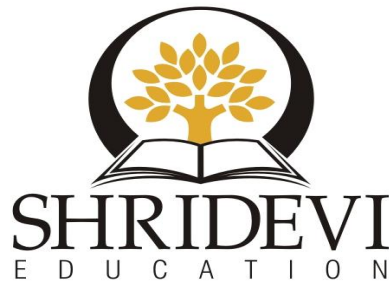


SHRIDEVI INSTITUTE OF ENGINEERING & TECHNOLOGY
SIRA ROAD, TUMKUR – 572106



LAB MANUAL

(As per CBCS Scheme 2022)

Project Management with git

(BCS358C)

DEPARTMENT OF AI&DS

Project Management with Git		Semester	3
Course Code	BCS358C	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	0: 0 : 2: 0	SEE Marks	50
Credits	01	Exam Marks	100
Examination type (SEE)	Practical		
Course objectives: <ul style="list-style-type: none">• .To familiar with basic command of Git• To create and manage branches• To understand how to collaborate and work with Remote Repositories• To familiar with version controlling commands			
Sl.NO	Experiments		
1	Setting Up and Basic Commands Initialize a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.		
2	Creating and Managing Branches Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."		
3	Creating and Managing Branches Write the commands to stash your changes, switch branches, and then apply the stashed changes.		
4	Collaboration and Remote Repositories Clone a remote Git repository to your local machine.		
5	Collaboration and Remote Repositories Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.		
6	Collaboration and Remote Repositories Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge.		
7	Git Tags and Releases Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.		
8	Advanced Git Operations		

	Write the command to cherry-pick a range of commits from "source-branch" to the current branch.
9	Analysing and Changing Git History Given a commit ID, how would you use Git to view the details of that specific commit, including the author, date, and commit message?
10	Analysing and Changing Git History Write the command to list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31."
11	Analysing and Changing Git History Write the command to display the last five commits in the repository's history.
12	Analysing and Changing Git History Write the command to undo the changes introduced by the commit with the ID "abc123".
Course outcomes (Course Skill Set): At the end of the course the student will be able to: <ul style="list-style-type: none">• Use the basics commands related to git repository• Create and manage the branches• Apply commands related to Collaboration and Remote Repositories• Use the commands related to Git Tags, Releases and advanced git operations• Analyse and change the git history	

Introduction to git

What is repository, what is Git, and why you should use Git.

A **repository**, also known as a repo, is a central storage location where it can store and manage the source code and related files. It acts as a version control system, keeping track of the changes made to files over time. A repository allows multiple developers to collaborate on a project, keeping everyone in sync and maintaining a history of all changes.

Git is a distributed version control system that helps to manage codebase effectively. It was created by Linus Torvalds, the same person who developed the Linux operating system. Git enables multiple developers to work on the same project simultaneously and efficiently.

Here are a few reasons why you should use Git:

Version Control: With Git, a complete history of all changes made to the code. It allows to track modifications, switch between different versions, and revert to previous states if needed. This helps in bug tracking, troubleshooting, and collaboration with other team members.

Collaboration: Git facilitates collaboration by providing features such as branching and merging. It can create branches to work on specific features or fixes independently, and later merge them back into the main branch. This allows multiple developers to work concurrently without interfering with each other's code.

Backup and Recovery: Git acts as a backup system for the code. Since every change is committed and stored in the repository, it can recover any previous version of the code easily. Even if local copy is lost or damaged, it can retrieve the code from the repository.

Code Integrity: Git ensures the integrity of your code by using checksums for each file. This means that the content of files is always verified, reducing the chances of code corruption.

Open Source Community: Git is widely used and supported by a vast open-source community. This means that there is plenty of documentation, tutorials, and resources available to help for learn and troubleshoot any issues you may encounter.

In summary, Git is a powerful version control system that allows to track changes, collaborate with others, backup the code, and ensure code integrity. It provides a reliable and efficient way to manage codebase, making it an essential tool for devCreating and Managing Branches.

What is the purpose of Git?

Git is primarily used to manage project, comprising a set of code/text files that may change.

What is version control

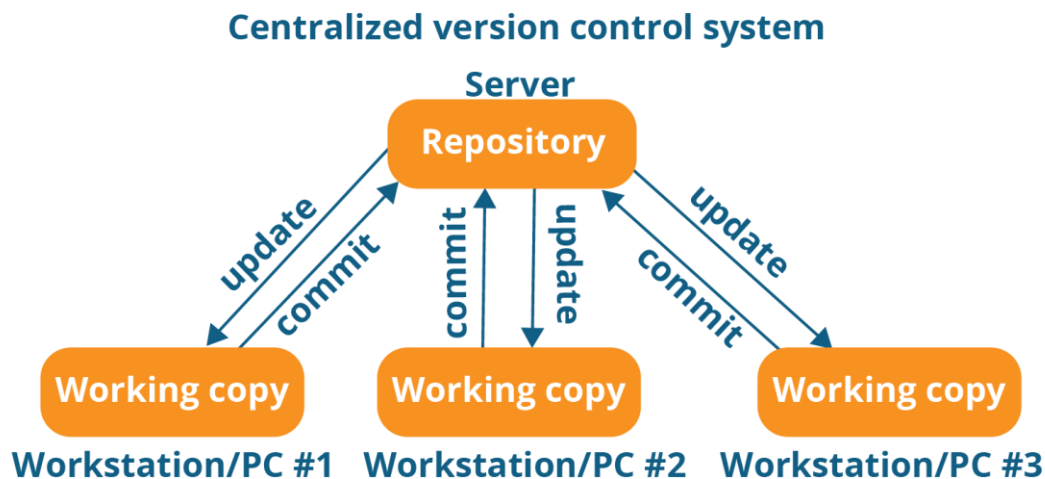
Version Control is the management of changes to documents, computer programs, large websites and other collection of information.

There are two types of VCS:

- Centralized Version Control System (CVCS)
- Distributed Version Control System (DVCS)

Centralized VCS

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. It works on a single repository to which users can directly access a central server.



The repository in the above diagram indicates a central server that could be local or remote which is directly connected to each of the programmer's workstation.

Every programmer can extract or **update** their workstations with the data present in the repository or can make changes to the data or **commit** in the repository. Every operation is performed directly on the repository.

Even though it seems pretty convenient to maintain a single repository, it has some major drawbacks. Some of them are:

- It is not locally available; meaning you always need to be connected to a network to perform any action.
- Since everything is centralized, in any case of the central server getting crashed or corrupted will result in losing the entire data of the project.

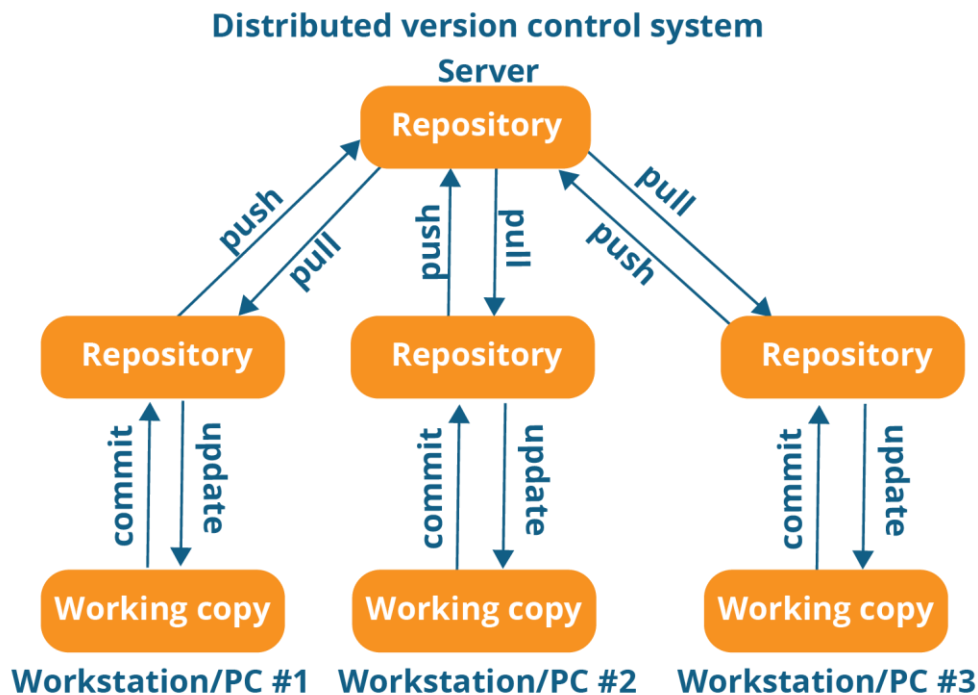
This is when Distributed VCS comes to the rescue.

Distributed VCS

These systems do not necessarily rely on a central server to store all the versions of a project file.

In Distributed VCS, every contributor has a local copy or “clone” of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository.

:



In the above diagram, every programmer maintains a local repository on its own, which is actually the copy or clone of the central repository on their hard drive. They can commit and update their local repository without any interference.



They can update their local repositories with new data from the central server by an operation called “**pull**” and affect changes to the main repository by an operation called “**push**” from their local repository.

What is GitHub?

GitHub is a Git repository hosting service that provides a web-based graphical interface. It is the world’s largest coding community. Putting a code or a project into GitHub brings it increased, widespread exposure. Programmers can find source codes in many different languages and use the command-line interface, Git, to make and keep track of any changes.

GitHub helps every team member work together on a project from any location while facilitating collaboration.

Git vs GitHub

	
1. It is a software	1. It is a service
2. It is installed locally on the system	2. It is hosted on Web
3. It is a command line tool	3. It provides a graphical interface
4. It is a tool to manage different versions of edits, made to files in a git repository	4. It is a space to upload a copy of the Git repository
5. It provides functionalities like Version Control System Source Code Management	5. It provides functionalities of Git like VCS, Source Code Management as well as adding few of its own features

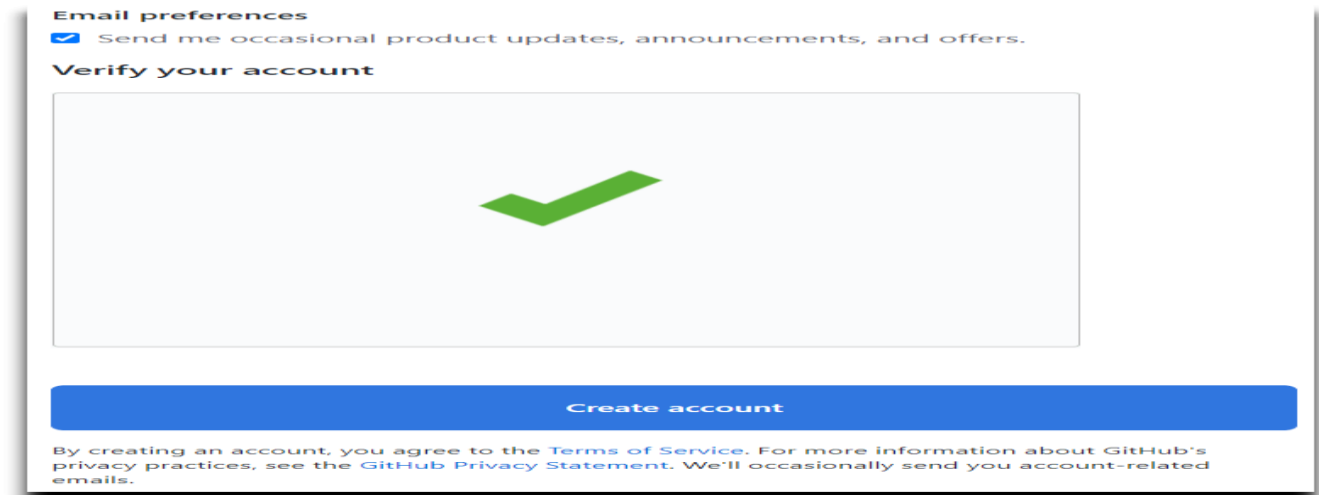
Step #1: How to Create Account in GitHub

Go to [GitHub.com](https://github.com) and Click on Sign up option on right top corner of desktop

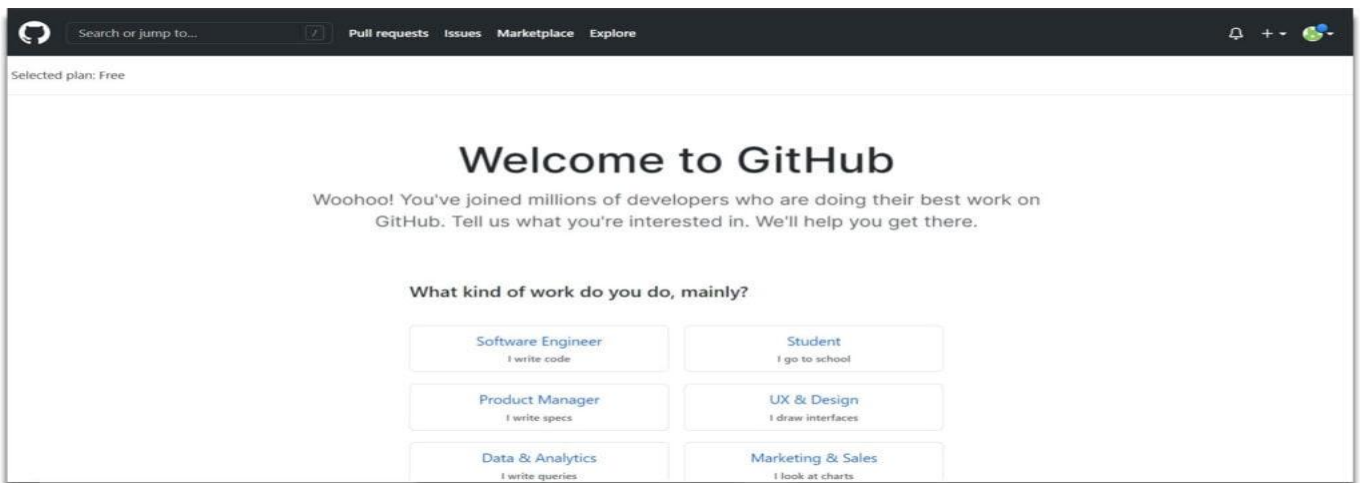


Enter all your basic information like **Username**, **Email address**, **Password** for account.

Click on **Create Account**.

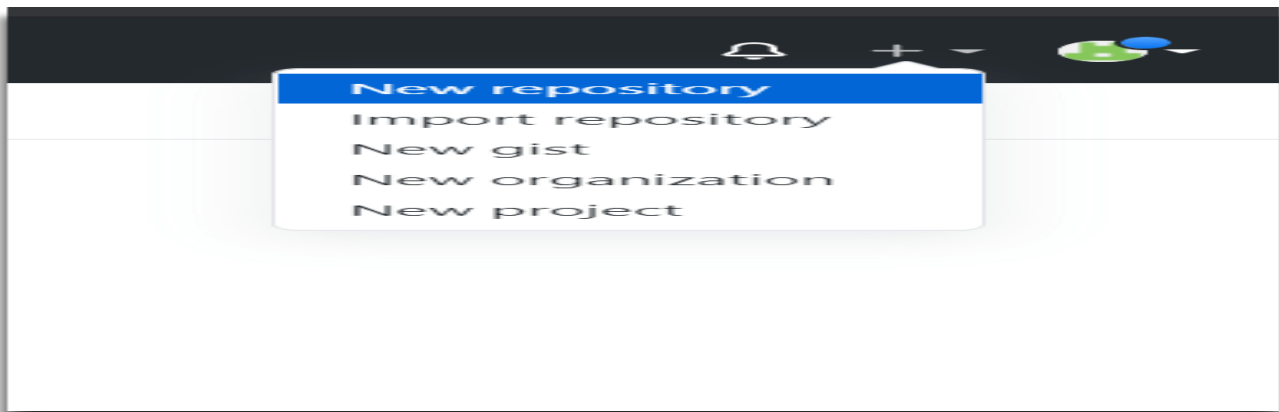


After creating account in GitHub this will be the first page which will be displayed.



Step #2: How to Create New Repository in GitHub

In the top right corner of GitHub's home page we can observe a add (+) button, we just have click on that and select “**New Repository**” option.

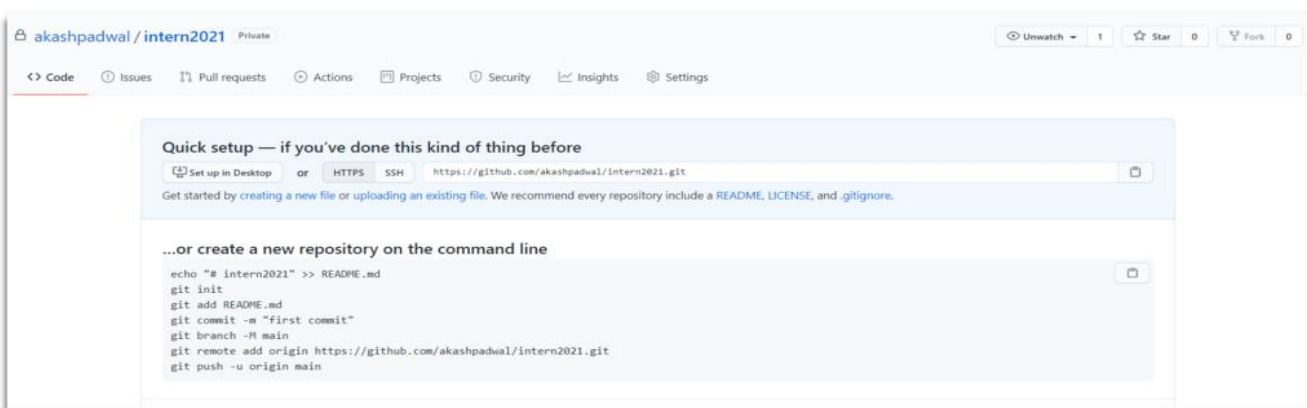


Give a name for your repository for Ex. “**intern2021**”. Write a short description about your repository, Let the access be **Private** or **Public**.

“**Intialize repository**”, Select the first option **Add a README file**, Now click on **Create repository**

A screenshot of the 'Create a new repository' form on GitHub. The form has a title 'Create a new repository' and a subtitle 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.' Below this, there are fields for 'Owner' (set to 'akashpadwal') and 'Repository name' (set to 'intern2021'). A note says 'Great repository names are short and memorable. Need inspiration? How about effective-octo-rotary-phone?'. There's a 'Description (optional)' field. Below these are radio buttons for 'Public' (selected) and 'Private'. Under 'Initialize this repository with:', there are three checkboxes: 'Add a README file' (selected), 'Add .gitignore', and 'Choose a license'. At the bottom is a green 'Create repository' button.

a repository is created, use below instructions on GitHub to create New repo on command line.



Copylink from github after creating repository demo.txt

git remote add origin paste the link(<http://github.com/username/demo.git>)

git push origin master

git remote -v // it shows origin <http://github.com/username/demo.git>

1.Setting Up and Basic Commands

Step to initialize a new Git repository, create a new file, add it to the staging area, and commit the changes.

Open the terminal or command prompt and navigate to the directory where you want to initialize the Git repository.

To initialize a new Git repository, use the following command: **git init** This will create a new, empty Git repository in the current directory.

Now, let's create a new file in the repository. we can use any text editor you prefer. For example, if you want to create a file named "example.txt", you can use the command: **touch example.txt** This command creates an empty file named "example.txt" in the current directory. or **vi exm.txt**

After creating the file, we need to add it to the staging area. The staging area is where we specify which changes we want to include in the next commit. Use the following command to add the file to the staging area: **git add example.txt** This command **git add "example.txt"** to the staging area.

Finally, we can commit the changes with an appropriate commit message. A commit is like a snapshot of the current state of the repository. Use the following command to commit the changes: **git commit -m "Add example.txt file"** Replace "Add example.txt file" with a meaningful commit message that explains the purpose of the commit.

GIT COMMANDS:

- **git --version**

This command to check the version

- **mkdir foldername** eg: **mkdir demo**
- **cd desktop**
- **cd demo**
- **git init**
- **git status**
- **git add demo.txt**
- **git commit -m "committing text file"**
- **git config user.name " "**
- **git config user.email " "**
- **git config --global user.name " " //To link with git hub account ,git hub username & password**
- **git config --global user.email " "**

Usage: **git config --global user.name "[name]"**

Usage: **git config --global user.email "[email address]"**

This command sets the author name and email address respectively to be used with your commits.

2. Creating and Managing Branches

Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."

Commands:

- Create a new branch named "feature-branch":

git branch feature-branch //This will create new branch called feature-branch

git checkout feature-branch

This command creates a new branch named "feature-branch" and switches to it.

git checkout master

- Switch back to the "master" branch:

This command switches back to the "master" branch.

- Merge the "feature-branch" into "master":

git merge feature-branch

This command will merge the changes from "feature-branch" into the "master" branch.

After completing these steps, your "feature-branch" changes will be integrated into the "master" branch.

git status //it will check the status

git push origin master

3. Creating and Managing Branches

Commands to stash your changes, switch branches, and then apply the stashed changes

Stash your changes: **git stash** This command will temporarily save your changes, allowing you to switch branches without committing them. Git will revert your working directory and staging area to the last commit, giving you a clean state to switch branches.

Switch branches: **git checkout <branch-name>** Replace **<branch-name>** with the name of the branch you want to switch to. This command allows you to move to a different branch in your Git repository.

Apply the stashed changes: **git stash apply** This command reapplies the most recent stash you created. It will restore your previously stashed changes, allowing you to continue working on them.

If you have multiple stashes, you can apply a specific stash by using its index. First, list the stashes using the command **git stash list**. Then, apply a specific stash by index using the command: **git stash apply stash@{<index>}** Replace **<index>** with the index number of the stash you want to apply. Additionally, if you want to remove the stash after applying it, you can use the command

git stash drop followed by the stash's index or **git stash drop stash@{<index>}**.

It's worth noting that stashing is useful when you want to switch branches without committing your changes. It allows you to work on multiple branches while keeping your changes separate and easily applicable when needed.

Commands:

git branch feature-branch2

git status

vi sample.txt

git add sample.txt

git checkout feature-branch2

git log --oneline

git stash

git stash list

git stash show stash@{1}

git stash apply

git stash pop

git stash list

4. Collaboration and Remote Repositories

Clone a remote Git repository to your local machine

The process of cloning a remote Git repository to your local machine:

Open the terminal or command prompt on your local machine and navigate to the directory where you want to clone the remote repository.

Obtain the URL of the remote Git repository you want to clone. This can usually be found on the repository's webpage or by asking the repository owner. The URL will typically end with **.git**.

To clone the remote repository, use the following command: **git clone <repository-url>** Replace **<repository-url>** with the URL of the remote repository you obtained in the previous step. This command will create a copy of the remote repository on your local machine.

Git will start downloading the remote repository's contents to your local machine. Once the cloning process is complete, you will have a local copy of the entire repository, including its commit history and branches.

You have successfully cloned a remote Git repository to your local machine. The cloned repository will be stored in a new directory with the same name as the remote repository.

Now, you can start working with the cloned repository on your local machine. You can make changes, create branches, commit your work, and push your changes back to the remote repository when you're ready to share or collaborate with others.

Cloning a remote repository is an essential step in collaborating on Git projects. It allows you to have a local copy of the project, work independently on your machine, and easily synchronize your changes with the remote repository.

command: git clone <repository-url>

5. Collaboration and Remote Repositories

Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.

Step1: Ensure that you are in the working directory of your local repository. Run `git fetch origin` to fetch the latest changes from the remote repository. Replace origin with the name of your remote repository if it is different. Check out the branch you want to rebase onto the updated remote branch. For example, if you want to rebase your feature-branch onto the updated master branch, run `git checkout feature-branch`.

Step2: Run `git rebase origin/master`, where origin/master is the remote branch you want to rebase onto. This command replays your commits on top of the updated remote branch. If there are any conflicts during the rebase process, Git will pause the rebase and display the conflicting files. You need to resolve the conflicts manually by editing the conflicting files. After resolving the conflicts, use `git add <file>` for each resolved file to stage them for the rebase. Once all conflicts have been resolved and files have been staged, continue the rebase process by running `git rebase --continue`. This will apply the next commit on top of the updated remote branch. If there are any further conflicts, repeat steps 5-7 until the rebase is successfully completed. After the rebase is complete, you may need to force push your updated branch to the remote repository if you have already pushed the previous commits. Use `git push -f origin feature-branch` to force push the updated branch. Be cautious with this step, as it rewrites the history and can cause issues for other collaborators. The local branch is now rebased onto the updated remote branch.

Commands step-by-step:

git fetch origin

git checkout feature-branch

git rebase master/origin feature-branch(Resolve any conflicts if prompted)

git add <resolved-file> (Stage each resolved file) **git rebase --continue** (Repeat steps 4-5 if necessary)

git push -f origin feature-branch (Force push the updated branch to the remote repository)

Note: Replace origin with the name of your remote repository if it is different, and change featurebranch and master with the actual branch names you are working with. Be cautious with the force push (`git push -f`), as it rewrites history and can cause issues for other collaborators.

6. Collaboration and Remote Repositories

Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge

First, ensure that you are currently on the "master" branch. You can switch to the "master" branch using the following command: **git checkout master**

Next, initiate the merge of the "feature-branch" into "master" by using the following command: **git merge feature-branch** This command will merge the changes from the "feature-branch" into the "master" branch.

At this point, Git will try to automatically merge the changes. If there are any conflicts (i.e., conflicting changes in the same files), Git will notify you and prompt you to resolve the conflicts manually. You can use a text editor or a specialized merge tool to address the conflicts.

Once you have resolved the conflicts, you can proceed with the merge. Git will create a new commit to represent the merge. To provide a custom commit message for the merge, use the following command: **git commit -m "Custom commit message for merging feature-branch into master"** Replace "Custom commit message for merging feature-branch into master" with your desired commit message. Make sure to provide a meaningful message that accurately describes the purpose of the merge.

After entering the command, Git will create the merge commit with the provided custom commit message. This commit represents the merge of the changes from the "feature-branch" into the "master" branch.

You have successfully merged the "feature-branch" into the "master" branch while providing a custom commit message for the merge.

Merging branches is a crucial step in collaboration, as it brings together different sets of changes from various branches into a single branch, such as "master." It allows different team members to work on separate features or bug fixes and later integrate them into the main branch.

Commands:

git commit -m "custom commit message"

7. Git Tags and Releases

Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.

First, ensure that you are in the desired Git repository directory on your local machine.

Identify the commit that you want to tag. You can find the commit hash by using the **git log** command, which displays the commit history.

To create a lightweight Git tag named "v1.0" for the identified commit, use the following command:

git tag v1.0 <commit-hash> Replace <commit-hash> with the specific commit hash that you want to tag. For example, if the commit hash is abc123, the command would be: **git tag v1.0 abc123** This command creates a lightweight tag with the name "v1.0" for the specified commit.

You have successfully created a lightweight Git tag named "v1.0" for the specific commit in your local repository.

Git tags are useful for marking specific points in history, such as releases or important milestones.

Lightweight tags are simply pointers to specific commits, containing no additional metadata.

It's important to note that lightweight tags are local to your repository and are not automatically pushed to a remote repository. If you want to share the tags with others, you will need to explicitly push them to the remote repository using the **git push** command.

To push the "v1.0" tag to a remote repository, you can use the following command: **git push origin v1.0** Replace origin with the name of the remote repository you want to push the tag to.

Commands:

git checkout master

git tag v1.0

git tag //it will show v1.0

git tag -a v1.1 -m "tag for release ver 1.1"

git show v1.0

git tag -l "v1.*"

git push origin v1.0 //push tags to origin

8.Advanced Git Operations

Write the command to cherry-pick a range of commits from "source-branch" to the current branch

First, ensure that you are currently on the branch where you want to apply the cherry-picked commits. Identify the range of commits you want to cherry-pick from the "source-branch". You can obtain the commit hashes or commit range using the git log command or other Git history visualization tools. To cherry-pick a range of commits from "source-branch" to the current branch, use the following command: **git cherry-pick <start-commit>..<end-commit>** Replace <start-commit> with the hash of the first commit in the range, and <end-commit> with the hash of the last commit in the range. For example, if you want to cherry-pick the commits with hashes abc123 to def456, the command would be: **git cherry-pick abc123..def456** This command applies the changes introduced by the specified range of commits to the current branch, effectively cherry-picking them.

Commands:

Create folder Git cherry picking

cd Git cherry picking

git init

vi alpha.txt

git add . | git commit -m "1st commit"

vi beta.txt

git add . | git commit -m "2st commit"

vi gamma.txt

git add . | git commit -m "3st commit"

git reflog //It shows all commit history

git add .

git status

git commit -m "all deleted"

git reflog

git cherry-pick id //----add commit history id eg:34946d4-----

git log // it show all commit history with commit id

9. Analysing and Changing Git History

Given a commit ID, how would you use Git to view the details of that specific commit, including the author, date, and commit message?

To view the details of a specific commit in Git, including the author, date, and commit message, you can use the following command: **git show <commit ID>** Replace `**<commit ID>**` with the actual commit ID you want to view. This command will display the commit details, including the author's name, email, date, and the commit message.

To view the details of a specific commit in Git, you can use the **git show** command followed by the commit ID. Here's a step-by-step explanation of how to use Git to view the details of a commit: 1. Open your terminal or command prompt and navigate to the Git repository where the commit is located. 2. Obtain the commit ID of the specific commit you want to view. You can find the commit ID by using commands like **git log** or **git reflog**. The commit ID is a unique identifier for each commit in Git. 3. Once you have the commit ID, use the following command to view the details of that commit: **git show <commit ID>** Replace `**<commit ID>**` with the actual commit ID you want to view. 4. After executing the command, Git will display the details of the commit. This includes information such as the author's name, email, date of the commit, and the commit message. The commit message is a brief description of the changes made in that commit. It provides context and helps understand the purpose of the commit. By using the **git show** command with the appropriate commit ID, you can easily view the details of a specific commit in Git, including the author, date, and commit message.

git log // it show all commit history with commit id

git show commitId // paste commitid

10. Analysing and Changing Git History

Write the command to list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31."

To list all commits made by the author "shamsiya parveen" between "2023-01-01" and "2023-12-31" in Git, you can use the following command:

git log --author="shamsiya parveen" --after="2023-01-01" --before="2023-12-31"

This command will display a list of commits made by "shamsiya parveen" within the specified date range.

git log: This is the command to view the commit history in Git.

--author="shamsiya parveen": This option filters the commit history based on the specified author name, in this case, "shamsiya parveen". Only commits made by this author will be displayed.

--after="2023-01-01": This option filters the commit history to show only the commits made after the specified date, which is "2023-01-01" in this case.

--before="2023-12-31": This option filters the commit history to show only the commits made before the specified date, which is "2023-12-31" in this case.

By combining all these options together, the command **git log --author="shamsiya parveen" --after="2023-01-01" --before="2023-12-31"** will display a list of commits made by "shamsiya parveen" between the dates "2023-01-01" and "2023-12-31".

command:

git log --author="shamsiya parveen " --after="2023-01-01" --before="2023-12-31"

11. Analysing and Changing Git History

Write the command to display the last five commits in the repository's history.

To display the last five commits in the repository's history, you can use the following command:

git log -n 5

git log: This is the command to view the commit history in Git.

-n 5: This option limits the output to the specified number of commits, in this case, 5. It tells Git to display only the most recent 5 commits.

By running **git log -n 5**, you'll get a list of the last five commits made in the repository. Each commit will be displayed with its commit message, author, date, and a unique commit hash.

Command:

git log -n 5

12. Analysing and Changing Git History

Write the command to undo the changes introduced by the commit with the ID "abc123".

To undo the changes introduced by the commit with the ID "abc123" in Git, you can use the **git revert** command. This command creates a new commit that undoes the changes made in the specified commit.

The command to undo the changes introduced by the commit with the ID "abc123" is:

textCopy code

git revert abc123

When you execute this command, Git will create a new commit that undoes the changes made in the "abc123" commit. The new commit will have a message indicating the revert and a unique commit ID. The git revert command is a safe way to undo changes since it does not rewrite the existing commit history. It adds a new commit that undoes the changes, which allows you to maintain a clear and accurate history of your project.

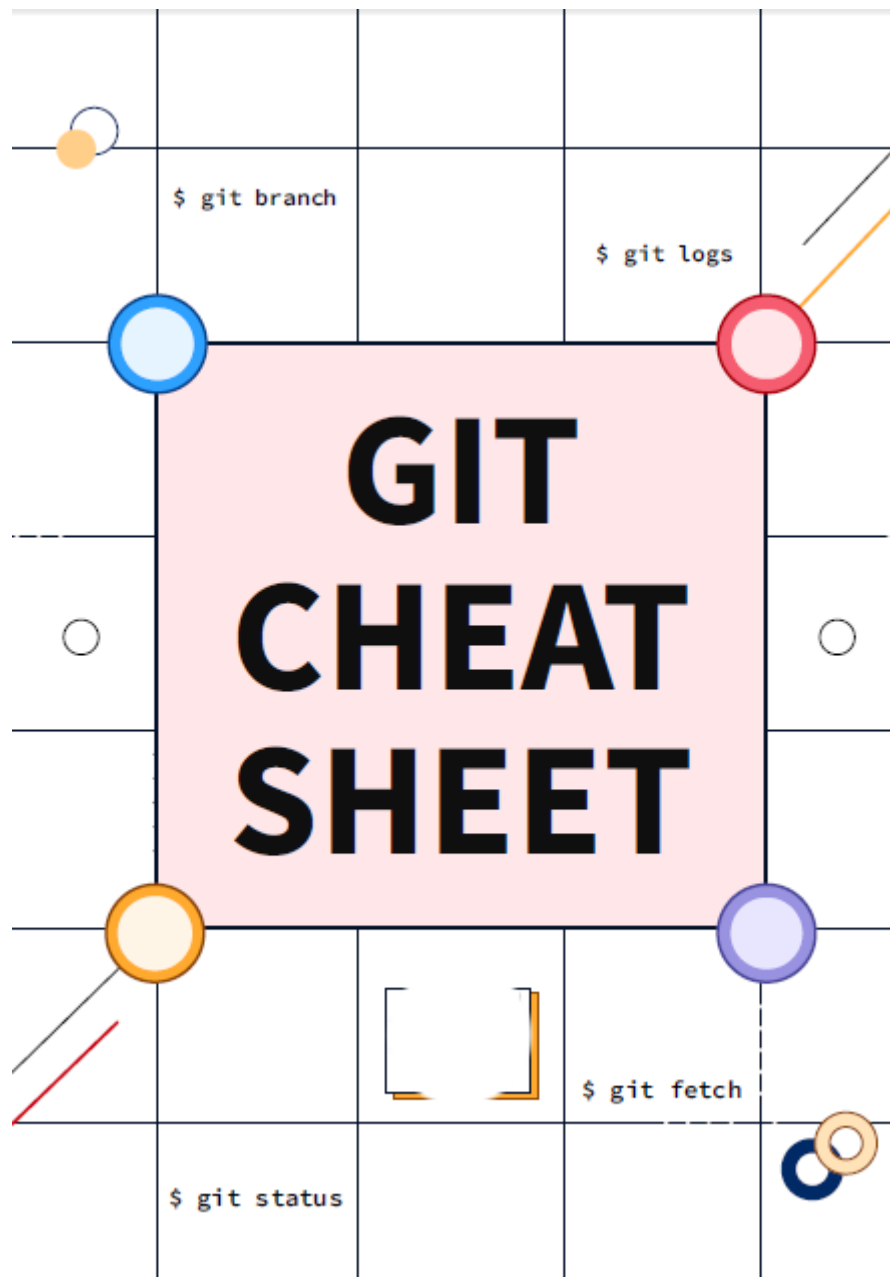
However, it's important to note that git revert is not the same as git reset. While git revert undoes a specific commit by creating a new commit, git reset allows you to move the branch pointer to a previous commit, effectively removing any commits after that point. **git reset** is a more powerful command and should be used with caution as it can permanently delete commits and history.

Commands:

Git revert "abc123"

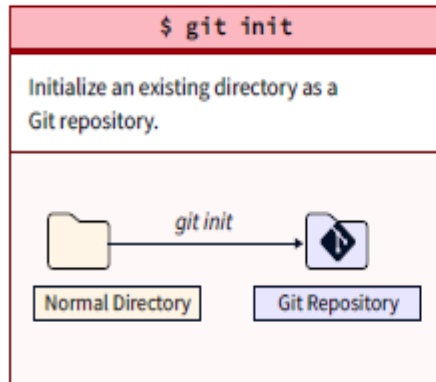
Git log --oneline // it shows commit history in oneline

Git reset --hard head.1



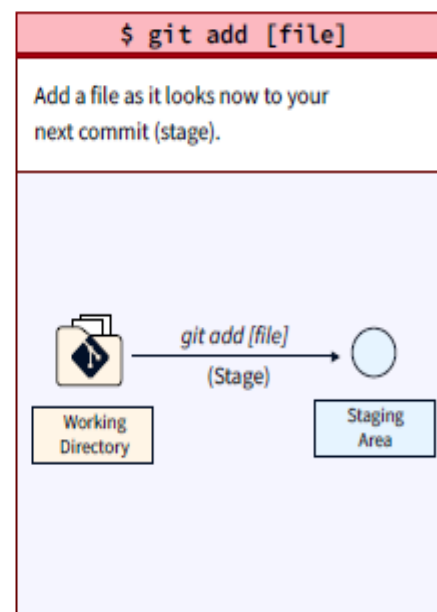
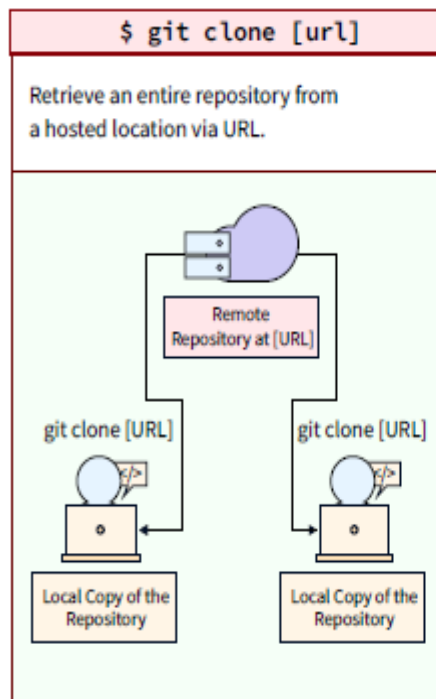
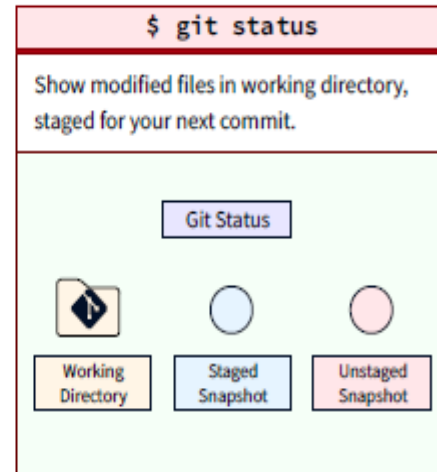
SETUP & INIT

Configuring user information, initializing and cloning repositories.



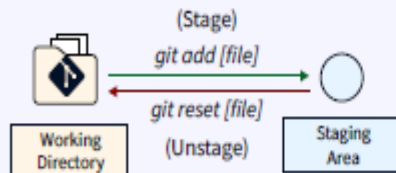
STAGE & SNAPSHOT

Working with snapshots and the Git staging area.

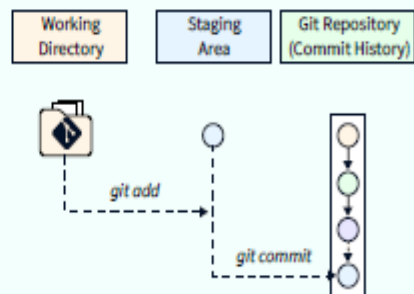


\$ git reset [file]

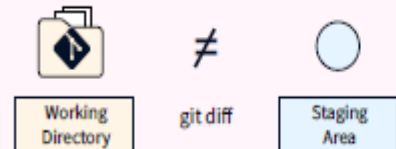
Unstage a file while retaining the changes in working directory.

**\$ git commit -m "[descriptive message]"**

Commit your staged content as a new commit snapshot.

**\$ git diff**

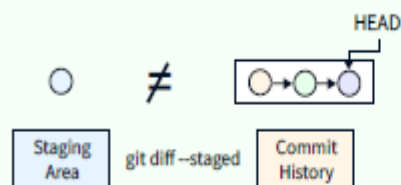
Diff of what is changed but not staged

**BRANCH & MERGE**

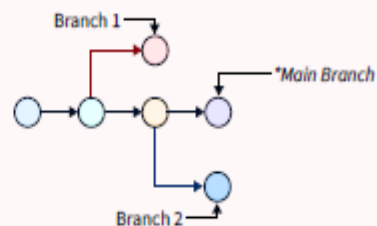
Isolating work in branches, changing context, and integrating changes.

git diff --staged

Diff of what is staged but not yet committed.

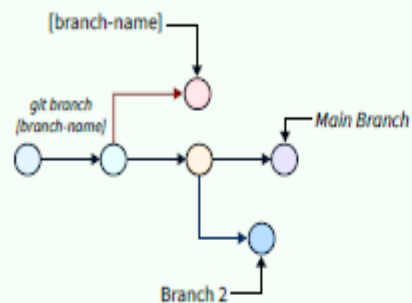
**\$ git branch**

List your branches. A * will appear next to the currently active branch.

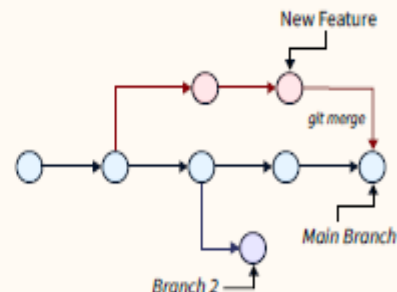


\$ git branch [branch-name]

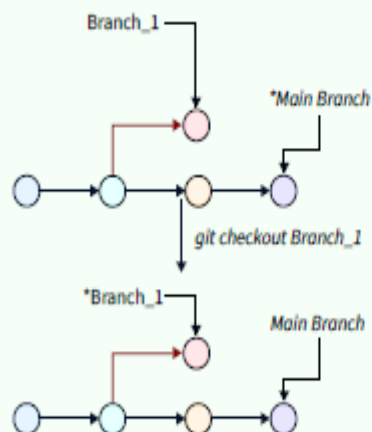
Create a new branch at the current commit.

**\$ git merge [branch]**

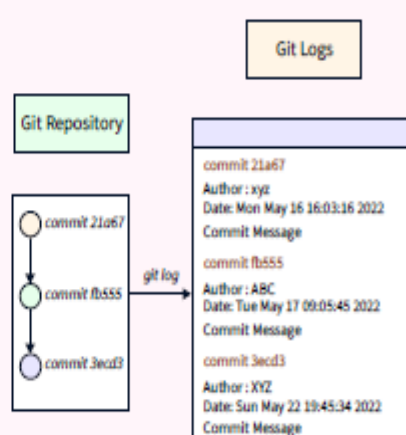
Merge the specified branch's history into the current one.

**\$ git checkout**

Switch to another branch and check it out into your working directory.

**\$ git log**

Add a file as it looks now to your next commit (stage).

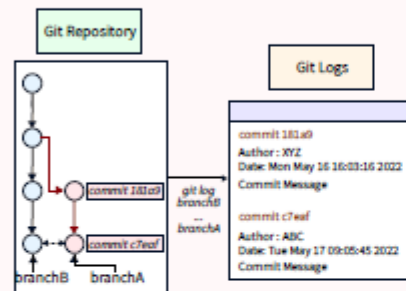


INSPECT & COMPARE

Configuring user information,
initializing and cloning repositories.

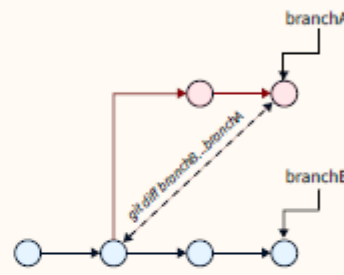
\$ git log branchB..branchA

Show the commits on branchA that
are not on branchB.



\$ git diff branchB...branchA

Show the diff of what is in branchA
that is not in branchB.

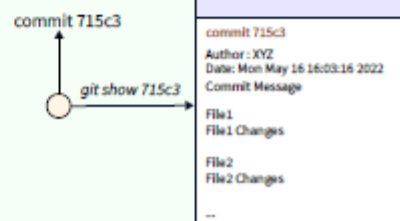


\$ git log --follow [file]

Show the commits that changed file,
even across renames.

\$ git show [SHA]

Show any object in Git in
human-readable format.

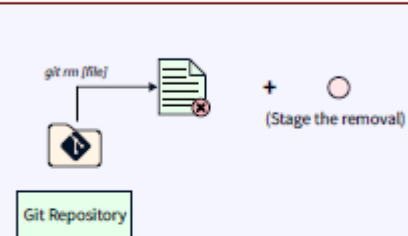


TRACKING PATH CHANGES

Versioning file removes and path changes.

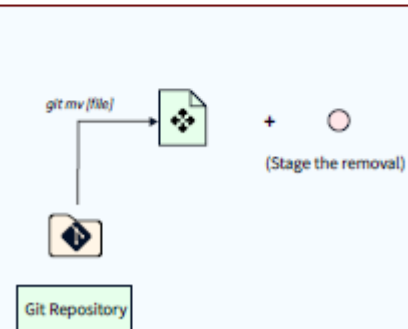
```
$ git rm [file]
```

Delete the file from the project and stage the removal for commit.



```
$ git mv [existing-path] [new-path]
```

Change an existing file path and stage the move.



```
$ git log --stat -M
```

Show all commit logs with indication of any paths that moved.

IGNORING PATTERNS

Preventing unintentional staging or committing of files.

```
logs/  
*.notes  
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

```
.gitignore
```

```
logs/  
*.notes  
pattern*/
```



```
$ git config --global core.excludesfile [file]
```

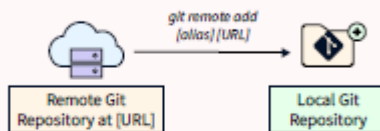
System wide ignore pattern for all local repositories

SHARE & UPDATE

Retrieving updates from another repository and updating local repos.

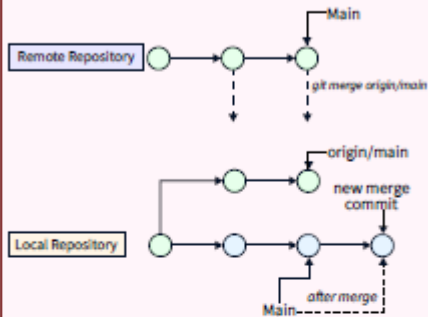
```
$ git remote add [alias] [url]
```

Add a git URL as an alias.



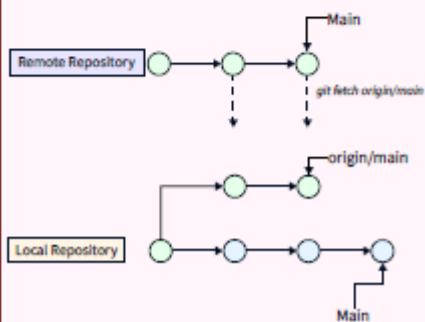
```
$ git merge [alias]/[branch]
```

Merge a remote branch into your current branch to bring it up to date.



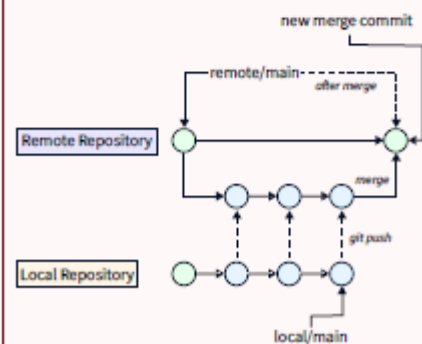
```
$ git fetch [alias]
```

Fetch down all the branches from that Git remote.



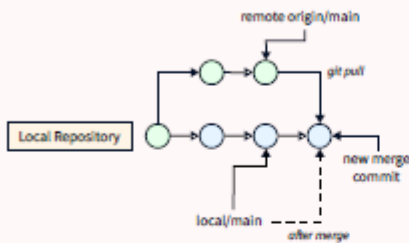
```
$ git push [alias] [branch]
```

Transmit local branch commits to the remote repository branch.

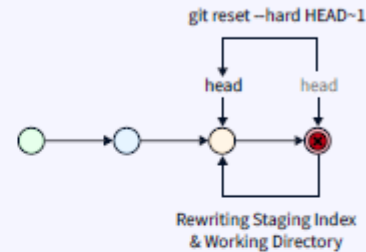


\$ git pull

Fetch and merge any commits from the tracking remote branch.

**\$ git reset --hard [commit]**

Clear staging area, rewrite working tree from specified commit.

**REWRITE HISTORY**

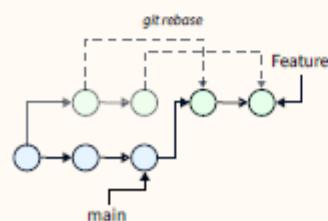
Rewriting branches, updating commits and clearing history.

TEMPORARY COMMITS

Temporarily store modified, tracked files in order to change branches.

\$ git rebase [branch]

Apply any commits of the current branch ahead of specified one.

**\$ git stash**

Save modified and staged changes.

