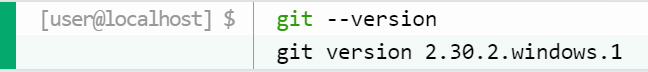
Git is a version control system.

Git helps you keep track of code changes.

Git is used to collaborate on code.

Example



What is Git?

It is used for:

* Tracking code changes
* Tracking who made changes
* Coding collaboration

What does git do:

* Manage projects with Repositories
* Clone a project to work on a local copy
* Control and track changes with Staging and Committing
* Branch and merge to allow for work on different parts and versions of a project
* Pull the latest version of the project to a local copy
* Push local updates to the main project

Working with Git

* Initialize Git on a folder, making it a Repository
* Git now creates a hidden folder to keep track of changes in that folder
* When a file is changed, added, or deleted, it is considered modified
* You select the modified files you want to Stage
* The staged files are committed, which prompts Git to store a permanent snapshot of the files
* Git allows you to see the full history of every commit
* Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit !

What is GitHub?

* GitHub makes tools that use Git
* GitHub is the largest host of source code in the world, and has been owned by Microsoft since 2018

*ESSENTIAL COMMANDS*

When it comes to commands, there come numerous commands in a software developer head or one that is mastering over git but even only a few of them are used frequently in the enterprising domains that are used frequently by developers in order to boost workflow. So here we will be listing a couple of them prior to dividing them into categories in order to perceive real quick. These commands are peculiar curated allowing to go from scratch from creating a new repository and playing further creating branches, merging to master branch.

Categories are as follows:

* To create
* To make local changes
* To commit history
* Branches and tags
* To update and publish
* To merge and reuse
* To undo

Let us do define commands that do fall under these categories that are listed below as follows:

Type 1: CREATE

* Clone an existing repository: git clone
* Create a new local repository: git init

Type 2: LOCAL CHANGES

* Changed files in your working directory: git status
* Changes to tracked files: git diff
* Add all current changes to the next commit: git add
* Add some changes to the next commit: git add -p
* Commit all local changes In tracked files: git commit -a
* Commit previously staged changes: git commit
* Change the last commit: git commit –amend

Type 3: COMMIT HISTORY

* Show all commits. starting with newest: git log
* Show changes over time for a specific file: git log -p
* Who changed what and when in: git blame

Type 4: BRANCHES & TAGS

* List all existing branches: git branch -av
* Switch HEAD branch: git checkout
* Create a new branch based on your current HEAD: git branch
* Create a new tracking branch based on a remote branch: git checkout – -track
* Delete a local branch: git branch -d
* Mark the current commit with a tag: git tag

Type 5: UPDATE and PUBLISH

* List all currently configured remotes: git remote -v
* Show Information about a remote: git remote show
* Add new remote repository, named remote: git remote add
* Download all changes from but don’t integrate into HEAD: git fetch
* Download changes and directly merge/integrate into HEAD: git pull
* Publish local changes on a remote: git push
* Delete a branch on the remote: git branch -dr
* Publish your tags: git push –tags

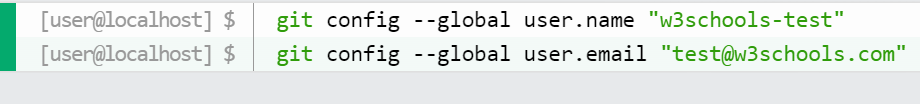
Type 6: MERGE & REUSE

* Merge into your current HEAD: git merge
* Rebase your current HEAD onto git rebase
* Abort a rebase: git rebase – -abort
* Continue a rebase after resolving conflicts: git rebase – -continue
* Use your configured merge tool to solve conflicts: git mergetool
* Use your editor to manually solve conflicts and (after resolving) mark tile as resolved: git add, git rm

Type 7: UNDO

* Discard all local changes in your working directory: git reset -hard HEAD
* Discard local changes in a specific file: git checkout HEAD
* Revert a commit (by producing a new commit with contrary changes): git revert
* Reset your HEAD pointer to a previous commit and discard all changes since then: git reset –hard
* Preserve all changes as unstaged changes: git reset
* Preserve uncommitted local changes: git reset – – keep

If we already have the installation change the username and e-mail address to your own. You will probably also want to use this when registering to GitHub with the next command



Use global to set the username and e-mail for every repository on your computer.

If you want to set the username/e-mail for just the current repo, you can remove global.

mkdir makes a new directory.

cd changes the current working directory.

Now that we are in the correct directory. We can start by initializing Git!

Note: If you already have a folder/directory you would like to use for Git:

Navigate to it in command line, or open it in your file explorer, right-click and select "Git Bash here"

*INITIALIZE GIT*

Once you have navigated to the correct folder, you can initialized Git on that folder:



*GIT NEW FILES*

Create a new file using a text editor. Save it to the git folder and at the terminal collocate this:

Graphical user interface, application

Description automatically generated with medium confidence

And to check the status:

Graphical user interface, text, application

Description automatically generated

Files in git repository folder can be in one of 2 states:

* Tracked – files that Git knows about and are added to the repository
* Untracked – files that are in your working directory, but not added to the repository

When you first add files to an empty repository, they are all untracked. To get Git to track them, you need to stage them, or add them to the staging environment.

*GIT STAGING EVIROMENT*

One of the core functions of Git is the concepts of the Staging environment, and the commit.

Staged files are files that are ready to be committed to the repository you are working on.

For add a file to the Staging environment use:



So check the status:

Graphical user interface, text

Description automatically generated

We can also stage more than one file at a time.

It’s recommended to add a README.md file that describes the repository.

A basic external style sheet of css.

To add all files in the current directory to the staging environment:



Using **--all** instead of individual filenames will **stage** all changes (new, modified, and deleted) files.

The shorthand command for **git add –all** is **git add -A**

*GIT COMMIT*

Adding commits keep track of our progress and changes as we work. Git considers each commit change point or “save point”. It is a point in the project you can go back to if you find a bug, or want to make a change.

When we commit, we should always include a message.

By adding clear messages to each commit, it is easy for yourself (and others) to see what has changed and when.

Text

Description automatically generated

The **commit** command performs a commit, and the **-m “message”** adds a message.

The staging environment has been committed to our repo, with the message:

“First release of hello world!”.

Sometimes, when you make small changes, using the staging environment seems like a waste of time. It is possible to commit changes directly, skipping the staging environment. The **-a** option will automatically stage every changed, already tracked file.

If we make a change to a file that we already have, and then we check the status of our repository using the –short option, we will see the changes in a more compact way like this:

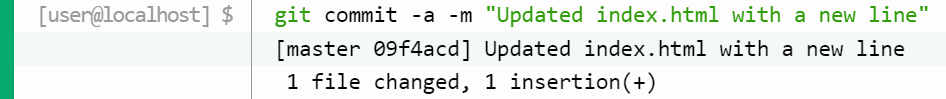
Graphical user interface, application

Description automatically generated with medium confidence

Note: Short status flags are:

* ?? - Untracked files
* A - Files added to stage
* M - Modified files
* D - Deleted files

Once we see our file modified, we proceed to commit it directly:



To view the history of commits for a repository, you can use the **long** command:

Text

Description automatically generated with medium confidence

*GIT BRANCHES*

In Git, a **branch** is a new/separate version of the main repository.

If we had a large project and are interested in updating the design of it, in Git, that would work like:

* With a new branch called new-design, edit the code directly without impacting the main branch
* EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
* Create a new branch from the main project called small-error-fix
* Fix the unrelated error and merge the small-error-fix branch with the main branch
* You go back to the new-design branch, and finish the work there
* Merge the new-design branch with main (getting alerted to the small error fix that you were missing)

Branches allow to work on different parts of a project without impacting the main branch.

When the work is complete, a branch can be merged with the main project.

You can even switch between branches and work on different projects without them interfering with each other.

Branching in Git is very lightweight and fast!

New branch, imagine we are working in our local repository, and we do not want to disturb or possibly wreck the main project.

So, we create a new branch:



Now that created a new branch called “hello-world-images”

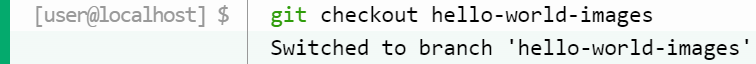
Let’s confirm that we have created a new branch:

Table

Description automatically generated with low confidence

We can see the new branch with the name “hello-world-images”, but the \* beside master specifies that we are currently on that branch.

**Checkout** is the command used to check out a branch. Moving us from the current branch, to the one specified at the end of the command:



Now we have moved our current workspace from the master branch to the new branch.

If one file is edited and add an image to the working folder and a line of code in the **.html** file.

Then, after changing the file and added a new file in the working directory (same directory as the main branch).

If then check the status of the current branch:

Graphical user interface, text, application

Description automatically generated

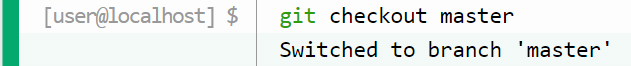
Stage the file and commit

Text

Description automatically generated

*MERGE BRANCHES*

First, we need to change the master branch:



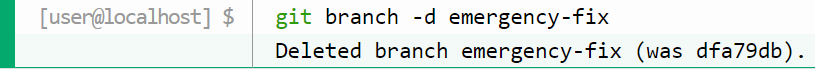
Then, we merge the current branch (master) with another one called emergency-fix:

Text

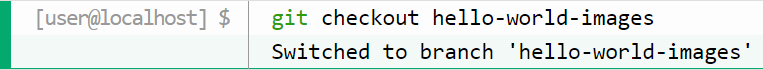
Description automatically generated with medium confidence

Since emergency-fix branch came directly from master, and no other changes had been made to master while we were working, Git sees this as a continuation of master. So, it can “Fast-forward”, just pointing both master and emergency-fix to the same commit.

As master and emergency-fix are essentially the same now, we can delete emergency-fix, as it is no longer needed:



Now we can move over to “hello-world-images” and keep working. Add another image file (img\_hello\_git.jpg) and change index.html, so it shows:



After doing what we requested. We are done with the work, and we can stage and commit for this branch:

A screenshot of a computer

Description automatically generated with medium confidence

In this example index.html has been changed in both branches. Now, we’ll try to merge “hello-world-images” into master.

Text

Description automatically generated

But merge fails, this is because there is a conflict between the versions for index.html. Let’s check the status:

Table

Description automatically generated with low confidence

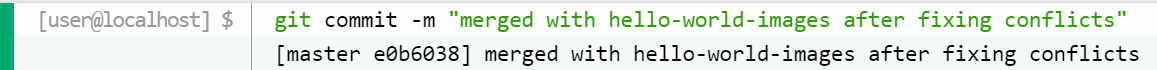
That confirm there is a conflict in index.html, but the images file is ready and staged to be committed.

So, we need to fix that by checking the differences between versions and actualize the file index.html

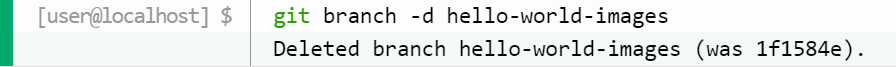
Graphical user interface, text, application

Description automatically generated

The only thing that is pending is to commit to conclude the merge



And delete the branch “hello-world-images



Merge in Git allows you to join two or more development work created using git branch into a single branch. It incorporates the changes from named commits and diverges them into the current branch.

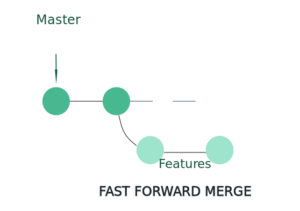
Git provides various methods to merge different commits into a base commit. These methods are called Merge Strategies. These base commits are combined to form merge commits. A merge commit is just like the regular commit except that it has two parent commits. Out of multiple strategies for the merging process, the git will automatically choose one if not specified explicitly. This automatic selection to the merge strategy is based on the branches provided for merging.

There are various types of merge strategies:

* Fast forward
* Recursive
* Ours
* Octopus
* Resolve
* Subtree

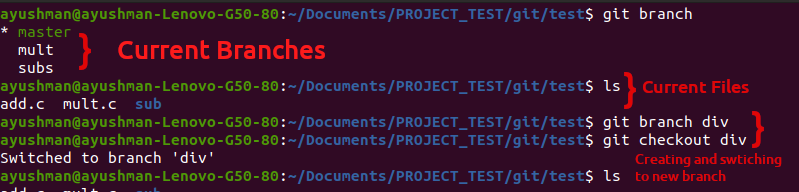
Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches. Any of the given strategies can be used to perform the merging process according to the needs of the project. The most used strategies are “Fast forward merge” and “Recursive merge”.

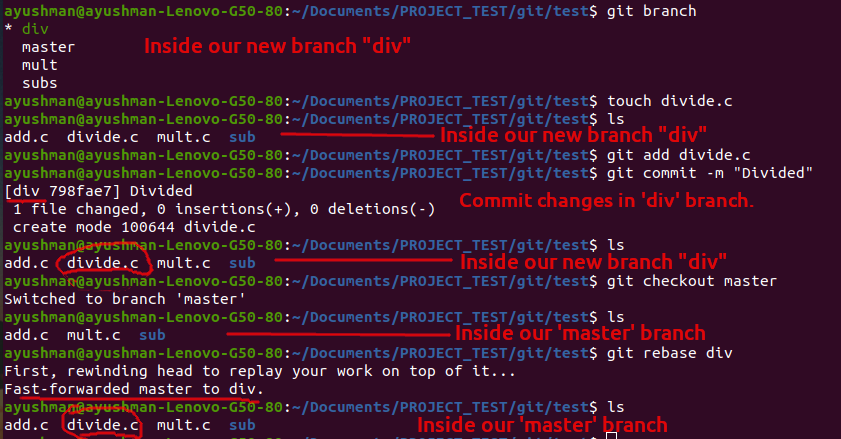
**Fast forward merge.**



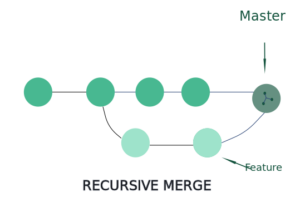
In this commonly used merge strategy, history is just one straight line. When you create a branch, make some commits in that branch, the time you’re ready to merge, there is no new merge on the master. That way master’s pointer is just moved straight forward and history is one straight line.

Command: $ git rebase



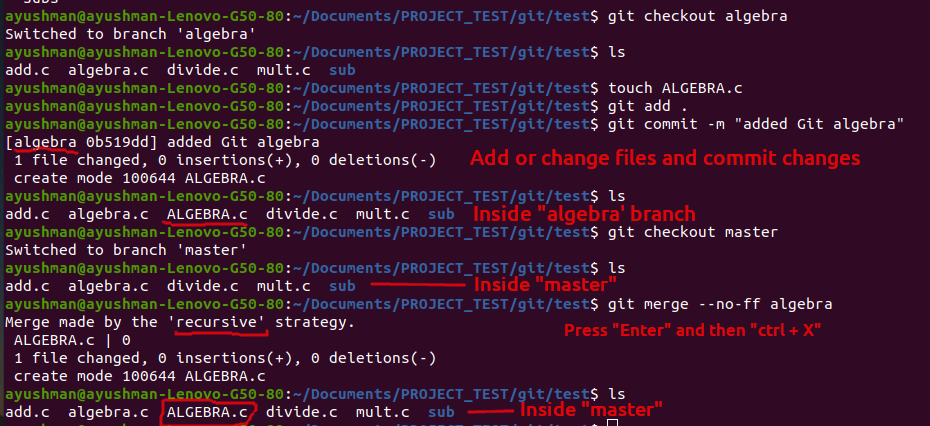


**Recursive merge.**

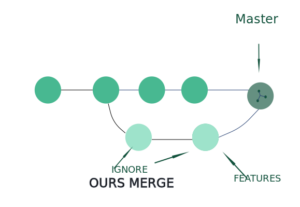


Here, after you branch and make some commits, there are some new original commits on the master. So, when it’s time to merge, git recurses over the branch and create a new merge commit. The commit continues to have two parents.

Command: $ git merge—no-ff

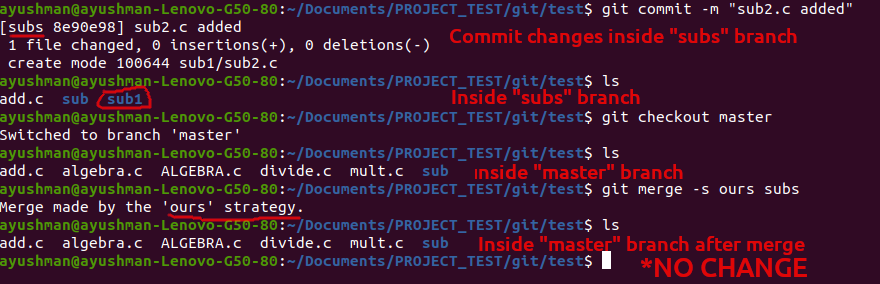


**Ours merge.**

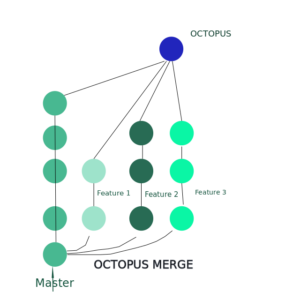


This merge strategy resolves any number of heads, but the resulting tree of the merge is always that of the current branch head, effectively ignoring all changes from all the other branches. It is meant to be used to supersede the old development history of side branches.

Command: $ git merge -s ours

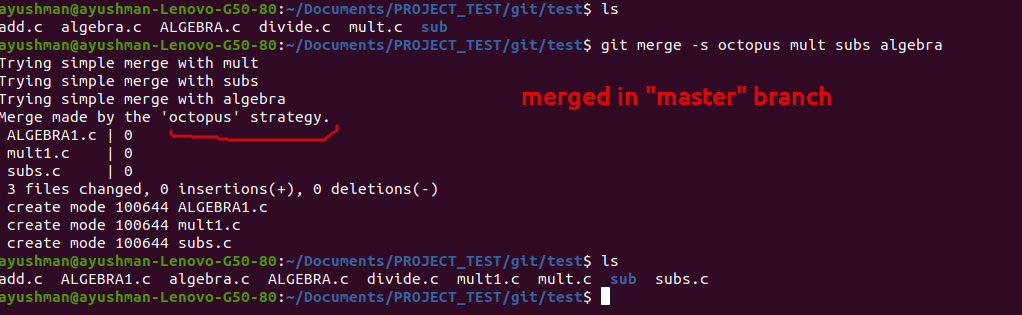


**Octopus merge.**

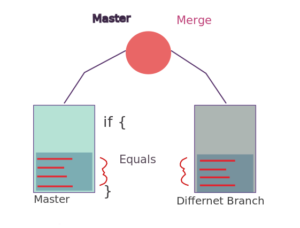


Octopus merge strategy resolves cases with more than two heads but refuses to do a complex merge that needs manual resolution. This is the default merge strategy when pulling or merging more than one branch.

Command: $git merge -s octopus

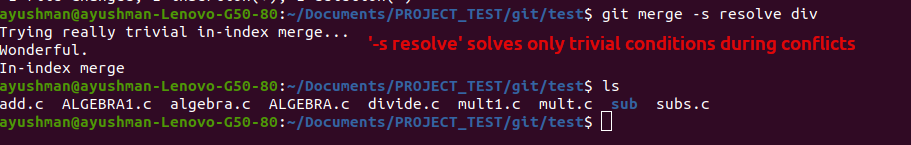


**Resolve merge.**



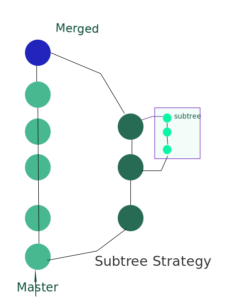
This strategy can only resolve two heads (i. e. the current branch and another branch you pulled from) using a 3-way merge algorithm. It tries to carefully detect criss-cross merge ambiguities is considered generally safe and fast.

Command: $ git merge -s resolve



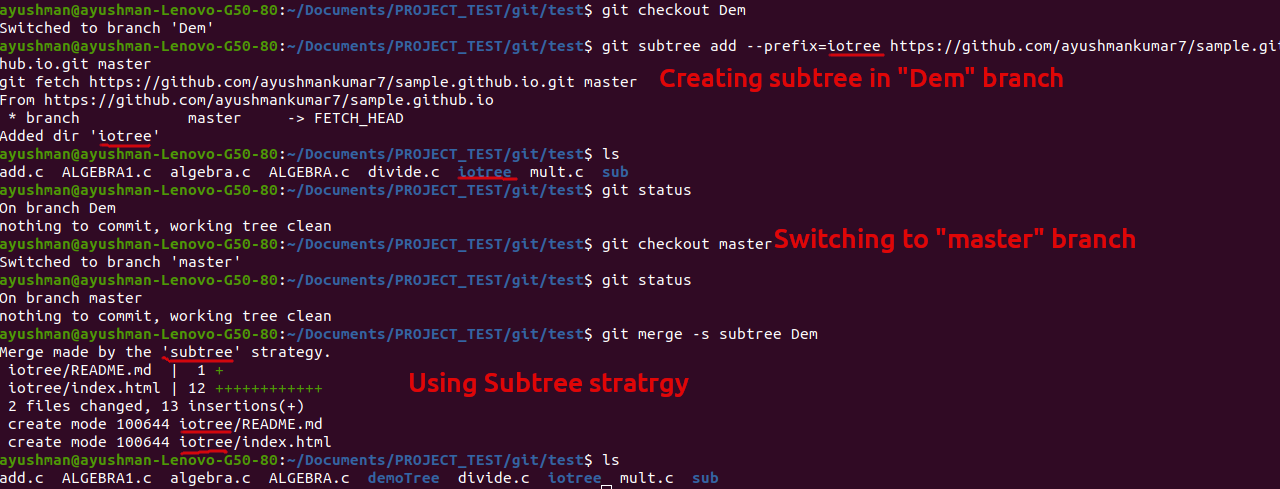
**-s solve** solves only trivial conditions. If code differs between branches, the conflict has to be solved manually.

**Subtree merge.**



This is a modified recursive strategy. When merging trees A and B, if B corresponds to a subtree of A, B is first adjusted to match the tree structure of A, instead of reading the trees at the same level. This adjustment is also done to the common ancestor tree.

Command: $ git merge -s subtree

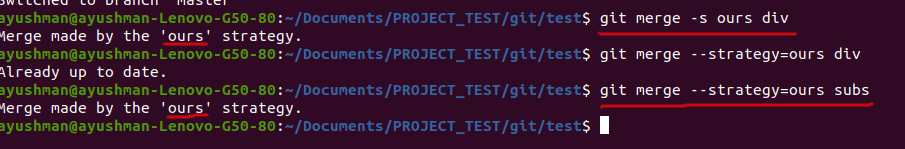


**Manually call named merge strategy**

-s<Strategy> and -strategy=<Strategy>: These strategies can be supplied more than once to specify them in the order they should be tried. If there is no -s option, a built-in list of strategies is used instead of (git merge-recursive) when merging a single head, git merge-octopus otherwise).

Command: $ git merge -s recursive

$ git merge –strategy=octopus



*GIT REVERT*

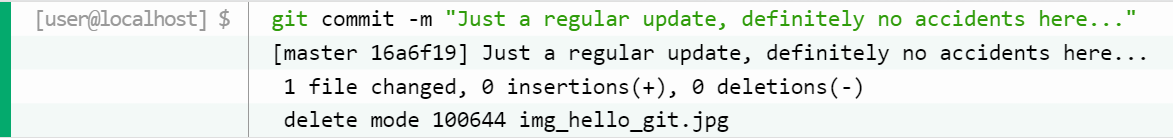
**revert** is the command we use when we want to take a previous **commit** and add it as a new **commit,** keeping the log intact.

This are the steps to do it:

* Find the previous commit
* Use it to make a new commit

For example:

Commit, where you deleted a file by mistake:



*Git revert find commit in log*

First thing, we need to find the point we want to return to. To do that, we need to go through the log. To avoid the very long log list, we are going to use the –oneline option, which gives you just one line per commit showing:

* The first seven characters of the commit hash
* The commit message

So, finding the point we want to revert:

Text

Description automatically generated

We want to revert to the previous commit: 52418f7 (HEAD -> master) Just a regular update, definitely no accidents here..., and we see that it is the latest commit.

*GIT REVERT HEAD*

We revert the latest commit using git revert HEAD (revert the latest change, and then commit), adding the option --no-edit to skip the commit message editor (getting the default revert message):

Text

Description automatically generated

We check the log again:

Text

Description automatically generated

To revert to earlier commits, use git revert HEAD~x (x being a number. 1 going back one more, 2 going back two more, etc.)

*GIT RESET*

Git reset is used when we want to unstage a file and bring our changes back to the working directory git reset can also be used to remove commits from the local repository.

Command: git reset HEAD <filename>

For example:

Whenever we unstage a file, all the changes are kept in the working area.

Text

Description automatically generated

We are back to the working directory, where our changes are present but the file is now unstaged. Now there are also some commits that we don’t want to get committed and we want to remove them from our local repository. To see how to remove the commit from our local repository let’s stage and commit the changes that we just did and then remove that commit.

Text

Description automatically generated

We have 2 commits now, with the latest being the Added Hello World commit which we are going to remove. The command that we would be using now is -

git reset HEAD~1

Points to be noted –

* HEAD~1 here means that we are going to remove the topmost commit or the latest commit that we have done.
* We cannot remove a specific commit with the help of git reset , for ex : we cannot say that we want to remove the second commit or the third commit , we can only remove latest commit or latest 2 commits … latest N commits.(HEAD~n) [n here means n recent commits that needs to be deleted].

Text

Description automatically generated

After using the above command, we can see that our commit is being deleted and also our file is again unstaged and is back to the working directory. There are different ways in which git reset can keep your changes.

* git reset –soft HEAD~1 – This command will remove the commit but would not unstage a file. Our changes still would be in the staging area.
* git reset –mixed HEAD~1 or git reset HEAD~1 – This is the default command that we have used in the above example which removes the commit as well as unstages the file and our changes are stored in the working directory.
* git reset –hard HEAD~1 – This command removes the commit as well as the changes from your working directory. This command can also be called destructive command as we would not be able to get back the changes so be careful while using this command.

Points to keep in mind while using git reset command –

* If our commits are not published to remote repository, then we can use git reset.
* Use git reset only for removing commits that are present in our local directory and not in remote directory.
* We cannot remove a specific commit with the help of git reset, for ex : we cannot say that we want to remove the second commit or the third commit , we can only remove latest commit or latest 2 commits … latest N commits.(HEAD~n) [n here means n recent commits that needs to be deleted].
* We just discussed above that the git reset command cannot be used to delete commits from the remote repository, then how do we remove the unwanted commits from the remote repository The command that we use here is –

*GIT CHECKOUT*

It is used to discard the changes in the working repository.

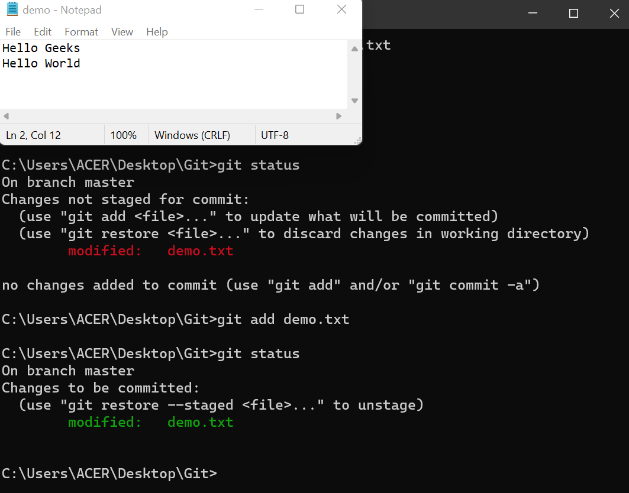
Command: git checkout <filename>

Example:

Graphical user interface, text, application

Description automatically generated

When we write git checkout command and see the status of our git repository and also the text document we can see that our changes are being discarded from the working directory and we are again back to the test document that we had before. Now, what if we want to unstage a file. We stage our files before committing them and at a certain point, we might want to unstage a file. Let’s add Hello World again to our text document and stage them using the git add command.



We want to unstage a file and the command that we would be using to unstage our file is –

Difference Table

|  |  |  |
| --- | --- | --- |
| git checkout | git reset | git revert |
| Discards the changes in the working repository. | Unstages a file and bring our changes back to the working directory. | Removes the commits from the remote repository. |
| Used in the local repository. | Used in local repository. | Used in the remote repository. |
| Does not make any changes to the commit history. | Alters the existing commit history, | Adds a new commit to the existing commit history. |
| Moves HEAD pointer to a specific commit. | Discards the uncommitted changes. | Rollbacks the changes which we have committed. |
| Can be used to manipulate commits or files. | Can be used to manipulate commits or files. | Does not manipulate your commits or files. |

*SYNCING FORKS IN GIT AND GITHUB*

Diagram

Description automatically generated

Suppose you already have a Forked Repository(on Github) which you want to sync with the Original Repository(from which you forked).

If you haven’t cloned the Forked Repo on your Local machine then just use the following command:

git clone https://github.com/<Username>/<Repository-name>.git <filename>

# filename: It is the name of file/folder for the Cloned Repository on your Local Machine.

Now we will add the Original repository as a remote repository(upstream):

git remote add upstream https://github.com/<Owner>/<Repository>.git

# add the original repository as remote repository called “upstream”

Now we will fetch the changes from the upstream repository:

git fetch upstream

# Fetch the branches and their respective commits from the upstream repository.

We will now switch to the master branch of the fork’s local repository and merge the changes in the upstream repository to the local repo’s master branch:

git checkout master

# switch to the master branch of your fork’s local repo

git merge upstream/master

# Merge the changes from upstream/master into your local master branch.

This brings your fork’s local repository in sync with the original repository. However, if you also want to update the Forked repository on GitHub then you would have to push the master branch (or the branch in which you merged the changes of the upstream repo).

git push origin master

# origin: forked repository on Github

And that’s it! Your Forked Repository on Github is also now synced with the Original Repository!!

***BEST PRACTICES***

* Formalize Git conventions for your team

Everyone should follow standard conventions for branch naming, tagging, and coding. Every organization has standards or best practices, and many recommendations are freely available on the internet. What's important is to pick a suitable convention early on and follow it as a team.

Also, different team members will have different levels of expertise with Git. You should create and maintain a basic set of instructions for performing common Git operations that follow the project's conventions.

* Merge changes properly

Each team member should work on a separate feature branch. But even when separate branches are used, everyone eventually modifies some common files. When merging the changes back into the master branch, the merge typically will not be automatic. Human intervention may be needed to reconcile different changes made by two authors to the same file. This is where you have to learn to deal with Git merge techniques.

Modern editors have features to help with [Git merge conflicts](https://opensource.com/article/20/4/git-merge-conflict). They indicate various options for a merge in each part of a file, such as whether to keep your changes, the other branch's changes, or both. It may be time to pick a different code editor if yours doesn't support such capabilities.

* Rebase your future branch often

As you continue to develop your feature branch, rebase it against master often. This means executing the following steps regularly:

git checkout master  
git pull  
git checkout feature-xyz  # name of your hypothetical feature branch  
git rebase master  # may need to fix merge conflicts in feature-xyz

These steps [rewrite history](https://opensource.com/article/20/4/git-rebase-i) in your feature branch (and that's not a bad thing). First, it makes your feature branch look like master with all the updates made to master up to that point. Then all your commits to the feature branch are replayed on top, so they appear sequentially in the Git log. You may get merge conflicts that you'll need to resolve along the way, which can be a challenge. However, this is the best point to deal with merge conflicts because it only impacts your feature branch.

After you fix any conflicts and perform regression testing, if you're ready to merge your feature back into master, do the above rebase steps one more time, then perform the merge:

git checkout master  
git pull  
git merge feature-xyz

In the interim, if someone else pushes changes to master that conflict with yours, the Git merge will have conflicts again. You'll need to resolve them and repeat the regression testing.

There are other merge philosophies (e.g., without rebasing and only using merge to avoid rewriting history), some of which may even be simpler to use. However, I've found the approach above to be a clean and reliable strategy. The commit history is stacked up as a meaningful sequence of features.

With "pure merge" strategies (without rebasing regularly, as suggested above), the history in the master branch will be interspersed with the commits from all the features being developed concurrently. Such a mixed-up history is harder to review. The exact commit times are usually not that important. It's better to have a history that's easier to review.

* Squash commits before merging

When working on your feature branch, it's fine to add a commit for even minor changes. However, if every feature branch produced 50 commits, the resulting number of commits in the master branch could grow unnecessarily large as features are added. In general, there should only be one or a few commits added to master from each feature branch. To achieve this, squash multiple commits into one or a handful of commits with more elaborate messages for each one. This is typically done using a command such as:

git rebase -i HEAD~20  # look at up to 20 commits to consider squashing

When this is executed, an editor pops up with a list of commits that you can act upon in several ways, including pick or squash. Picking a commit means keeping that commit message. Squashing implies combining that commit's message into the previous commit. Using these and other options, you can combine commit messages into one and do some editing and cleanup. It's also an opportunity to get rid of the commit messages that aren't important (e.g., a commit message about fixing a typo).

In summary, keep all the actions associated with the commits, but combine and edit the associated message text for improved clarity before merging into master. Don't inadvertently drop a commit during the rebase process.

After performing such a rebase, I like to look at the git log one last time to make final edits:

git commit --amend

Finally, forcing an update to your remote feature branch is necessary, since the Git commit history for the branch has been rewritten:

git push -f

* Use tags

After you have finished testing and are ready to deploy the software from the master branch, or if you want to preserve the current state as a significant milestone for any other reason, create a Git tag. While a branch accumulates a history of changes corresponding to commits, a tag is a snapshot of the branch's state at that instant. A tag can be thought of as a history-less branch or as a named pointer to a specific commit immediately before the tag was created.

Configuration control is about preserving the state of code at various milestones. Being able to reproduce software source code for any milestone so that it can be rebuilt when necessary is a requirement in most projects. A Git tag provides a unique identifier for such a code milestone. Tagging is straightforward:

git tag milestone-id -m "short message saying what this milestone is about"  
git push --tags   # don't forget to explicitly push the tag to the remote

Consider a scenario where software corresponding to a given Git tag is distributed to a customer, and the customer reports an issue. While the code in the repository may continue to evolve, it's often necessary to go back to the state of the code corresponding to the Git tag to reproduce the customer issue precisely to create a bug fix. Sometimes newer code may have already fixed the issue but not always. Typically, you'd check out the specific tag and create a branch from that tag:

git checkout milestone-id        # checkout the tag that was distributed to the customer  
git checkout -b new-branch-name  # create new branch to reproduce the bug

* Make the software executable print the tag

In most embedded projects, the resulting binary file created from a software build has a fixed name. The Git tag corresponding to the software binary file cannot be inferred from its filename. It is useful to "embed the tag" into the software at build time to correlate any future issues precisely to a given build. Embedding the tag can be automated within the build process. Typically, the tag string git describe generates is inserted into the code before code compilation so that the resulting executable will print the tag string while booting up. When a customer reports an issue, they can be guided to send you a copy of the boot output.

Skipping the staging environment is not generally recommended.

Skipping the stage step can sometimes make you include unwanted changes.