**Git & GitHub**

**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" --> branch from local (system)
* git push origin --delete "branch name" --> delete from remote (github)

**Scenario -** *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)

**­Let’s understand some basic terms:**

**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).

**Understanding the overall scenario:**



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)

**Creating a tag for some past commit:**

We can take backup for the past commits too.

**Step-1:** Use the following command to get the reference of commit (=checksum number)

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

Whenever a change is made in the project (i.e., committed then a unique checksum number is generated).

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



**Step-2:** Take the checksum number of the commit or part of it is also enough.

**Step-3:** To create a past commit use the command,

**git tag <tag name> <reference of the commit>**

For suppose, I need to take a backup for the commit made on May 10th, 2018. Use the following commands shown in the below screenshot.

1. Create tag using checksum number
2. View it whether created or not
3. Push the tag to the remote repository.



**Step-4:** Open remote repository > click releases > check it you can see v1.4 tag is created and at that point back up is taken.



By following above steps we can create a tag for some past commit and check the backup in zip file which can be downloaded.

**Issue -** I had a problem that is after exiting git log in windows git bash, text is disappeared on git bash and unable to access it properly > press ctrl+c two times(press **:q**)

**Note:**

git init > This command creates a hidden directory called .git

git uses .git folder to track changes



git looks our project as working directory



**Quick basic navigation and keyboard shortcuts for using GitBash:**

Move to a directory - cd e:/(forward slash)

To bring cursor to the beginning - Ctrl+A

To bring cursor to the ending - Ctrl+E

To delete content to beginning from the place of cursor - Ctrl+U

To delete content till ending from the place of cursor - Ctrl+K

To clear the screen – Ctrl+L/clear

To exit the git log interface you must type “**:q**”

Listing files within directory – ls (horizontal)

ls –l > long format

ls –l r\* > displays all the files starts with letter r

ls –a > shows hidden files

ls –la or ls -al> long format with hidden files

**Scenario:** *Pull changes from your Git repository on Git hub Cloud*

(Pulling a new file from git to local repository)

**Step-1:** Create a file in Git repo (in git hub).

**Step-2:** Pull changes from a remote repository

We need to get that new file into your local repository.

To pull the file into your local repository, do the following:

Open your git bash window and navigate to your local repository.

1. cd /e
2. git clone <https://github.com/doddam/myRepo.git> - clone the repository to local
3. git pull --all enter this command to pull all the changes from git repo



The [**git pull**](https://www.atlassian.com/git/tutorials/syncing/git-pull) command merges the file from your remote repository (github) into your local repository with a single command.

**Step-3:** Navigate to your repository folder on your local system and you'll see the file you just added.

**Scenario:** *Use a Git branch to merge a file/Merging a branch into master branch using git bash*

Branches are most powerful when you're working on a team. You can work on your own part of a project from your own branch, pull updates from github, and then merge all your work into the main branch when it's ready.

A branch represents an independent line of development for your repository. Think of it as a brand-new working directory, staging area, and project history. Before you create any new branches, you automatically start out with the main branch (called master).

**Step-1: Create a new branch and make a change**

Create a branch where you can add future plans that you aren't ready to commit. When you are ready to make those plans known to all, you can merge the changes into your GitHub repository and then delete the no-longer-needed branch.

It's important to understand that branches are just pointers to commits. When you create a branch, all Git needs to do is create a new pointer - it doesn’t create a whole new set of files or folders. Before you begin, your repository looks like this:



To create a branch, do the following:

1. Go to your git bash terminal window and navigate to the top level of your local repository and create a new branch using command

$ git branch <branch\_name>

$ **git branch future-plans**

This command creates a branch but does not switch you to that branch, so your repository looks something like this:



The repository history remains unchanged. All you get is a new pointer to the current branch. To begin working on the new branch, you have to check out the branch you want to use.

Checkout the new branch you just created to start using it.

$ **git checkout future-plans**

Switched to branch 'future-plans'

The git checkout command works hand-in-hand with git branch. Because you are creating a branch to work on something new, every time you create a new branch (with git branch), you want to make sure to check it out (with  git checkout ) if you're going to use it. Now that you’ve checked out the new branch, your Git workflow looks something like this:



1. Open git bash, navigate to the local repository in the system.
2. Add a file to the new branch or just make any modifications in the existing file.
3. Enter git status in the terminal window. You will see something like this:

$ **git status**  
On branch future-plans  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)  
  (use "git checkout -- <file>..." to discard changes in working directory)  
    modified: f1.txt  
no changes added to commit (use "git add" and/or "git commit -a")

Notice the On branch future-plans line? If you entered git status previously, the line was “On branch master” because you only had the one master branch. Before you stage or commit a change, always check this line to make sure the branch where you want to add the change is checked out.

1. Stage your file.

$ **git add f1.txt**

1. Enter the git commit  command in the terminal window, as shown with the following:

$ git commit f1.txt -m “making a change in a branch”  
[future-plans e3b7732] making a change in a branch  
 1 file changed, 4 insertions(+)

With this recent commit, your repository looks something like this:



Now it's time to merge the change that you just made back into the master branch.

## **Step-2: Merge your branch > “fast-forward merging”**

You can merge your future-plans branch into the main branch on your local system.

Because you created only one branch and made one change, use the fast-forward branch method to merge.  You can do a fast-forward merge because you have a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast-forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one.



To complete a **fast-forward merge** do the following:

1. Go to your terminal window and navigate to the top level of your local repository.
2. Enter the git status command to be sure you have all your changes committed and find out what branch you have checked out.

$ **git status**   
On branch future-plans  
nothing to commit, working directory clean

1. Switch to the master branch.

$ **git checkout master**   
Switched to branch 'master'  
Your branch is up-to-date with 'origin/master'.

1. Merge changes from the future-plans branch into the master branch. It will look something like this:

$ **git merge future-plans**  
Updating fcbeeb0..e3b7732  
Fast-forward  
 f1.txt | 4 ++++  
 1 file changed, 4 insertions(+)

You've essentially moved the pointer for the master branch forward to the current head and your repository looks something like this:



1. Because you don't plan on using future-plans anymore, you can delete the branch.

**$ git branch -d future-plans**Deleted branch future-plans (was e3b7732).

When you delete future-plans, you can still access the branch from master using a commit id. For example, if you want to undo the changes added from future-plans, use the commit id you just received to go back to that branch.

1. Enter git status to see the results of your merge, which show that your local repository is one ahead of your remote repository.   
   It will look something like this:

$ git status   
On branch master  
Your branch is ahead of 'origin/master' by 1 commit.  
  (use "git push" to publish your local commits)  
nothing to commit, working directory clean

Here's what you've done so far:

* Created a branch and checked it out
* Made a change in the new branch
* Committed the change to the new branch
* Integrated that change back into the main branch
* Deleted the branch you are no longer using.

Next, we need to push all this work back up to Bitbucket, your remote repository. 

**Step-3: Push your change to GitHub**

Here's how to push your change to the remote repository:

1. From the repository directory in your terminal window, enter following command to push the changes. It will result in something like this:

$ **git push origin master**



1. Now open your git hub account > click on master branch > changes made are seen as shown in the following screenshot,



Check the following commands from the git bash terminal for easy understanding:

mdodda@GUEST152 MINGW64 /e

$ git clone https://github.com/doddam/Test.git

Cloning into 'Test'...

remote: Counting objects: 3, done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

mdodda@GUEST152 MINGW64 /e

$ cd Test

mdodda@GUEST152 MINGW64 /e/Test (master)

$ ll

total 1

-rw-r--r-- 1 mdodda 1049089 8 May 22 15:22 README.md

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git branch future-plans

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git checkout future-plans

Switched to branch 'future-plans'

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ vi f1.txt

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ git status

On branch future-plans

Untracked files:

(use "git add <file>..." to include in what will be committed)

f1.txt

nothing added to commit but untracked files present (use "git add" to track)

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ git add f1.txt

warning: LF will be replaced by CRLF in f1.txt.

The file will have its original line endings in your working directory.

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ git commit f1.txt -m "made a change in the branch"

warning: LF will be replaced by CRLF in f1.txt.

The file will have its original line endings in your working directory.

[future-plans 10cda5b] made a change in the branch

1 file changed, 2 insertions(+)

create mode 100644 f1.txt

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ git status

On branch future-plans

nothing to commit, working tree clean

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ git checkout master

Switched to branch 'master'

Your branch is up to date with 'origin/master'.

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git branch

future-plans

\* master

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git merge future-plans

Updating 5238da6..10cda5b

Fast-forward

f1.txt | 2 ++

1 file changed, 2 insertions(+)

create mode 100644 f1.txt

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git branch -d future-plans

Deleted branch future-plans (was 10cda5b).

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

nothing to commit, working tree clean

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git push origin master

Counting objects: 3, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 299 bytes | 149.00 KiB/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/doddam/Test.git

5238da6..10cda5b master -> master

**Fork:**

A fork is a copy of a repository that you manage.

Forks let you make changes to a project without affecting the original repository.

You can fetch updates from or submit changes to the original repository with pull requests.

Forking a repository is similar to copying another repository, with two major differences:

1. You can use a pull request to suggest changes from your fork to the original repository, also known as the upstream repository.
2. You can bring changes from the upstream repository to your local fork by synchronizing your fork with the upstream repository.

**Fork an example repository:**

Forking a repository is a simple two-step process. GitHub created a repository to practice with!

1. On GitHub, navigate to the link [**octocat/Spoon-Knife**](https://github.com/octocat/Spoon-Knife) repository.



1. In the top-right corner of the page, click **Fork**.



Once you fork the Spoon-Knife repo, asks for credentials of GitHub – so give the credentials and the repo is cloned from octocat account to your account as shown below.



Now, you have a *fork* of the original octocat/Spoon-Knife repository, can check your repositories list now



**Keep your fork synced:**

Sync a fork of a repository to keep it up-to-date with the upstream repository.

Before you can sync your fork with an upstream repository, you must [configure a remote that points to the upstream repository](https://help.github.com/articles/configuring-a-remote-for-a-fork) in Git.

It's good practice to regularly sync your fork with the upstream repository. To do this, you'll need to use Git on the command line. You can practice setting the upstream repository using the same [octocat/Spoon-Knife](https://github.com/octocat/Spoon-Knife) repository you just forked!

**Step-1:** Set up Git

If you haven't yet, you should first [set up Git](https://help.github.com/articles/set-up-git). Don't forget to [set up authentication to GitHub from Git](https://help.github.com/articles/set-up-git#next-steps-authenticating-with-github-from-git) as well.

**Step-2:** Create a local clone of your fork (copying all the files from fork to the local system)

Right now, you have a fork of the Spoon-Knife repository, but you don't have the files in that repository on your computer. Let's create a clone of your fork locally on your computer.

1. On GitHub, navigate to **your fork** of the Spoon-Knife repository.
2. Under the repository name, click **Clone or download**.

Clone or download button

1. In the Clone with HTTPs section, click  to copy the clone URL for the repository.



1. Open Git Bash.
2. Type git clone, and then paste the URL you copied in Step2. It will look like this, with your GitHub username instead of YOUR-USERNAME:

$ git clone <https://github.com/YOUR-USERNAME/Spoon-Knife>

$ git clone <https://github.com/doddam/Spoon-Knife> - sample link to tell “doddam” is my github username.

1. Press **Enter**. Your local clone will be created.  
   

Now, you have a local copy of your fork of the Spoon-Knife repository!

**Step-3:** Configure Git to sync your fork with the original Spoon-Knife repository

When you fork a project in order to propose changes to the original repository, you can configure Git to pull changes from the original, or *upstream*, repository into the local clone of your fork.

1. On GitHub, navigate to the [octocat/Spoon-Knife](https://github.com/octocat/Spoon-Knife) repository.
2. Under the repository name, click **Clone or download**.

Clone or download button

1. In the Clone with HTTPs section, click to copy the clone URL for the repository.



1. Open Git Bash.
2. Change directories to the location of the fork you cloned in [Step 2: Create a local clone of your fork](https://help.github.com/articles/fork-a-repo/#step-2-create-a-local-clone-of-your-fork).
3. To go to your home directory, type just cd with no other text.
4. To list the files and folders in your current directory, type ls.
5. To go into one of your listed directories, type cd your\_listed\_directory.
6. To go up one directory, type cd ...
7. Type git remote -v and press **Enter**. You'll see the current configured remote repository for your fork.



1. Type git remote add upstream, and then paste the URL you copied in Step 2 and press **Enter**. It will look like this:



1. To verify the new upstream repository you've specified for your fork, type git remote –v again. You should see the URL for your fork as origin, and the URL for the original repository as upstream.



Now, you can keep your fork synced with the upstream repository.

**What is a pull request exactly?**

Pull requests let you tell others about changes you've pushed to a GitHub repository. Once a pull request is sent, interested parties can review the set of changes, discuss potential modifications, and even push follow-up commits if necessary.

If you have [distributed version control](https://en.wikipedia.org/wiki/Distributed_revision_control) systems, every developer has a copy of the full repository. If you change something to the software, you commit your changes to your local repository. If different repositories should have these changes, you can push the changes (moving changes to another repository you have the right to write to) or pull the changes (copy revision from other repositories to your own). As many project have a main repository, a pull-request is the request that the maintainer pulls your changes.

Simple example is when you clone/fork the open source repository, do some changes/commits and in order to merge the changes into the main repository, you are required to send the pull request which consist all your commits grouped into one logical piece so it can be reviewed by other developers.

**Making changes in forked repo:**

Added a new file in the Spoon-Knife repo that is forked from my account.



Click on new pull request > navigates to the original repo and compares for the changes.



Click on create pull request and a pull request is opened as shown in the below screenshot > add the comment and again click on pull request.



Thus the new file is added to the original repository



We can see here, my file is added below



Thus the changes made in the forked repo are pulled to the main repo…. :-)

**git pull vs pull request:**

git pull is used to pull changes/files to the local repository from remote repository.

**Eg:** A file say “demo-pull” is added in remote > committed

Now check in your local you can’t find the file so type the command

“**git pull**” in order to get the new files from remote repository



Now you can check that file is in your local repository,



So finally,

If you use **git pull**, you pull the changes from the remote repository into yours.

If you send a **pull request** to another repository, you ask their maintainers to pull your changes into theirs (you more or less ask them to use a git pull from your repository).

Have seen that this is related to a **Fork and Pull** collaborative development model and is used for **code reviews**

**git pull:**

git pull fetches the latest changes of the current branch from a remote and applies those changes to your local copy of the branch. Generally this is done by merging, i.e., the local changes are merged into the remote changes.

**git pull** is shorthand for **git fetch** followed by **git merge FETCH\_HEAD.**

working directory

|=>.git

| |=>objects <= contains data for each commit

| |=>refs

| |=>heads

| |-master <= file containing current commit of local master branch

| |=>remotes

| |=>origin

| |-master <= file containing current commit of remote origin's master branch

|-FETCH\_HEAD <= file updated by `git fetch`, contains info of what was fetched

https://i.stack.imgur.com/zUInQ.png

**fig:** git fetch, git merge and git pull

**git fetch** just "downloads" the changes from the remote to your local repository. git pull downloads the changes and merges them into your current branch.

**git pull** vs **git fetch**:

**git pull** - pulls from a remote branch and merges it.

**git fetch** - only fetches from the remote branch but it does not merge.

i.e., git pull = (git fetch + git merge)

If you run **git pull**, you do not need to merge the data to local.

If you run **git fetch**, it means you must run **git merge** for getting the latest code to your local machine. Otherwise, the local machine code would not be changed without merge.

The command git fetch makes your local copy up to date by getting data from remote repository. The reason we need this is because somebody else might have made some changes to the code and you want to keep yourself updated.

Check the below image to know how **git fetch** and **git pull** working together:



**Stashing:**

Often, when you’ve been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the git stash command.

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time.

Think of the Stash as a clipboard on steroids: it takes all the changes in your working copy and saves them for you on a new clipboard. You're left with a clean working copy, i.e. you have no more local changes.

Later, at any time, you can restore the changes from that clipboard in your working copy - and continue working where you left off.

You can create as many Stashes as you want - you're not limited to storing only one set of changes. Also, a Stash is not bound to the branch where you created it: when you restore it, the changes will be applied to your current HEAD branch, whichever this may be.

**When to Stash:**

Stashing helps you get a clean working copy. While this can be helpful in many situations, it's strongly recommended...

* ...before checking out a different branch.
* ...before pulling remote changes.
* ...before merging or rebasing a branch.

In short, use **git stash** to save changes temporarily in local memory.

**Stashing your work:**

To demonstrate, you’ll go into your project and start working on a couple of files and possibly stage one of the changes. If you run git status, you can see your dirty state:

I modified the file Prog.java



mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ vi Prog.java

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git status

On branch master

Your branch is up to date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: Prog.java

no changes added to commit (use "git add" and/or "git commit -a")

Used **git stash save “some message”** command to save my changes temporarily in local memory i.e., on stack

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git stash save "content changed"

Saved working directory and index state On master: content changed

After stashing your changes, check the git status.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git status

On branch master

Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

You can easily get an overview of your current Stashes:

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git stash list

stash@{0}: On master: content changed

The newest Stash will always be at the top of the list, named "stash@{0}". Older Stashes have higher numbers.

When you're ready to restore a saved Stash, you have two options:

(a) Calling "git stash pop" will apply the newest Stash *and* clear it from your Stash clipboard.

(b) Calling "git stash apply <stashname>" will also apply the specified Stash, but it will *remain saved*. You can delete it later via "git stash drop <stashname>".

You can choose to *not* specify the Stash when using any of these commands. Then, Git will simply take the newest Stash (always "stash@{0}").

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git stash pop

On branch master

Your branch is up to date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: Prog.java

no changes added to commit (use "git add" and/or "git commit -a")

Dropped refs/stash@{0} (a8d1afae2ff12ec9d4cebe2bd5fa2babc9e8ece8)

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git status

On branch master

Your branch is up to date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: Prog.java

no changes added to commit (use "git add" and/or "git commit -a")

Now you can check the file below as our temporary changes are left same before and after stashing.



*In detail:*

**Stashing your work:**

The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy. For example:

**$ git status**

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

**$ git stash**

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

**$ git status**

On branch master

nothing to commit, working tree clean

At this point you're free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply your stash when you're ready.

Note that the stash is local to your Git repository; stashes are not transferred to the server when you push.

**Re-applying your stashed changes:**

You can reapply previously stashed changes with git stash pop:

**$ git status**

On branch master

nothing to commit, working tree clean

**$ git stash pop**

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)

Popping your stash removes the changes from your stash and reapplies them to your working copy.

Alternatively, you can reapply the changes to your working copy and keep them in your stash with git stash apply:

**$ git stash apply**

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

This is useful if you want to apply the same stashed changes to multiple branches.

Now that you know the basics of stashing, there is one caveat with git stash you need to be aware of: by default Git won't stash changes made to untracked or ignored files.

**Stashing untracked or ignored files:**

By default, running git stash will stash:

* Changes that have been added to your index (staged changes)
* Changes made to files that are currently tracked by Git (unstaged changes)

But it will not stash:

* New files in your working copy that have not yet been staged
* Files that have been ignored

So if we add a third file to our example above, but don't stage it (i.e. we don't run git add), git stash won't stash it.

**$ script.js**

**$ git status**

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Untracked files:

script.js

**$ git stash**

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

**$ git status**

On branch master

Untracked files:

script.js

Adding the -u option (or --include-untracked) tells git stash to also stash your untracked files:

**$ git status**

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Untracked files:

script.js

**$ git stash -u**

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

**$ git status**

On branch master

nothing to commit, working tree clean

You can include changes to ignored files as well by passing the -a option (or --all) when running git stash.



**Managing multiple stashes:**

You aren't limited to a single stash. You can run git stash several times to create multiple stashes, and then use git stash list to view them. By default, stashes are identified simply as a "WIP" – work in progress – on top of the branch and commit that you created the stash from. After a while it can be difficult to remember what each stash contains:

**$ git stash list**

stash@{0}: WIP on master: 5002d47 our new homepage

stash@{1}: WIP on master: 5002d47 our new homepage

stash@{2}: WIP on master: 5002d47 our new homepage

To provide a bit more context, it's good practice to annotate your stashes with a description, using git stash save "message":

**$ git stash save "add style to our site"**

Saved working directory and index state On master: add style to our site

HEAD is now at 5002d47 our new homepage

**$ git stash list**

stash@{0}: On master: add style to our site

stash@{1}: WIP on master: 5002d47 our new homepage

stash@{2}: WIP on master: 5002d47 our new homepage

By default, git stash pop will re-apply the most recently created stash: stash@{0}

You can choose which stash to re-apply by passing its identifier as the last argument, for example:

**$ git stash pop stash@{2}**

Stashing untracked files using different options:

$ **git stash save -u** (or)

$ **git stash save --include-untracked**

*In short:*

**git stash pop**- To apply the stash and then immediately drop it from your stack.

**git stash drop** - Top of the stash from the stack will be deleted.

**git stash drop <stash\_id>** - If you no longer need a particular stash, you can delete it with the command.

**git stash clear** - You can delete all of your stashes from the stack

**git stash pop** = = (**git stash apply** && **git stash drop**)

**Scenario** - *Stashing changes from one branch to other with ease or before checking out a different branch*

* Created a new file and working with that file “welcome” on master branch using git console.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ cat > welcome

welcome to the stash concept......!!!

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git add welcome

warning: LF will be replaced by CRLF in welcome.

The file will have its original line endings in your working directory.

* Now I want to move to a different branch NewB to fix a minor bug I've just found, so I need to stash my changes.
* Before checking out the master branch, I have used the command **git stash** to save file temporarily.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git stash save "stash demo"

Saved working directory and index state On master: stash demo

* You can easily get an overview of your current Stashes by using the following command:

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git stash list

stash@{0}: On master: stash demo

stash@{1}: On master: added a file - newfile

stash@{2}: WIP on master: a28e114 deltd files

stash@{3}: On master: added b.txt file

* You can save a stash on one branch, switch to another branch later, and try to reapply the changes using the **git stash apply** command. (reapply changes from one branch to other branch)
* If you want to apply one of the older stashes, you can specify it by naming it, like this:

**git stash apply stash@{2}.** If you don’t specify a stash, Git assumes the most recent stash and tries to apply it:

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git checkout NewB

Switched to branch 'NewB'

Your branch is ahead of 'origin/NewB' by 2 commits.

(use "git push" to publish your local commits)

mdodda@GUEST152 MINGW64 /e/MyGitProject (NewB)

$ cat welcome

cat: welcome: No such file or directory

Here I used git stash apply command to apply the stashes saved in master branch to the other branch NewB

mdodda@GUEST152 MINGW64 /e/MyGitProject (NewB)

$ git stash apply stash@{0}

On branch NewB

Your branch is ahead of 'origin/NewB' by 2 commits.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: welcome

mdodda@GUEST152 MINGW64 /e/MyGitProject (NewB)

$ cat welcome

welcome to the stash concept......!!!

mdodda@GUEST152 MINGW64 /e/MyGitProject (NewB)

$ git commit -m "welcome file added"

[NewB 93a61b7] welcome file added

1 file changed, 1 insertion(+)

create mode 100644 welcome

mdodda@GUEST152 MINGW64 /e/MyGitProject (NewB)

$ git checkout master

Switched to branch 'master'

Your branch is up to date with 'origin/master'.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ cat welcome

cat: welcome: No such file or directory

* Now check the file “welcome” is on the branch “NewB” where I have stashed the changes in one branch and applied it on other branch i.e., from master branch to NewB branch.

**Issue:**

"error: you need to resolve your current index first Prog.java: needs merge"

solution: git reset –merge

**What happens if you create a new branch?**

Well, doing so creates a new pointer for you to move around. Let’s say you create a new branch called *testing.* You do this with the git branch command:

**$ git branch testing**

This creates a new pointer at the same commit you’re currently on

**How does Git know what branch you’re currently on?**

**Git: What's HEAD**

It keeps a special pointer called HEAD. Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer to the local branch you’re currently on. In this case, you’re still on master. The git branch command only created a new branch - it didn’t switch to that branch.



HEAD file pointing to the branch you’re on.

**How to find what commit is current HEAD?**

Look in the file .git/HEAD.

You can do so by the command

# show the commit id the current HEAD points to

cd my\_project\_dir > moving to the current project directory

cat .git/HEAD

Sample output:

ref: refs/heads/master

The refs/heads/master is again a reference. It is a file, you can look at .git/refs/heads/master

The content on my disk is: 4929082f12ef06321143316fca5d04ce439b96bc

**What is detached HEAD?**

When HEAD is pointing to a commit ID directly, instead of a named branch, it's called detached HEAD.

# Git: Diff between {Working Dir, Staged Area, Last Commit}

There are 3 major concepts of places:

* Working Directory → files in your working directory.
* Staging Area (aka cache, index) → a temp area that git add is placed into.
* HEAD → A reference to a specific commit (think of it as a variable). Normally, it points to the last commit in local repository. (that is, after you did git commit).



All are local on your disk.

One important concept is Commit ID. Every commit has a ID. The commit id is a 40 digits hexadecimal, for example: 3b6ea398cc2d69212b04c29f06b8d15c0af34e34.

**Check the following commands that shows the differences between working directory, staging area and last commit:**

**How to get commit ID?**

# To show the last 3 commit's commit id

git log -3

**How to diff between {working dir, staging area}?**

# Diff working dir, staging area

git diff --color

# Diff working dir, staging area, 1 file

git diff --color filename

**How to diff between {staging area, last commit}?**

# Diff staging area, last commit. (--staged is same as --cached)

git diff --color --staged ‹commitID›

Eg: git diff --color --staged 7a020cb76d7ac5694e602acb9448ba2ba78d26db

**How to diff between {last commit, working dir}?**

First, use git log to get a commit ID.

# Diff last commit, working dir

git diff --color ‹commitID›

git diff --color 7a020cb76d7ac5694e602acb9448ba2ba78d26db

**Git Diff between 2 Commits:**

**How to diff between 2 commits in the same branch?**

#1 First run following command

git log file\_name

#2 To find the commits IDs. Then, do

git diff commit\_ID\_1 commit\_ID\_2 file\_name

Eg:

git diff 3819bfa40adf7b2cd7063a98f4f8b9bd1fb4f639 4929082f12ef06321143316fca5d04ce439b96bc test

(or)

git diff 3819bfa40adf7b2cd7063a98f4f8b9bd1fb4f639 4929082f12ef06321143316fca5d04ce439b96bc

You can also use the first few characters of commit id instead of whole id.

**git status:** Summary of {last commit, staging area, working dir}

**How to find what files are changed?**

# show changes between {staging area, last commit} and {staging area, working dir}

**git status .**



The **first section** “# Changes to be committed:” is the diff between {staging area, last commit}.

The **second section** “# Changes not staged for commit:” is the diff between {working dir, staging area}.

The **third section** “# Untracked files:” is also the diff between {working dir, staging area}.