GIT

Check point => adding check point on small small bit of work like as add nav bar, add bottom sheet, add toolbar, update nav bar etc

Create another checkpoint in between them and do switching between them
Also can do the combination of them.

git config --global user.name "Vikas Rana"

git config --global user.email vicay0001@gmail.com
Ls => to use to show folder directories items on the terminal
cd to change directory

(use "git add <file> ..." to update what will be committed)
(use "git restore <file> ..." to discard changes in working directory)

no changes added to commit (use " git add " and/or " git commit -a")
 git add .
 git commit -m "askdjasldk”
 git log
 cd ..

git help
Pwd ⇒ To check the current location in the terminal.
Mkdir ⇒ To make a directory.
touch file_name.txt (to create file)
rm file_name.text to delete file
rm -rf folder_name (to delete directory)
open .   => for mac to open finder
ls -a => to show hidden files

start . => for window to open explorer(to use to show folder directories items on the terminal)

Repository ⇒

A Git "Repo" is a workspace which tracks and manages files within a folder.
Anytime we want to use Git with a project,app, etc we need to create a new git repository.We can have as many repos on our machine as needed,all with separate histories and contents

Our First Git Command!
⇒ git status
gives information on the current status of a git repository and its contents
It's very useful, but at the moment we don't actually have any repos to check the status of!

Our Actual First Git Command!
Use ⇒ git init
to create a new git repository.
Before we can do anything git-related,we must initialise a repo first!
This is something you do once per project.
Initialise the repo in the top-level folder containing your project.

Avoid to display hint ⇒
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"

For More Details you can visit below website :-

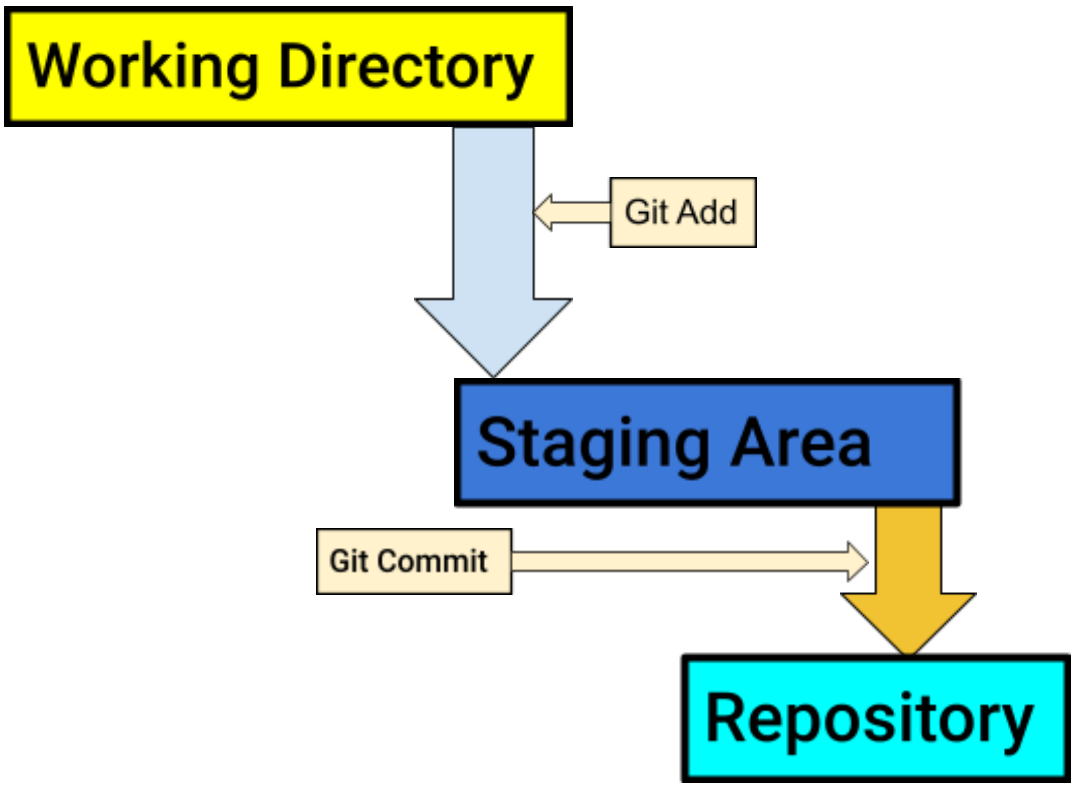
https://git-scm.com/docs

Git Tracks A Directory and All Nested Subdirectories

Important :> It's important to understand we don't want to be initialising another repo inside of an existing

Committing

The most important Git feature!



Add a Checkpoint ⇒

Adding
Use
git add ⇒
To add specific files to the staging area. Separate files with spaces to add multiple at once.

》git add file1 file2 // we can also add multiple files together

Adding
Use ⇒ git add .
to stage all changes at once

Git Commit
Running
> git commit
will commit all staged changes. It also opens up a text editor and prompts you for a commit message.
This can be overwhelming when you're starting out,so instead you can use …

MESSAGE FOR COMMIT ⇒
git commit -m " my message"

The -m flag allows us to pass in an inline commit message,rather than launching a text editor.
We'll learn more about writing good commit messages later on.

We use the git commit command to actually commit changes from the staging area.

When making a commit,we need to provide a commit message that summarises the changes and work snapshotted in the commit.

To open files directly from git bash to VS Code use
⇒ code .

NEXT WE WILL LEARN

- ☐ Git Ignore
- ☐ Writing atomic commits\
- ☐ Configuring default editor
- ☐ Writing good commit message
- ☐ Navigating the Git documentation
- ☐ Working with GUI
- ☐ Amending commit

The Git Docs
https://git-scm.com/ //reference menu & git book have a lot thing to learn.

Atomic Commit ⇒

When possible,a commit should encompass a single feature,change, or fix. In other words, Try to keep each commit focused on a single thing. This makes it much easier to undo or rollback changes later on. It also makes your code or project easier to review.

Writing
Commit Messages
Present or past tense? Does it really matter?

Present-Tense
Imperative Style ??
From the Git docs:
Describe your changes in imperative mood,e.g.
"make xyzzy do frotz" instead of "[This patch]
makes xyzzy do frotz" or "I changed xyzzy to do
frotz", as if you are giving orders to the
codebase to change its behaviour.

You do NOT have to
follow this pattern
Though the Git docs suggest using present-tense
imperative messages, many developers prefer to
use past-tense messages. All that matters is
consistency,especially when working on a team
with many people making commits

How to write a good commit message follow this link⇒

https://www.freecodecamp.org/news/writing-good-commit-messages-a-practical-guide/

Set and Config for VS Code editor follow it

https://git-scm.com/book/en/v2/Appendix-C%3A-Git-Commands-Setup-and-Config

Use this command to add credentials in vs code

To abbreviate and keep pretty details of commit use

git log -- oneline
git log --abbrev-commit

Git kraken is also a GUI to handle all command of GIT, like as source tree , github desktop etc

Amending Commits ⇒

Suppose you just made a commit and then realised you forgot to include a file! Or,maybe you made a typo in the commit message that you want to correct.

Rather than making a brand new separate commit, you can "redo"the previous commit using the --amend option.

```
> git commit -m ' some commit'
> git add forgotten_file
> git commit --amend
```

Ignoring Files

We can tell Git which files and directories to ignore in a given repository,using a .gitignore file.
This is useful for files you know you NEVER want to commit,including:
Secrets,API keys, credentials, etc.
• ● Operating System files (.DS_Store on Mac)
• Log files
• Dependencies & packages

.gitignore

Create a file called .gitignore in the root of a repository. Inside the file,we can write patterns to tell Git which files & folders to ignore:
.DS_Store will ignore files named .DS_Store.
• folderName/will ignore an entire directory.
• *.log will ignore any files with the .log extension.

If you want to ignore a file that is already checked in, you must untrack the file before you add a rule to ignore it. From your terminal, untrack the file.
Using ⇒

```
git rm --cached FILENAME
```

To know more about GitIgnore
https://docs.github.com/en/get-started/getting-started-with-git/ignoring-files

GIT BRANCHING

Every commit has its unique hash code

1. Question is how hashing is working behind the commit ?
2. What is a hash together ?
3. How is the hashing algorithm used ? its called SHA one.

Each commit has at least have one parent commit that came before it.

Contexts
On large projects,we often work in multiple contexts:
1. You're working on 2 different colour scheme variations for your website at the same time, unsure of which you like best.

2. You're also trying to fix a horrible bug,but it's proving tough to solve. You need to really hunt around and toggle some code on and off to figure it out.

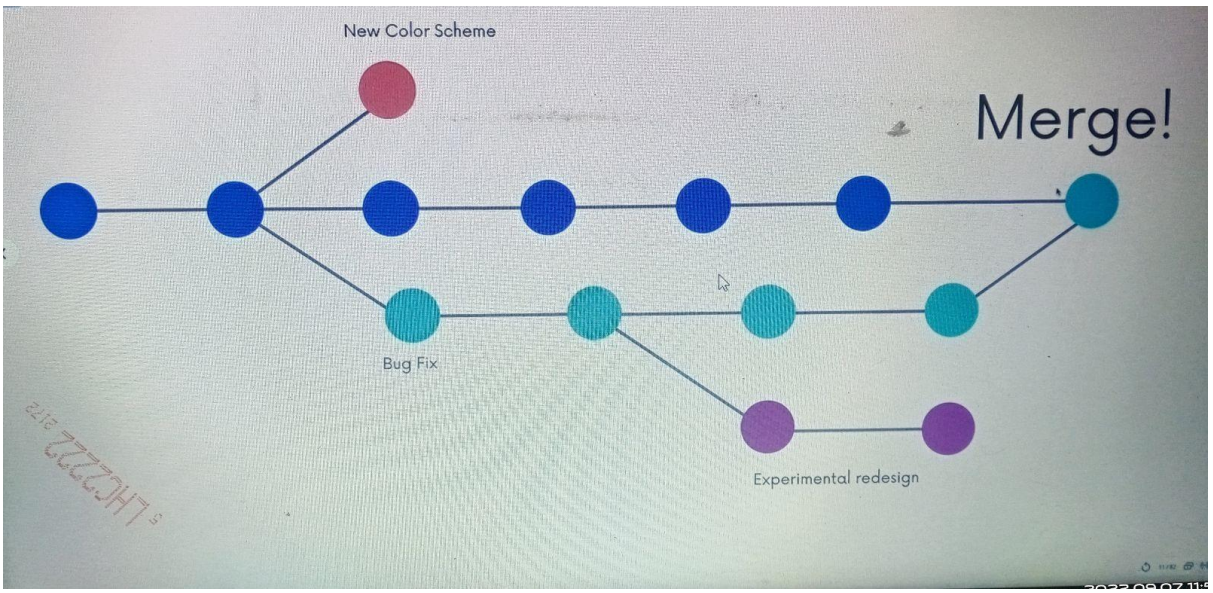
3. A teammate is also working on adding a new chat widget to present at the next meeting. It's unclear if your company will end up using it.

4.Another coworker is updating the search bar autocomplete.

5.Another developer is doing an experimental radical design overhaul of the entire layout to present next month.

Branches

Branches are an essential part of Git!
Think of branches as alternative timelines for a project.
They enable us to create separate contexts where we can try new things,or even work on multiple ideas in parallel.
If we make changes on one branch, they do not impact the other branches(unless we merge the changes)



You Probably Have
Seen This Before

On branch master nothing to commit, but on branch master something

The Master
Branch

In git,we are always working on a branch*

The

The default branch name is master.

It doesn't do anything special or have fancy powers. It's just like any other branch.

technically that's not 100%true as we'll see later

What is HEAD in Git? - Stack OverflowHEAD ⇒ master

What is HEAD their

HEAD
We'll often come across the term HEAD in Git.
HEAD is simply a pointer that refers to the current "location" in your repository. It points to a particular branch reference.
So far, HEAD always points to the latest commit you made on the master branch, but soon we'll see that we can move around and HEAD will change!
Whenever check out from one to another branch HEAD is Moving along.
And HEAD will take to the most recent working branch.

Viewing Branches

>> git branch

Use git branch to view your existing branches.
The default branch in every git repo is master, though you can configure this.

Look for the * which indicates the branch you are currently on.

Creating and Switching Branches

Creating Branches
Use
>> git branch <branch-name>
to make a
new branch based upon the current HEAD
This just creates the branch. It does not switch you to that branch (the HEAD stays the same)

Switching Branches

Once you have created a new branch,
use
>> git switch<branch-name>to switch to it.

Another way of switching ??

Historically, we used
>git checkout <branch-name>
to switch branches. This still works.
The checkout command does a million additional things,so the decision was made to add a standalone switch command which is much simpler.
You will see older tutorials and docs using checkout rather than switch. Both now work.

Creating & Switching together

Use git switch with the -c flag to create a new
git switch -c <branch-name>
branch AND switch to it all in one go.
Remember -c as short for"create"
Another way of switching & creating is
git checkout -b <branch-name>

To know more follow the link :- https://git-scm.com/docs/git-branch

Delete and Renaming Branches

-d or --delete
Delete a branch. The branch must be fully merged in its upstream branch, or in HEAD if no upstream was set with --track or --set-upstream-to.

-D Shortcut for --delete --force.

Renaming by using

>> git branch -m <new name of branch>

Merging Branches

Merging
Branching makes it super easy to work within self contained contexts, but often we want to incorporate changes from one branch into another!
We can do this using the git merge command

Merging

The merge command can sometimes confuse students early on. Remember these two merging concepts:

- We merge branches, not specific commits
- We always merge to the current HEAD branch

Merging

Made Easy

To merge,follow these basic steps:

1. Switch to or checkout the branch you want to merge the changes into (the receiving branch).

2. Use the git merge command to merge changes from a specific branch into the current branch.

To merge the bugfix branch into master ...

```
》git switch master
》git merge bugfix
```

git branch -v // use to get more information with branch list respectively

What if we add a commit on master?

This happens all the time! Imagine one of your teammates merged in a new

feature or change to master while you were working on a branch

What happens when I try to merge?
Rather than performing a simple fast forward, git performs a"merge commit"
We end up with a new commit on the master branch.
Git will prompt you for a message.

Commit can also have multiple parents in case of merging when master branch and another branch are working.
In that case Merge made by recursive strategy

Heads Up!
Depending on the specific changes your are trying to merge,Git may not be able to automatically merge.
This results in merge conflicts,which you need to manually resolve.

> CONFLICT(content): Merge conflict in blah.txt
Automatic merge failed; fix conflicts and then commit the result.

<<<<<<< HEAD
I have 2 cats
I also have chickens

==========

==========

I used to have a dog :(

>>>>>>> bug-fix

Conflict Markers

The content from your current HEAD (the branch you are trying to merge content into)is displayed between the <<<<<<< HEAD and =====

Resolving Conflicts

Whenever you encounter merge conflicts,follow these steps to resolve them:

1. Open up the file(s)with merge conflicts
2.Edit the file(s)to remove the conflicts. Decide which branch's content you want to keep in each conflict. Or keep the content from both.
3. Remove the conflict "markers" in the document
4. Add your changes and then make a commit!

Git Diff

We can use the git diff command to view changes between commits,branches,files,ou working directory,and more!

We often use git diff alongside commands like git status and git log,to get a better picture of a repository and how it has changed over time.

git diff

Without additional options, git diff lists all the changes in our working directory that are NOT staged for the next commit.

> git diff

Compares Staging Area and Working Directory

>git diff HEAD

lists all changes in the working tree since your last commit. Means staged and unstaged all files will reveals with this command.

>git diff --staged or --cached

will list the changes between the staging area and our last commit. "Show me what will be included in my commit if I run git commit right now"

Compare b/w the current head and last commit using
> git diff HEAD HEAD~ or git diff HEAD~1

Comparing Branches

>git diff branch1..branch2
will list the changes
between the tips of branch1 and branch2

Compare between branches but with single file i.e
> git diff branch1 branch2 filename

Diff-ing Specific Files

We can view the changes within a specific file by providing git diff with a filename.
> git diff HEAD [filename]
> git diff --staged [filename]

Comparing Commits

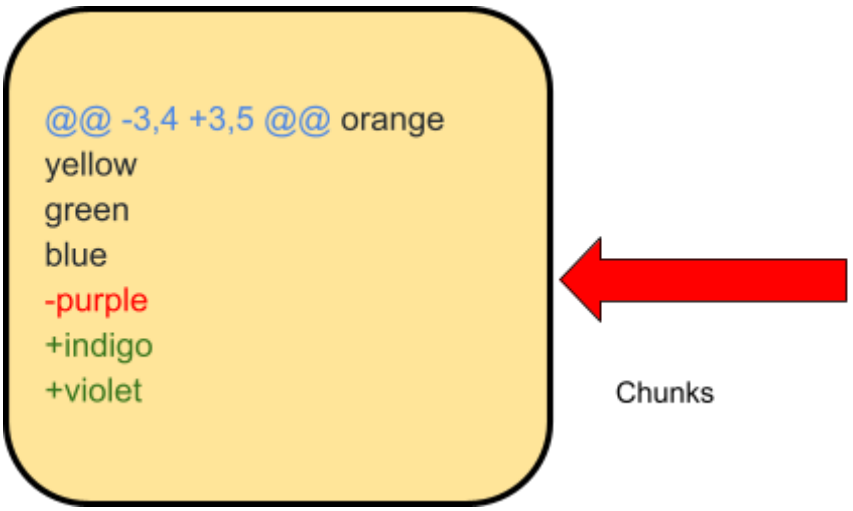
To compare two commits, provide git diff with the commit hashes of the commits in question.

》git diff commit1..commit2

Compared Files

For each comparison,Git explains which files it is comparing. Usually This is two versions of the same file. Git also declares one file as "A" and the other as "B".

git –diff a/rainbow.txt b/rainbow.txt
index 72d1d5a..f2c8117 100644   // Here Files MetaData
—— a/rainbow.txt   // here A means old committed file & assigned symbols are markers.
+++ b/rainbow.txt   // here B means new changes file without commit & assigned symbol are markers.



Chunk Header

@@ -3,4 +3,5 @@

Each chunk starts with a chunk header,found between @@ and @@
From file a,4 lines are extracted starting from line 3.
From file b,5 lines are extracted starting from line 3
As shown in the above figure.

File Metadata

You really do not need to care about the file metadata.
The first two numbers are the hashes of the two files being compared.
The last number is an internal file mode identifier.

Markers

File A and File B are each assigned a symbol.
File A gets a minus sign (-)
● File B gets a plus sign (+)

Chunks

A diff won't show the entire contents of a file,but instead only shows portions or "chunks"that were modified.
A chunk also includes some unchanged lines before and after a change to provide some context.

Changes

Every line that changed between the two files is marked with either a + or- symbol.
lines that begin with -come from file A
lines that begin with + come from file B
As shown in the above figure.

STASHING ⇒

Git provides an easy way of stashing these uncommitted changes so that we can return to them later,without having to make unnecessary commits.

Firstly :- If we return to the previous branch without stage and commit the work of current branch then they come with us to the previous working branch
So the solution is that first commit the current branch then navigate to previous branch , it avoid this problem

Secondly :- Before switching to another branch you must commit or stash changes there.

Command to use stash :-
> git stash
You can also use git stash save instead

They save your current uncommitted, your staged and unstaged changes so that you can come back to them.

git stash is super useful command that helps you save changes that you are not yet ready to commit.You can stash changes and then come back to them later. Running git stash will take all uncommitted changes (staged and unstaged)and stash them,reverting the changes in your working copy.

Use git stash pop ⇒

To remove the most recently stash
changes in your stash and re-apply them to your working copy.

Now that I have stashed my changes,I can switch branches,create new commits,etc.
I head over to master and take a look at my coworker's changes.
When I'm done,I can I can really do anything without having to
worry about those changes coming with
stashed away at any me.

Stash Apply

You can use git stash apply to apply whatever is stashed away,without removing it from the stash.This can be useful branches if you. want to apply stashed changes to
Multiple.

> git stash apply

Stashing Multiple Times :-

You can add multiple stashes onto the
stack of stashes. They will all be stashed in
the order you added them.

>git stash list ⇒ to reveal all stashes

Applying Specific Stashes :-

git assumes you want to apply the most recent stash when you run git stash apply,but you can also specify a particular stash like

>> git stash apply stash@{2}

Dropping Stashes

To delete a particular stash,you can use

git stash drop<stash-id>

i.e

git stash drop stash@{2}

Clearing The Stash

To clear out all stashes,

run

>> git stash clear

Undoing Changes & Time Travel

Brief of Checkout ⇒ Checkout

The git checkout command is like a Git Swiss Army knife. Many developers think it is overloaded, which is what lead to

the addition of the git switch and git restore commands

We can use checkout to create branches,switch to new branches,restore files,and undo history!

Detached HEAD ??

git-checkout Documentation

You are in a 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

This is all to say that HEAD usually refers to a branch NOT a specific commit.
But we can change head too for doing changes in commits

Detached HEAD

git-checkout Documentation

Don't panic when this happens! It's not a bad thing!
You have a couple options:
1.Stay in detached HEAD to examine the contents of the old commit. Poke around,view the files,etc.
2. Leave and go back to wherever you were before-reattach the HEAD
3. Create a new branch and switch to it. You can now make and save changes, since HEAD is no longer Detached.
Using ⇒

>git checkout <commit-hash>

To Detach HEAD use

>> git checkout --detach

If you want to create a new branch within a single branch then first you must detach HEAD using commit hash code & then you can create a new branch here easily..

Checkout

git checkout supports a slightly odd syntax for referencing previous commits relative to a particular commit.
HEAD~1 refers to the commit before HEAD (parent)
HEAD~2 refers to 2 commits before HEAD (grandparent)
This is not essential,but I wanted to mention it because it's quite weird looking if you've never seen it.

Referencing commit Relative to HEAD.

Discarding Changes

Suppose you've made some changes to a file but don't want to keep them. To revert the file back to whatever it looked like when you last committed,you can use:
git checkout HEAD <filename> to discard any changes in that file,reverting back to the HEAD.
Using ⇒ git checkout HEAD <file>

Another Option

Here's another shorter option to revert a file ...
Rather than typing HEAD,you can substitute -- followed by the file(s)you want to restore.

This is working same as git restore file_name.txt
& you can also use it for multiple files as shown below
git checkout —file1.txt file2.txt

Un - Modifying with Git Restore ⇒

git restore is a brand new Git command that helps with undoing operations.
Because it is so new, most of the existing Git tutorials and books do not mention it,but it is worth knowing!
Recall that git checkout does a million different things, which many git users find very confusing. git restore was introduced alongside git switch as alternatives to some of the uses for checkout.

Un modifying Files with Restore

git restore<file-name> restores using HEAD as the default source,but we can change that using the --source option.
For example,git restore --source HEAD-1 home.html will restore the contents of home.html to its state from the commit prior to HEAD. You can also use a particular commit hash as the source.

Unstaging Files with Restore

If you have accidentally added a file to your staging area with git add and you don't wish to include it in the next commit,you can use git restore to remove it from staging.
Use the --staged option like this:
git restore --staged app.js

Git Reset

Suppose you've just made a couple of commits on the master branch,but you actually meant to make them on a separate branch instead. To undo those commits,you can use git reset.
git reset <commit-hash> will reset the repo back to a specific commit. The commits are gone

Reset --hard

If you want to undo both the commits AND the actual changes in your files,you can use the --hard option.
for example,git reset --hard HEAD-1 will delete the last commit and associated changes.

Git revert

Yet another similar sounding and confusing command that has to do with undoing changes.

git revert is similar to git reset in that they both"undo" changes,but they accomplish it in different ways.
git reset actually moves the branch pointer backwards, eliminating commits.
git revert instead creates a brand new commit which reverses/undos the changes from a commit.Because it results in a new commit,you will be prompted to enter a commit message.

Which One Should I Use?

Both git reset and git revert help us reverse changes,but there is
a significant difference when it comes to collaboration (which we have yet to discuss but is coming up soon!)
If you want to reverse some commits that other people already have on their machines,you should use revert.
If you want to reverse commits that you haven't shared with others,use reset and no one will ever know!

==========GITHUB==========

What Is Github?

Github is a hosting platform for git repositories. You can put your own Git repos on Github and access them from anywhere and share them with people around the world.
Beyond hosting repos, Github also provides additional collaboration features that are not native to Git(but are super useful). Basically,Github helps people share and collaborate on repos.

Difference

Git ==>>>

Git is the version control software that runs locally on your machine. You don't need to register for an account. You don't need the internet to use it. You can use Git without ever touching Github.

Github ==>>>

Github is a service that hosts Git repositories in the cloud and makes it easier to collaborate with other people.You do need to sign up for an account to use Github. It's an online place to share work that is done using Git.

Collaboration

If you ever plan on working on a project with at least one other person, Github will make your life easier! Whether you're building a hobby project with your friend or you're collaborating with the entire world,Github is essential!

Cloning

So far we've created our own Git repositories from scratch,but often we want to get a local copy of an existing repository instead.
To do this,we can clone a remote repository hosted on Github or similar websites. All we need is a URL that we can tell Git to clone for use.

git clone

To clone a repo,simply run git clone<url>.
Git will retrieve all the files associated with the repository and will copy them to your local machine.
In addition,Git initialises a new repository on your machine,giving you access to the full Git history of the
cloned project.

Permissions?

Anyone can clone a repository from Github,provided the repo is public.You do not need to be an owner or collaborator to clone the repo locally to your machine.
You just need the URL from Github.
Pushing up your own changes to the Github repo ... that's another story entirely! You need permission to do that!

We are not limited
to Github Repos!
git clone is a standard git command.
It is NOT tied specifically to Github. We can use it to clone
repositories that are hosted anywhere! It just happens that most of the hosted repos are on Github these days.

SSH Keys

You need to be authenticated on Github to do certain operations, like pushing up code from your local machine. Your terminal will prompt you every single time for your Github email and password,unless ...
You generate and configure an SSH key!Once configured,you can connect to Github without having to supply your username/password.

CREATING YOUR FIRST GITHUB REPO

Option 1:

Existing Repo

If you already have an existing repo locally that you want to get on Github ...
• Create a new repo on Github
.• Connect your local repo (add a remote)
Push up your changes to Github
Tell your local repo about the Github repo
Then push from local repo to the new github repo

If not existing any repo on local & remote then create one

Option 2:

Start From Scratch

If you haven't begun work on your local repo,you can ...
• Create a brand new repo on Github
.Clone it down to your machine
.Do some work locally
. Push up your changes to Github

Remote

Before we can push anything up to Github,we need to tell Git about our remote repository on Github. We need to set up a"destination" to push up to.
In Git,we refer to these "destinations" as remotes. Each remote is simply a URL where a hosted repository lives.

Viewing Remotes

To view any existing remotes for you repository,we can run
git remote or git remote -v(verbose,for more info)
This just displays remotes. If you haven't added any remotes yet,you won't see anything!

Adding A New Remote

A remote is really two things: a URL and a label.
To add a new remote,we need to provide both to Git.

》git remote add <name> <url>

What is Origin?

Origin is a conventional Git remote name, but it is not at all special. It's just a name for a URL.
When we clone a Github repo,the default remote name setup for us is called origin. You can change it. Most people leave it.

…or create a new repository on the command line

```
echo "# github-demo-novel" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main // use to rename the branch name
git remote add origin
https://github.com/VikasRana007/github-demo-novel.git
git push -u origin main
```

…or push an existing repository from the command line

```
git remote add origin
  {Github Base URL}  {Profile Name} {repo_name} end with.git
https://github.com/VikasRana007/github-demo-novel.git
git branch -M main
git push -u origin main
```

After Adding Remote You need to check out

Checking Our WorkTry viewing your remotes with git remote -v,and you should now see a remote showing up! Remember,by setting up a remote we are just telling Git about a remote repository URL. We have not "communicated"with the Github repo at all yet.

Other commands for rename & remove

They are not commonly used,but there are commands to rename and delete remotes if needed.
git remote rename<old><new>

git remote remove<name>

Pushing

Now that we have a remote set up,let's push some work up to Github! To do this,we need to use the git push command.

git push <remote> <branch>

We need to specify the remote we want to push up to AND the specific local branch we want to push up to that remote.

Option 1:

Existing Repo

If you already have an existing repo locally that you want to get on Github ....

- *Create a new repo on Github*
- *Connect your local repo(add a remote)*
- *Push up your changes to Github*

An Example

git push origin master tells git to push up the

> git push origin master

master branch to our origin remote.

Push In Detail

While we often want to push a local branch up to a remote branch of the same name,we don't have to! To push our local pancake branch up to a remote branch called waffle we could do:

git push origin pancake:waffle

The -u option

The -u option allows us to set the upstream of the branch we're pushing You can think of this as a link connecting our local branch to a branch on Github. Running

git push -u origin master

sets the upstream of the local master branch so that it tracks the master branch on the origin repo.

Remote Tracking Branches

"At the time you last communicated with this rem repository,here is where x branch was pointing
They follow this pattern <remote>/<branch>.
origin/master references the state of the master branch on the remote repo named origin.
upstream/logoRedesign references the state of the logoRedesign branch on the remote named upstream (a common remote name)

Remote Branches

Run ⇒ git branch -r

To view the remote branches our local repository knows about.

You can checkout these remote branch pointers

> git checkout origin/master
Note: switching to ' origin/master' .
You are in ' detached HEAD ' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this blah blah blah blah

Working With Remote Branches

Remote Branches

Once you've cloned a repository,we have all the data and Git history for the project at that moment in time.However,that does not mean it's all in my workspace!The github repo has a branch called puppies, but when I run git branch I don't see it on my machine! All I see is the master branch. What's going on?

So that to check Remote branches
Use⇒ git branch -r

By default,my master branch is
already tracking origin/master.
I didn't connect these myself!

I want to work on the puppies branch locally!

I could checkout origin/puppies, but that puts me in detached HEAD.

I want my own local branch called puppies,and I want it to be connected to origin/puppies,just like my local master branch is connected to origin/master.

It's super easy!

Run

git switch <remote-branch-name>

to create a new local branch from the remote branch of the same name.

git switch puppies makes me a local puppies branch AND sets it up to track the remote branch origin/puppies.

NOTE! Old way of switching b/w remote branches

The new command git switch makes this super easy to do! It used to be slightly more complicated using git checkout 》git checkout --track origin/puppies

Fetching

Fetching allows us to download changes from a remote repository,BUT those changes will not be automatically integrated into our working files.It lets you see what others have been working on,without
having to merge those changes into your local repo.
Think of it as "please go and get the latest information from Github,but don't screw up my working directory."

Git Fetch

The git fetch <remote> command fetches branches and history from a specific remote repository. It only updates remote tracking branches.
git fetch origin would fetch all changes from the origin remote repository.

》 git fetch < remote>

If not specified,<remote> defaults to origin

We can also fetch a specific branch from a remote using

git fetch<remote> <branch>

For example,git fetch origin master would retrieve the latest information from the master branch on the origin remote repository.

git fetch < remote> <branch>

Pulling ⇒

git pull is another command we can use to retrieve changes from a remote repository. Unlike fetch,pull actually updates our HEAD branch with whatever changes are retrieved from the remote.
"go and download data from Github AND immediately update my local repo with those changes"

git pull = git fetch + git merge

update the remote tracking branch update my current branch with the latest changes from the whatever changes are on the remote remote repository tracking branch

git pull

To pull,we specify the particular remote and branch we want to pull using
>> git pull <remote> <branch>.
Just like with git merge,it matters WHERE we run this command from. Whatever branch we run it from is where the changes will be merged into.

git pull origin master would fetch the latest information from the origin's master branch and merge those changes into our current branch.

An even easier syntax!

If we run git pull without specifying a particular remote

or branch to pull from,git assumes the following:
• remote will default to origin
• branch will default to whatever tracking connection is configured for your current branch.
Note:this behaviour can be configured,and tracking connections can be changed manually. Most people don't mess with that stuff

is > git pull

Comparison FETCH & PULL

git fetch
Gets changes from remote branch(es)
• Updates the remote-tracking branches with the new changes
• Does not merge changes onto your current HEAD branch
• Safe to do at anytime

git pull
• Gets changes from remote branch(es)
• Updates the current branch with the new changes,merging them in
• Can result in merge conflicts
• Not recommended if you have uncommitted changes!

Public & Private REPOs

Public

Public repos are accessible to everyone on the internet. Anyone can see the repo on Github

Private

Private repos are only accessible to the owner and people who have been granted access.

Collaborations

READMES

A README file is used to communicate important information about a repository including:

-What the project does

- How to run the project

Why it's noteworthy

-Who maintains the project

==Note==

If you put a README in the root of your project, Github will recognize it and automatically display it on the repo's home page.

README.md

READMES are Markdown files, ending with the .md extension. Markdown is a convenient syntax to generate formatted text. It's easy to pick up!