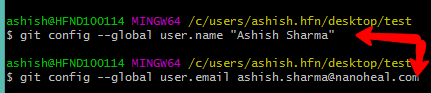
**Step 1- Configuring Git**

$ git config --global user.name "My Name"

$ git config --global user.email myEmail@example.com

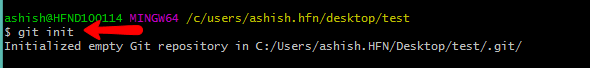


**Step 2- Initializing a new repository**

* Navigate to the project/repo directory in the git bash terminal.



* Execute the command – git init (This will enable Git for this particular folder and create a hidden .git directory where the repository history and configuration will be stored.)

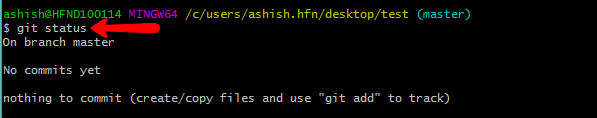


Now our git repository has been successfully created but is still empty.

**Step 3- Check status of your repository**

-returns information about the current state of the repository.

Execute this command – git status (to see current status of the project)

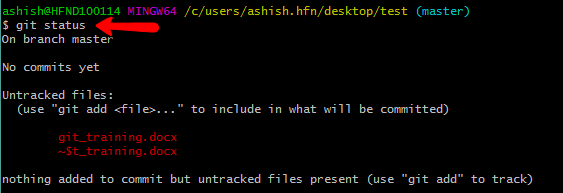


**Step 4- Adding some files into project**

* Add some txt files in the repository now.

This is done to have hands-on-experience.

Now again check status of the project:



Untracked means that the file is new and Git doesn’t know yet those if it should keep track of changes happening to the file or just ignore. To acknowledge the new file we need to stage it.

**Step 5- Staging the files**

This is done to add files to the staging area so Git can track changes made into the file.

Execute the command to stag the file: git add file\_name



\*\*Git\_training.docx is my filename.

In our case we have only one file so let's add that:

$ git add hello.txt

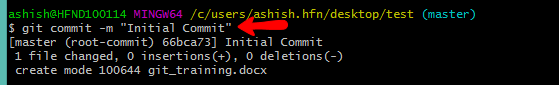
If we want to add everything in the directory, we can use:

$ git add –A

Again check status of the project and notice the difference.

**Step 6 -Committing the changes (**Saving the changes made in the project**)**

Execute the command: git commit –m “Commit message depicting some info about the commit”

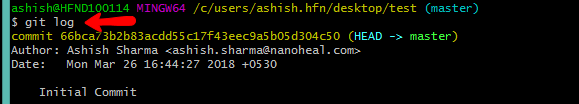


This will create a new commit with all the changes from the staging area.

**Step 7-Checking different commits**

Every commit has its unique id in the form of a string of numbers and symbols. To see a list of all commits and their ids we can use git log:

$ git log

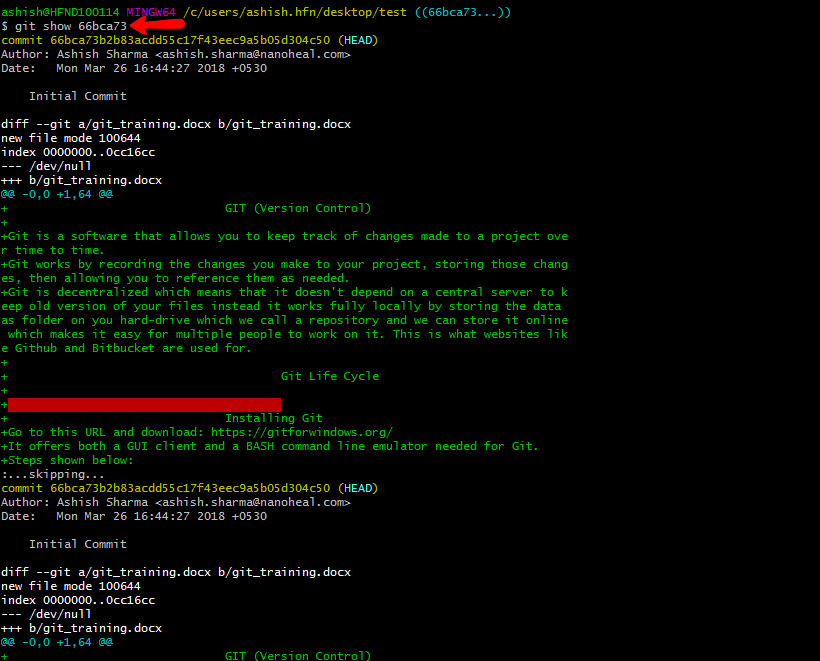


As you can see the ids are really long, but when working with them it's not necessary to copy the whole thing - the first several symbols are usually enough.

\*\*To see what was new in a commit we can run git show [commit]:

git show b10cc123 🡨 this is the first seven letters of the Sha-id.

A detailed log containing every information about that particular commit with the given sha-id will be shown.



\*\*In Git, the commit you are currently on is known as the HEAD commit. In many cases, the most recently made commit is the HEAD commit.

To see the HEAD commit, enter:

git show HEAD

The output of this command will display everything the git log command displays for the HEAD commit, plus all the file changes that were committed.

**Step 8- Checking the difference in commits**

\*\*In this case we want the diff of our most recent commit, which we can refer to using the HEAD pointer.

git diff HEAD

\*\*To see the difference between any two commits we can use git diff with the [commit-from]..[commit-to] syntax:

git diff 09bd8cc..ba25c0ff

\*\*run git diff with the --staged option to see the changes you just staged. You should see that what  was created.

git diff –staged

**Step 9- Connecting to a remote repository - git remote add**

In order to upload something to a remote repo, we first have to establish a connection with it. For the sake of this tutorial our repository's address will be *your Github repository address.*

To have repository address we have to create account on github or any vcs*.* To link our local repository with the one on GitHub, we execute the following line in the terminal:

\*This is only an example. Replace the URI with your own repository address.

$ git remote add origin <https://github.com/codehub7/test_git.git>

origin is the name of our remote repository



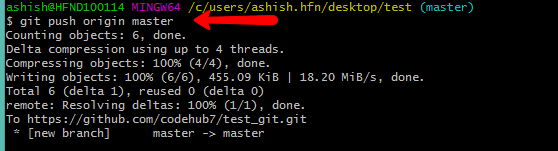
A project may have many remote repositories at the same time. To be able to tell them apart we give them different names. Traditionally the main remote repository in git is called *origin*.

**Step 10-Uploading to a server - git push**

Now we transfer our local commits to the server. This process is called a **push**, and is done every time we want to update the remote repository.

The Git command to do this is git push and takes two parameters - the name of the remote repo (we called ours *origin*) and the branch to push to (*master* is the default branch for every repo).

git push origin master

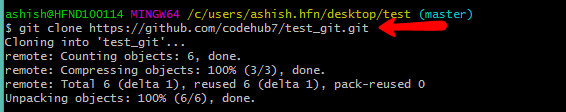


Depending on the service you're using, you will need to authenticate yourself for the push to go through. If everything was done correctly, when you go in your web browser to the remote repository created earlier, *files* should be available there.

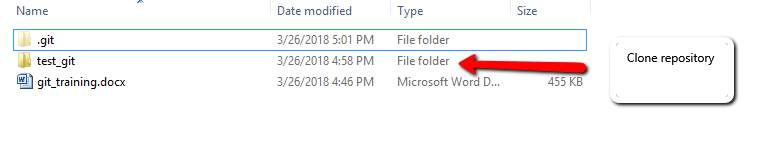
**Step 11-Cloning a repository - git clone**

At this point, people can see and browse through your remote repository on Github. They can download it locally and have a fully working copy of your project with the git clone command:

$ git clone <https://github.com/codehub7/test_git.git>



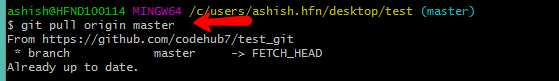
A new local repository is automatically created, with the github version configured as a remote.



**Step 12- Getting changes from a server - git pull**

If you make updates to your repository, people can download your changes with a single command - **pull**:

git pull origin master



Since nobody else has committed since we cloned, there weren't any changes to download.

**Step 13- Creating branches**

When we develop a new feature, it is beneficial to work on a copy of the original project, called a *branch*. Branches have their own history and isolate their changes from one another, until you decide to merge them back together. This is done for a couple of reasons:

* An already working, stable version of the code won't be broken.
* Many features can be safely developed at once by different people.
* Developers can work on their own branch, without the risk of their codebase changing due to someone else's work.

*1. Creating new branches - git branch*

The default branch of every repository is called ***master***. To create additional branches use the git branch <name>command:

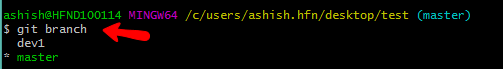
git branch dev1🡨 name of the branch



This just creates the new branch, which at this point is exactly the same as our *master*.

\*\*Now, when we run git branch, we will see there are two options available:

git branch



Master is the current branch and is marked with an asterisk. However, we want to work on our new amazing features, so we need to switch to the other branch. This is done with the git checkout command, expecting one parameter - the branch to switch to.

*2. Switching branches - git checkout*

git checkout dev1

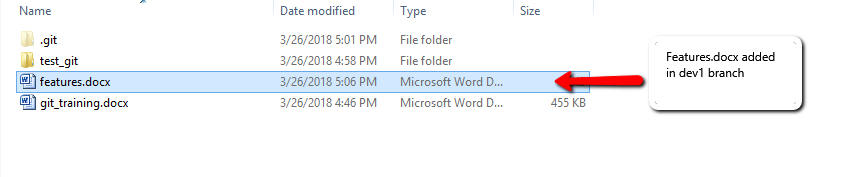


After checkout if you again run git branch command the branch switched will be marked with asterisk.

*3. Merging branches - git merge*

Before merging new branch with master branch our dev1 branch should also contain some file to merge so we will create some .txt or any type of files in it.

We will create it, add it, and commit it.

git add features.docx

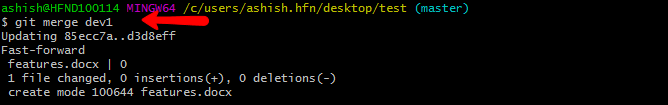
git commit -m "New feature complete."

The new feature is complete; we can go back to the master branch.

git checkout master

Now, if we open our project in the file browser, we'll notice that *feature.txt* has disappeared. That's because we are back in the master branch, and here *feature.txt* was never created. To bring it in, we need to git merge the two branches together, applying the changes done dev1 to the main version of the project.

git merge dev1



The master branch is now up to date. The dev1 branch is no longer needed and can be removed.

git branch -d dev1



This will delete the following named branch.

**Step 14- Unstage files**

\*\*You can unstage files by using the git reset command.

git reset filename

\*\*We can *unstage* that file from the staging area using

git reset HEAD filename

This command *resets* the file in the staging area to be the same as the HEAD commit. It does not discard file changes from the working directory; it just removes them from the staging area.

\*\*We can also do this with this command to reset to that particular commit:

git reset commit\_SHA 🡨sha-id of commit

**Step 15- Removing All the Things**

You can finally remove files you want to remove by using the git rm command which will not only remove the actual files from disk, but will also stage the removal of the files for us.

We can use a wildcard to get all the .txt files in one sweep:

git rm '\*.txt'

**Step 16- Reverting a file to a previous version**

Git allows us to return any selected file back to the way it was in a certain commit. This is done via the git checkout command, which we used earlier to switch branches, but can also be used to switch between commits.

In the following example we will take *hello.txt*and reverse everything we've done to it since the initial commit. To do so we have to supply the id of the commit we want to go back to, as well as the full path to our file.

git checkout 09bd8cc1 hello.txt

The command contains the sha-id of the commit to which we want to move and the file name.

\*\*The newest commit can be accessed by the HEAD alias.

git revert HEAD

\*\*For other commits it's best to use an id.

git revert b10cc123

When reverting older commits, keep in mind that merge conflicts are very likely to appear. This happens when a file has been altered by another more recent commit, and now Git cannot find the right lines to revert, since they aren't there anymore.

**Step 17-Resolving Merge Conflicts**

Conflicts regularly appear when merging branches or pulling someone else's work. Sometimes conflicts are handled automatically by git, but other times the person dealing with them has to decide.

This is an example where we're trying to merge two branches called john\_branch and tim\_branch. Both John and Tim are writing in the same file a function that displays all the elements in an array.

John is using a for loop:

// Use a for loop to console.log contents.

for(var i=0; i<arr.length; i++) {

console.log(arr[i]);

}

Tim prefers forEach:

// Use forEach to console.log contents.

arr.forEach(function(item) {

console.log(item);

});

They both commit their code on their respective branch. Now if they try to merge the two branches they will see the following error message:

$ git merge tim\_branch

Auto-merging print\_array.js

CONFLICT (content): Merge conflict in print\_array.js

Automatic merge failed; fix conflicts and then commit the result.

Git wasn't able to merge the branches automatically, so now it's up to the developers to manually resolve the conflict. If they open the file where the conflict resides, they'll see that Git has inserted a marker on the conflicting lines.

<<<<<<< HEAD

// Use a for loop to console.log contents.

for(var i=0; i<arr.length; i++) {

console.log(arr[i]);

}

=======

// Use forEach to console.log contents.

arr.forEach(function(item) {

console.log(item);

});

>>>>>>> Tim's commit.

Above the ===== we have the current HEAD commit, and below the conflicting one. This way we can clearly see the differences, and decide which the better version is, or write a new one altogether. In this situation we go for the latter and rewrite the whole thing, removing the markers to let Git know we're done.

// Not using for loop or for each.

// Use Array.toString() to console.log contents.

console.log(arr.toString());

When everything is set, a merge commit has to be done to finish the process.

git add –A🡨 this add all the files to the staging area

git commit -m "Array printing conflict resolved." 🡨committing with proper info

**Step 18- Setting up .gitignore**

In most projects there are files or entire folders that we don't want to ever commit. We can make sure that they aren't accidentally included in our git add -A by creating a *.gitignore* file:

1. Manually create a text file called .gitignore and save it in your project's directory.
2. Inside, list the names of files/directories to be ignored, each on a new line.
3. The .gitignore itself has to be added, committed and pushed, as it is just like any other file in the project.

Good examples for files to be ignored are:

* log files
* task runner builds
* the node\_modules folder in node.js projects
* folders created by IDEs like Netbeans and IntelliJ
* personal developer notes

A .gitignore banning all of the above will look like this:

\*.log

build/

node\_modules/

.idea/

my\_notes.txt

The slash at the end of some of the lines signals that this is a folder and we are ignoring everything inside it recursively. The asterisk serves its usual function as a wild card.