**Git & GitHub**

**Ref:** <https://www.youtube.com/watch?v=sBTAkHOxvOk>

**Ref:** Git and GitHub Beginner Tutorial by Raghav Pal

**Introduction:**

1. What is GIT?

* VCS - version control system
* To track changes in files / folders
* To collaborate in teams
* Free and open source

Git - Centralised VCS | Distributed VCS …?



Centralised VCS



Distributed VCS (DVCS)

**GIT = DVCS**

2. What is GIT HUB?

* website to upload your repositories online
* provides backup
* provides visual interface to your repo
* makes collaboration easier

3. Is GIT related to GIT HUB?

**GIT != GIT HUB**

4. Simple work flow of Git………..!!!



**Install git on windows:**

**Step - 1:** Check if git installed or not

**Step - 2:** Download and install git (while installing > if windows prompt is selected, we can use both bash & cmd; can use only bash if it is selected)

**Step - 3:** Add your project to git (goto project location which is to be added to git > right click > git bash here)

**Step - 4:** Check the following commands & can be used from git bash or from command prompt.

* git config --global user.email "abcd@gmail.com"
* git config --global user.name "usrnme"
* git --version
* git init
* git status
* git add (adding to staging area)
* git commit -m "any message"

Create an account for git hub & create a repository > **Repo1**:

Adding project to remote repository (i.e., github)

* git remote add origin https://github.com/doddam/Repo1.git > copy url from git hub
* git push -u origin master > pushes data into repository
* git log
* git --help

**Branching and Merging:**

1. What are branches?

2. How to create branch?

3. How to checkout branch?

4. How to merge branch to master?

5. How to delete branch (local and remote)?

**About Branching and Merging:**

In a collaborative environment, it is common for several developers to share and work on the same source code. Some developers will be fixing bugs while others would be implementing new features. Therefore, there has got to be a manageable way to maintain different versions of the same code base.

This is where the branch function comes to the rescue. Branch allows each developer to branch out from the original code base and isolate their work from others. Another good thing about branch is that it helps Git to easily merge the versions later on.

Simply, it can be said as an **independent line of development** as one can take advantage of branch when working on new features or bug fixes as it helps to isolate your work from that of other team members.



* Different branches can be merged into any one branch provided that they belong to the same repository.
* The diagram below illustrates how development can take place in parallel using branches.
* Changes in the primary branch or other branches will not affect your branch, unless you decide to pull the latest changes from those branches.
* It is a common practice to create a new branch for each task (eg. bug fixing, new features etc.), which is a good practice because it allows others to easily identify what changes to expect, and also for backtracking purposes to understand why a particular code change is implemented.



**Master branch:**

Upon making the first commit in a repository, Git will automatically create a master branch by default. Subsequent commits will go under the master branch until you decide to create and switch over to another branch.



**Create branch:**

Let's create a new branch with the name "issue1".

Use the branch command with a name to create a new branch with that name.

$ git branch <branchname>

Create a new branch named issue1.

$ git branch issue1

If you do not specify any parameters, the branch command will list all branches that correspond to this repository. The asterisk indicates the current active branch.

$ git branch

issue1

\* master

At this point, the history tree should look like this,



**Merge branches:**

Let's merge "issue1" with "master"



Use the merge command to merge branches.

$ git merge <commit>

By running the command above, the specified commit will be merged to the current active branch. Most of the time, you will want to merge a branch with the current active branch and you can do so by passing in the branch name in <commit>.

To merge commits into the master branch, let's now switch over to the master branch.

$ git checkout master

Switched to branch 'master'

# Delete branch:

Now that "issue1" has been successfully merged with "master", we can delete it.

We can delete a branch by calling the branch command and passing in the -d option, followed by the branch name.

$ git branch -d <branchname>

Run the following command to delete "issue1".

$ git branch -d issue1

Deleted branch issue1 (was b2b23c4).

We can verify that "issue1" has been deleted by calling "git branch". Only the master branch should be listed.

$ git branch

\* master



**Steps to be followed for branching & merging:**



Do not to modify the code in master branch, any changes need to be done to the code from git follow the below steps:

Create other branch > push the code or make changes required > test it and validate it > if ok > then merge this branch to master branch

git branch "branch name" > cmd to create a new branch

* git checkout "brnach name" > to start working out with the branch
* touch test2.txt > create a new file
* git status
* git add . | git add “file name” | git add \*.\* (with all extensions)
* git commit -m "test2.txt is added"
* git push -u origin MyNewBranch

[test2.txt file is added to the new branch but not to the master branch,

open github and check that the branch is created,

open the project path in local system and check you can see test2.txt is added]

* git checkout master > test2.txt file is invisible
* git merge MyNewBranch > when you are merging new branch to master branch we need to check out to master branch and then merge it
* git push -u origin master

**Faced an error:**

! [rejected] master -> master (fetch first)

error: failed to push some refs to ''

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

**Solved it using the git command:**

* “git push origin master - -force | git push origin master –f”
* git branch -d "branch name" --> delete branch from local (system)
* git push origin --delete "branch name" --> delete from remote (github)

**How to send email from GitHub:**

How to trigger notification email from GitHub whenever there is any change/commit in the project?

**Step-1:** GitHub > Repository > Settings > integration & services > add email

**Step-2:** Test and validate by making some change in the project

**Check the below screenshots:**

I have the following repositories in git hub,



Click on any of the repository... I have done with “hello-world” repo> go to settings



Goto options tab > select integrations & services 

Click on “Add service”



Search for the service “Email” in text box



Once you select the Email service > you need to fill the following with Email address in Address tab > Tick the checkbox - Send from author > click on Add service

Now you can use the Email service once the service is added.



**Test the scenario whether service is running successfully or not:**

Make any changes to your repository just to test and validate > “hello-world”

I have added a new file in the repo > committed > check email that you have configured in settings > a notification email is triggered from GitHub as we have made a change in the project.



**Branches versus Tags**

The workspace is (almost always) associated with a branch, called master by default. When it is, a commit will automatically update the master reference to point to that new commit; in other words, branches are **mutable references** (changeable object).

**Tag:**

A tag, on the other hand, is created “to point to a specific commit” and thereafter does not change, even if the branch moves on. In other words, tags are **immutable references** (unchangeable object).

Tags are created using git tag are the base for the commit identifiers git describe creates. In another words, in Git you don't tag branches. You are tagging commits. It is correct to say that tag is just an annotated pointer to a commit.

**Annotated Tags:**

Git has two flavors of tags:

1. Annotated : lightweight tag
2. Non-annotated : can give a message and all notes available in description below completely stored as a git object in git repository

When using them, there is little difference between the two; both will allow you to refer to a specific commit in a repository.

An annotated tag creates an additional tag object in the Git repository, which allows you to store information associated with the tag itself. This may include release notes, the meta-information about the release, and optionally a signature to verify the authenticity of the commit to which it points.

**Step-1:** Open git bash and go to local directory, checkout the branch where you want to create the tag

git checkout <branch name>

git checkout master

**Step-2:** Create tag with some name

git tag <tag name>

git tag v1.0

**Step-3:** Check whether tag is created or not



**Step-4:** Creating annotated tag and check for git tag command.

git tag -a <tag name> -m “message”

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



Where, v1.0 – a lightweight tag v1.1 – annotated tag

Once tags are created, you can view them in the repository.

Go torepository > click on releases > check the below screenshot



Click on that zip file > download > open and check the code that is taken backup.

**Commands to display or show tags:**

1. git tag
2. git show v1.0
3. git tag - - l “v1.\*”



Displaying all tags using wild cards:



**How to push tags to remote?**

**Commands to push tags to remote:**

1. git push origin v1.0
2. git push origin - -tags
3. git push - - tags



Goto the repository > refresh it > check in releases section



Can use second and third option to push all tags at a time to remote.



**Delete tags from local:**

git tag –d v1.4 (or)

git tag - -delete v1.3



Tags are deleted from local but still tags existed on remote repository.

**Delete tags from remote:**

git push origin –d v1.4

git push origin - -delete v1.4

git push origin :v1.4





Now check the repository tags are deleted on remote:



**To delete multiple tags:**

git tag –d v1.0 v1.1 (local)

git push origin –d v1.0 v1.1 (remote)

**Understanding the overall scenario:**



Above tagging scenario is explained here,

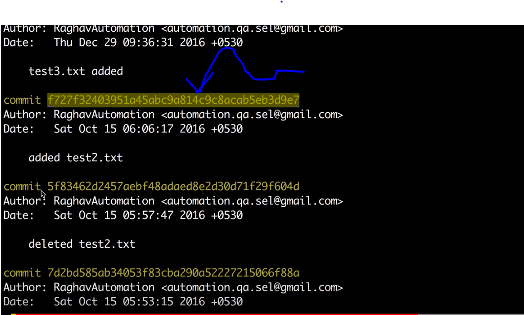
I checked out my master branch, whenever I make a change in my project I will commit,

Made two changes 1, 2 and when I think my repository is stable i.e., at third change I will take a backup or create a historic point > created a tag v1.0

Still I have changes in my project, changed and committed 4 and after 5th commit, I need to take a backup so at that point I have created a tag > created a tag v1.1

Every commit tag has a number is said to be a checksum (40-digit number)

Command:  **git log** (to check all the commits)



Arrow indicates a checksum number. Checksum number is generated by a SHA-1 algorithm

Whenever a change is made, a unique checksum number is generated.

**SHA-1 (short for *Secure Hash Algorithm 1*):** is one of several [cryptographic hash functions](https://www.lifewire.com/cryptographic-hash-function-2625832).

SHA-1 is most often used to verify that a [file](https://www.lifewire.com/what-is-a-file-2625878) has been unaltered. This is done by producing a [checksum](https://www.lifewire.com/what-does-checksum-mean-2625825) before the file has been transmitted, and then again once it reaches its destination.

The transmitted file can be considered genuine only if **both checksums are identical**.

C**ryptographic hash function:** It is a kind of algorithm that can be run on a piece of data, like an individual [file](https://www.lifewire.com/what-is-a-file-2625878) or a password, to produce a value called a checksum.

The main use of a cryptographic hash function is to verify the authenticity of a piece of data. Two files can be assured to be identical only if the checksums generated from each file, using the same cryptographic hash function, are identical.

**Checksum:** is the outcome of running an algorithm, called a [cryptographic hash function](https://www.lifewire.com/cryptographic-hash-function-2625832), on a piece of data, usually a single [file](https://www.lifewire.com/what-is-a-file-2625878).