same codebase simultaneously, keeping track of individual changes, enabling efficient collaboration, and providing mechanisms for branching, merging, and version management. Git maintains a complete history of a project's development, facilitating the ability to revert to previous states, compare changes, and manage different lines of development. It is widely used in software development to streamline teamwork, enhance code quality, and ensure effective management of codebases.

---------------------------------------------------------------------------------------------------------------------

**Git init:**

Git init is a command in Git that initializes a new, empty Git repository in a directory.

Examples:

**1. Creating a New Repository:(shell)**

**$ mkdir my\_project**

**$ cd my\_project**

**$ git init**

This sequence of commands creates a new directory called "my\_project," navigates into it, and initializes a new Git repository there.

**2. Initializing an Existing Directory as a Git Repository:(shell)**

**$ cd existing\_directory**

**$ git init**

If you have an existing directory with files and you want to start using Git to track changes, running git init in that directory will set up version control for those files.

**3. Initializing a Bare Repository (Server Repository) :(csharp)**

**$ git init --bare my\_project.git**

Adding the --bare flag creates a "bare" repository, which is used as a central repository on a server to facilitate collaboration without allowing direct editing of files.

---------------------------------------------------------------------------------------------------------------------

**Git checkout:**

git checkout is a Git command used for switching between branches.

The three main purposes of git checkout:

**1.Switching Branches: (ruby)**

You can use git checkout to switch from one branch to another. When you switch branches, your working directory and files are updated to reflect the state of the branch you're switching to. This allows you to work on different features or bug fixes in separate branches without affecting each other.

For example, if you have a branch named feature\_branch and you want to switch to it:

**$ git checkout feature\_branch**

**2.Navigating Commit History: (ruby)**

You can also use git checkout to navigate through the commit history of a repository. By checking out a specific commit, you can examine the code as it existed at that point in time. This is particularly useful for reviewing code or troubleshooting issues. For example, to check out a specific commit identified by its SHA-1 hash (e.g., c8a3f7b):

**$ git checkout c8a3f7b**

**3.Detached HEAD State: (ruby)**

When you check out a specific commit (rather than a branch), you enter a state known as "detached HEAD" state. In this state, you're not on a named branch but instead directly on a commit. You can view the code at that commit, but any new commits you make will not be associated with a branch until you check out a branch again.

For example, checking out a commit by SHA-1 hash:

**$ git checkout c8a3f7b**

To return to a branch and exit the detached HEAD state:

**$ git checkout master**

-------------------------------------------------------------------------------------------------------------------

**Git add:**

git add is a Git command used to stage changes in your working directory, preparing them to be included in the next commit. When you make changes to files in your project, Git doesn't automatically track them for the next commit. Instead, you need to explicitly tell Git which changes you want to include in the commit by using the git add command.

Examples of how it works: **(csharp)**

**1.Staging Changes:**

When you modify files in your working directory, Git categorizes those changes as either "untracked," "modified," or "deleted." The git add command is used to stage these changes for commit. Staging is the process of marking specific changes to be included in the next commit, effectively telling Git that you want to record those changes as part of your project's history.

**2.Adding New Files:**

For new files that you've created and want to include in the next commit, you need to use git add to stage them. This tells Git to start tracking the file and include it in the commit.

**$ git add new\_file.txt**

**3.Updating Modified Files:**

If you've made changes to an existing file and want those changes to be part of the next commit, you need to stage the modified file using git add.

**$ git add modified\_file.txt**

**4.Removing Deleted Files:**

If you've deleted a file and want to include the deletion in the next commit, you use git add to stage the deletion.

**$ git add deleted\_file.txt**

**5.Selective Staging:**

You can use git add with specific file paths to stage only certain changes from multiple modified files. This allows you to create more focused commits that only include relevant changes.

**$ git add file1.txt file2.txt**

**6.Interactive Staging:**

Git also provides an interactive mode for staging changes. You can use "git add -i" or "git add --patch" to interactively select and stage changes, which is especially useful when you want to review and stage portions of a file.

-----------------------------------------------------------------------------------------------------------

**Git Commit:**

The git commit command will save all staged changes, along with a brief description from the user, in a “commit” to the local repository.

Below is how Git commit works:

**1.Staging Changes:**

Before you can create a commit, you need to use the git add command to stage the changes you want to include in that commit. Staging is the process of selecting specific changes from your working directory and moving them to the staging area.

**2.Creating a Commit:**

Once you have staged the desired changes, you can use the git commit command to create a new commit. When you create a commit, Git takes a snapshot of all the files that are currently staged in the index and records metadata such as the author's name, email, commit message, and a timestamp.

**3.Commit Message: (ruby)**

When you run git commit, Git opens a text editor where you can enter a commit message. The commit message is a brief description of the changes included in the commit. A well-written commit message helps you and others understand the purpose and context of the changes introduced by the commit.

**$ git commit**

Alternatively, you can provide a commit message directly using the -m flag:

**$ git commit -m "Add new feature"**

**4.Creating a Unique Identifier (SHA-1 Hash):**

Every commit in Git is assigned a unique identifier called a SHA-1 hash. This hash is based on the contents of the commit, its parent commit(s), the author's information, and the commit message. The hash ensures that each commit has a unique identifier and can be referenced consistently.

**5.Commit History:**

Each commit forms a link in the chain of commits, creating a history of changes. You can visualize the commit history using tools like ‘**git log’**, which shows the list of commits in chronological order. Commits also allow you to move backward and forward in time within the project's history.

----------------------------------------------------------------------------------------------------------

**Git push:**

git push is a command used to upload local commits and associated branch references to a remote repository. It's a key step in collaborating with others and sharing your changes with a central repository or a remote server. When you push your commits, you make them available for others to see, review, and incorporate into their own work.

Here are few examples on how it works:

**1.Remote Repository:**

A remote repository is a version of your Git repository stored on a server, often hosted by a service like GitHub, GitLab, or Bitbucket. The remote repository serves as a centralized location where team members can collaborate by sharing and syncing their code changes.

**2.Local Commits:**

Before you can push changes, you need to create commits in your local repository using the ‘**git commit**’ command. Commits represent snapshots of your project's code at specific points in time.

**3.Associating with a Remote: (csharp)**

Before you can push your commits to a remote repository, you need to associate your local repository with the remote. You do this by adding a remote URL to your local repository configuration using the **git remote add** command. The remote URL typically points to the repository's location on a remote server.

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

In this example, "origin" is a common name for the remote, and the URL points to the repository on GitHub.

**4.Pushing Commits:** (ruby)

After associating your local repository with a remote, you can use the **git push** command to send your commits to that remote repository. The basic syntax is as follows:

**$ git push <remote-name> <local-branch-name>:<remote-branch-name>**

For example, to push the commits from your local master branch to the master branch of the remote repository named "origin," you can use: (perl)

**$ git push origin master:master**

This command pushes your local **master** branch to the **master** branch on the remote repository named "origin."

**5.Updating Remote References:**

When you push your commits to a remote, the remote's branch references are updated to include the new commits. This allows others to see and access your changes.

**6.Collaboration:**

Other team members can then pull your changes from the remote repository into their own local repositories using the **git pull** command. This helps keep everyone's codebase up to date and in sync.

--------------------------------------------------------------------------------------------------------------

**Git Pull:**

git pull is a command used to update your local repository with changes from a remote repository. It combines two actions: fetching changes from the remote repository and merging those changes into your current local branch. This command is used to bring your local repository up to date with the latest changes made by other collaborators.

How the git pull gonna work:

**1. Fetching Changes:**

The first step of **git pull** is to fetch the latest changes from the remote repository. This involves retrieving the new commits, branches, and other updates that have been made in the remote repository since your last interaction.

**2.Merging Changes:**

After fetching the changes, **git pull** automatically merges them into your current local branch. If the local branch has no uncommitted changes, the merge typically happens automatically. However, if there are conflicts between the fetched changes and your local changes, you may need to resolve those conflicts manually.

**3. Updating Your Local Repository:**

Once the changes are fetched and merged, your local repository is updated to reflect the latest state of the remote repository. This helps ensure that your local codebase remains in sync with the shared repository.

Here's the basic syntax of the **git pull** command: (php)

**$ git pull <remote-name> <branch-name>**

For example, to pull changes from the master branch of the remote repository named "origin," you would use: (ruby)

**$ git pull origin master**

It's important to note that **git pull** performs both the fetching and merging steps in one command. If you want more control over these steps or prefer to fetch changes without automatically merging them, you can use the **git fetch** command followed by **git merge**: (shell)

**$ git fetch origin**

**$ git merge origin/master**

-------------------------------------------------------------------------------------------------------------

**Git Rebase:**

**git rebase** is a command used to modify the commit history of a branch by moving, combining, or altering commits. It is a powerful but potentially complex command that allows you to change the order of commits, squash multiple commits into one, and incorporate changes from another branch into your current branch. Rebase is particularly useful for creating a cleaner and more linear commit history.

How git rebase works:

**1.Moving Commits:**

When you perform a rebase, you choose a starting commit and specify a target commit. The rebase operation will take the commits in between and move them to the target commit. This can help you reorder your commits or incorporate changes from another branch into your current branch.

**2.Interactive Rebase:**

One of the most powerful features of **git rebase** is the interactive mode, invoked with **git rebase -i**. This allows you to interactively edit and reorganize your commit history. You can squash, fixup, reorder, edit commit messages, and more.

**3.Squashing Commits:**

With interactive rebase, you can squash multiple commits into a single commit, which is useful for combining related changes or cleaning up a series of smaller commits.

**4.Resolving Conflicts:**

During a rebase, if conflicts arise between your changes and the changes you're rebasing onto, you'll need to resolve these conflicts just as you would during a merge. After resolving conflicts, you can continue the rebase using **git rebase --continue**.

**5.Potential Pitfalls:**

While rebase can create a cleaner history, it's essential to use it judiciously. Rebase rewrites commit history, creating new commits with different parent commits. This can cause issues for collaborators who have based their work on the original commit history.

**6. git pull --rebase:**

Instead of using **git pull** (which combines fetch and merge), you can use **git pull --rebase** to fetch changes from the remote repository and reapply your local commits on top of them. This results in a linear commit history and can simplify the process of integrating remote changes.

Here's an example of using interactive rebase to squash commits:

Invoke interactive rebase:

**$ git rebase -i HEAD~3**

This opens an interactive text editor listing the last 3 commits.

Change "pick" to "squash" for the commits you want to squash into the previous commit.

Save and close the editor.

---------------------------------------------------------------------------------------------------

**Git Branch:**

A branch is a lightweight, movable pointer to a commit. It represents an independent line of development, allowing you to work on separate features, bug fixes, or experiments without affecting the main codebase. Each branch maintains its own commit history, enabling you to develop and isolate changes without disrupting the stability of the main branch.

How it works:

**1.Creating a New Branch:**

You can create a new branch using the git branch command. This command creates a new branch pointer that initially points to the same commit as the current branch. The basic syntax is: (ruby)

**$ git branch new\_feature**

This creates a new branch named **new\_feature** based on the current commit.

**2.Creating and Switching to a New Branch:**

You can create and immediately switch to a new branch using the **-b** flag with **git checkout** or **git switch**:

**$ git checkout -b new\_feature**

**Or**

**$ git switch -c new\_feature**

**3.Committing on a Branch:**

When you're on a branch, any new commits you make will be added to that branch's history. This allows you to work on distinct features or changes without affecting other branches.

**4.Merging Branches:**

To incorporate changes from one branch into another, you can use the **git merge** command. This combines the commit histories of the branches and integrates their changes. Conflicts may arise if changes conflict in the same parts of a file.

**5.Resolving Conflicts:** When merging or rebasing branches, conflicts may occur when changes conflict with each other. You need to resolve these conflicts manually by editing the affected files.

**6.Visualizing Branches:**

You can use tools like **git log**, graphical interfaces, or online platforms to visualize the branching structure and commit history. This helps you understand the relationships between branches and commits.