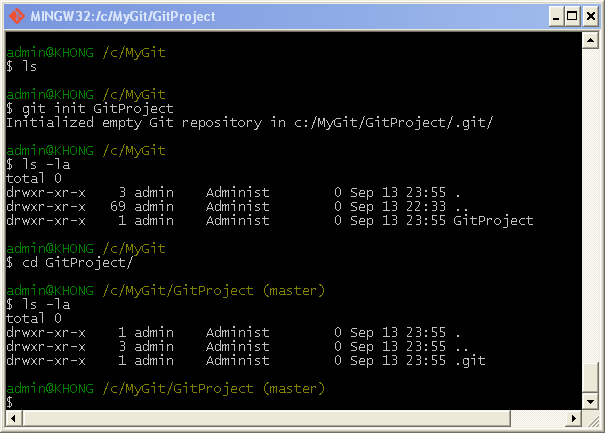
Git init

Let's start with Git Bash.

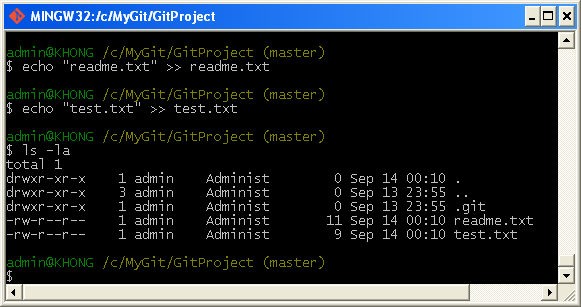
First, I created a new directory C:\MyGit. Then, issue a command "git init GitProjects", and it will generate a subdirectory named **GitProject**:



The **GitProject** contains all of our necessary repository files - a Git repository skeleton. At this point, nothing in our project is tracked yet.

Note that we could have issued the "git init" command within the directory, "GitProject", then it would have created **.git** subdirectory.

Now, we may want to create some files:



Using Git

Let's go into the GitProject directory where we have two files: readme.txt and test.txt.

Status

To check the status of git, we issue "git status".

The **status** is the main tool we use to determine which files are in which state is the git status command. If we run this command directly after a clone, we should see something like this:

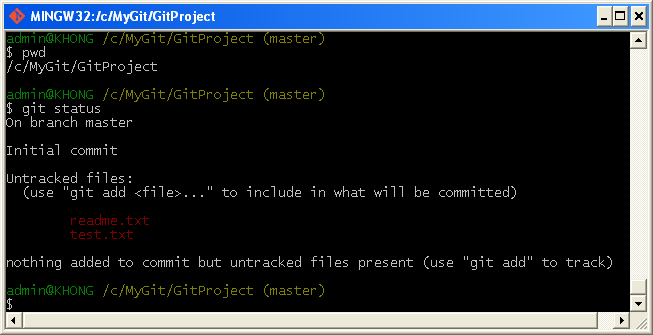
**$ git status**

**# On branch master**

**nothing to commit (working directory clean)**

This means we have a clean working directory - in other words, no tracked files are modified. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tells us which branch we're on.

In our case, we get:

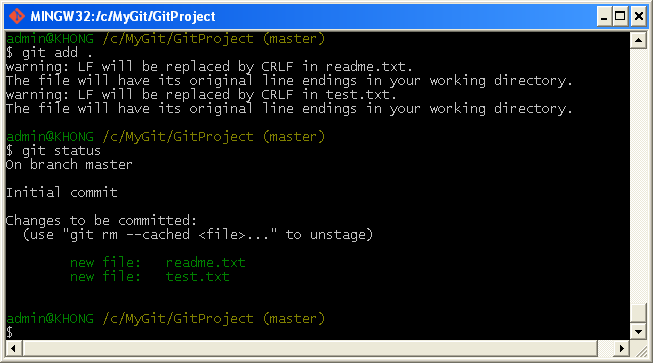


We can see we are on **master** branch and we have **untracked** two files (in red color). Untracked basically means that Git sees a file we didn't have in the previous snapshot (commit); Git won't include it in our commit snapshots until we explicitly tell it to do so. It does this so that we don't accidentally begin including generated binary files or other files that we did not mean to include. Let's start tracking the files.

add

If we want to version-control existing files (as opposed to an empty directory), we should probably begin tracking those files and do an initial commit. We can accomplish that with a few **git add** commands that specify the files we want to track, followed by a **commit**:

With "**git add .**", we can add all files in the folder and make them be tracked.

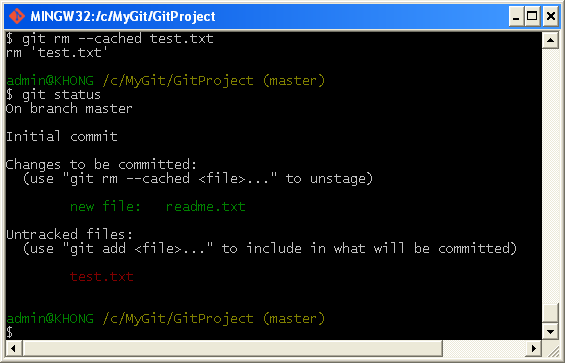


Now the files are displayed in green and git tells us both of them are new and they are now **tracked and staged**. Note that the two files are before the commit and the snapshot hasn't occurred yet.

We can tell that they staged because they're under the "Changes to be committed" heading. If we commit at this point, the version of the files at the time we ran 'git add' are what will be in the historical snapshot.

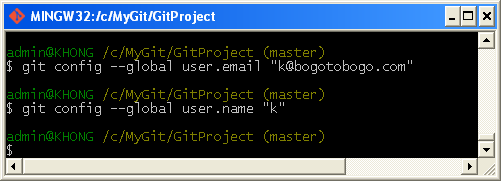
We may recall that when we ran 'git init' earlier, we then ran git add (files) - that was to begin tracking files in our directory. The git add command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

If we want to remove "test.txt", we do **git rm --cached test.txt**:



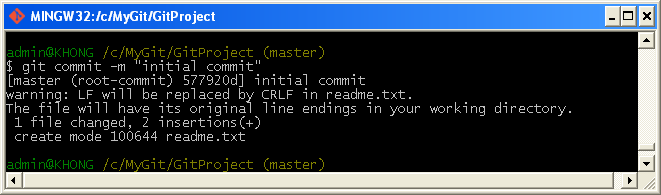
config email and name

Before we commit, we need to configure email and name:

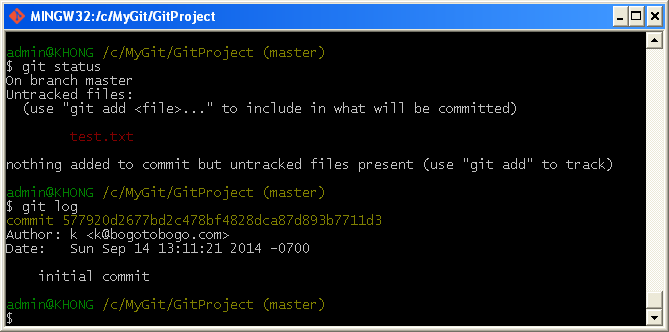


Commit

We now want to commit:

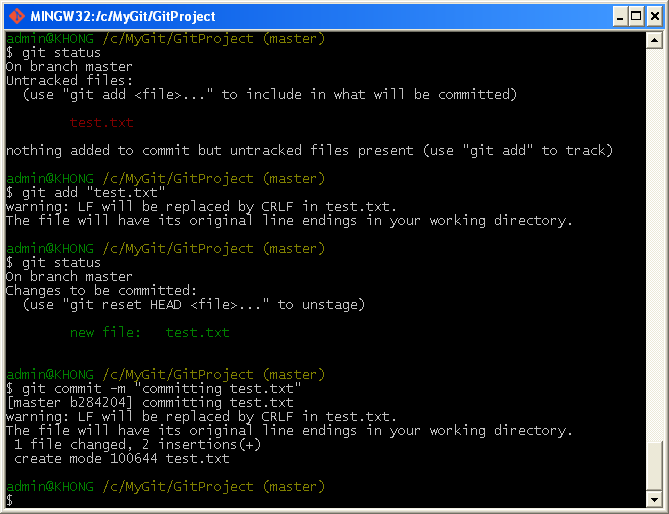


We can check what's been done so far by issuing **log**:

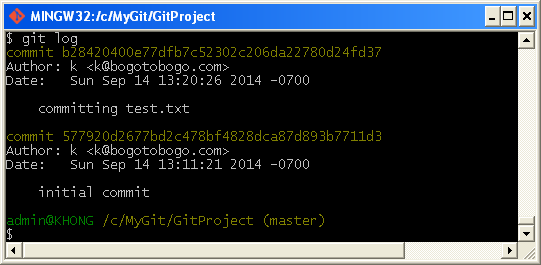


Git shows the name, email, and the message for the operation.

The other file, "test.txt" is not yet being tracked, so we want add that in.

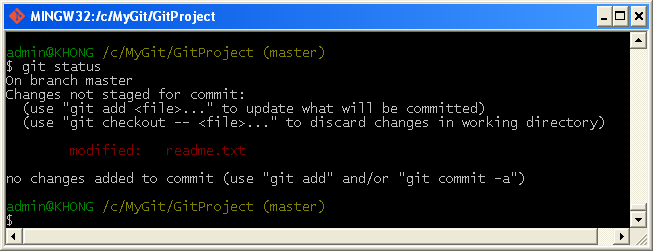


If we do "git log" again, it will show two commits we've made so far:



Modify

Now, let's modify "readme.txt", and run **status**:

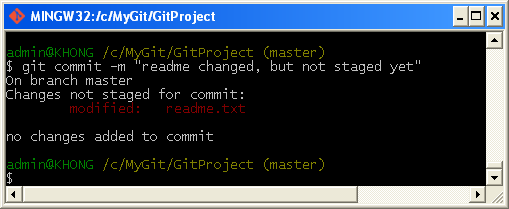


In this case, we changed a file that was already tracked.

If we change a previously tracked file and then run our status command again, we get something that looks like the picture above:

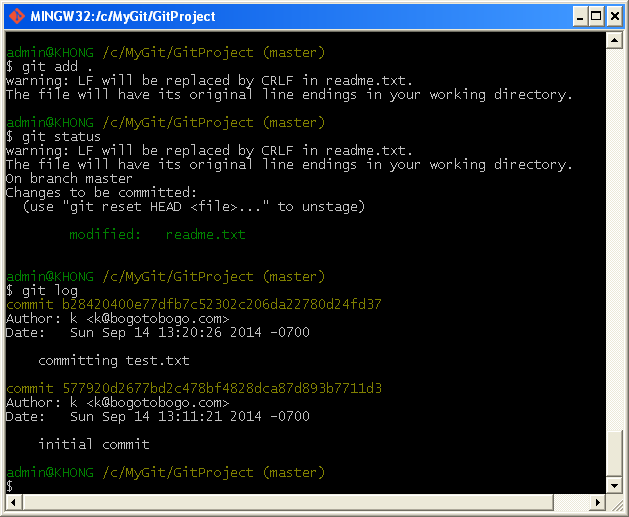
The "readme.txt" file appears under a section named "Changes not staged for commit" - which means that a file that is tracked has been modified in the working directory but not yet staged.

At this point, if we run **commit**, it will fail:



It failed because it's not at staging area. So, whenever we do commit, we should put a file at the staging area bef the commit.

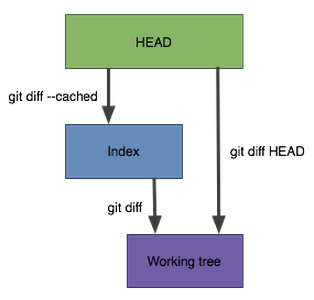
In other words, to stage it, we run the **git add** command (it's a multipurpose command - we use it to begin tracking new files, to stage files, and to do other things like marking merge - conflicted files as resolved).



By running **add**, we did put the file into the staging area:

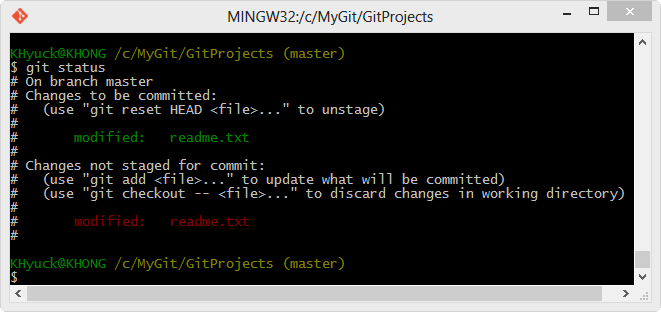
Now, we can commit the changes made to the "readme.txt" file:

Git diff

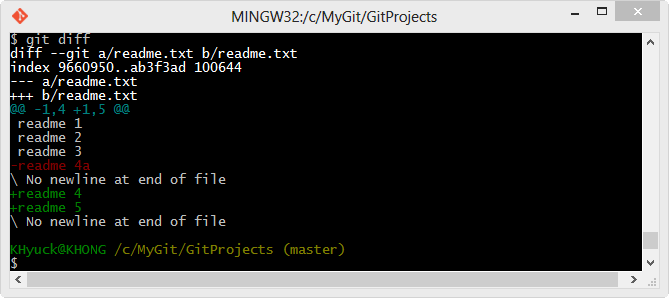


Usually, since the **git status** command is too vague and does not give enough information about the change, when we want to know exactly what we changed not just which files were changed, we can use the **git diff** command. We'll probably use it most often to answer these two questions: What have we changed but not yet staged? And what have we staged that we are about to commit? Although **git status** answers those questions very generally, **git diff**shows us the exact lines added and removed.

Let's say we edit and stage the readme file again and then edit the readme file without staging it. If we run status command, we see something like this:

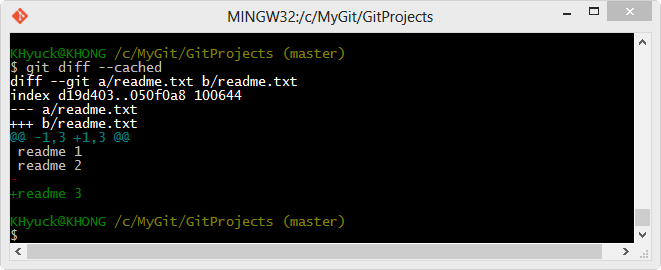


If we want to see what we've changed but not yet staged, type **git diff** with no other arguments:

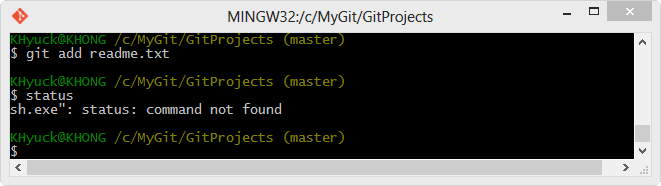


**git diff** command compares what is in our working directory with what is in our staging area. The result tells us the changes we've made that we haven't yet staged.

If we want to see what we've staged that will go into our next commit, we can use **git diff --cached**. (In Git versions 1.6.1 and later, we can also use **git diff --staged**, which may be easier to remember.) This command compares our staged changes to our last commit:



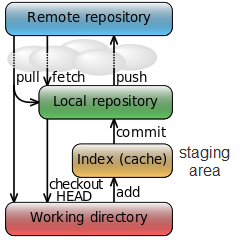
It's important to note that git diff by itself doesn't show all changes made since our last commit - only changes that are still unstaged. This can be confusing, because if we've staged all of our changes, git diff will give us no output as shown in the picture below:



Git commit

Though we learned how we commit in the previous chapter, in this chapter, we'll review it again.

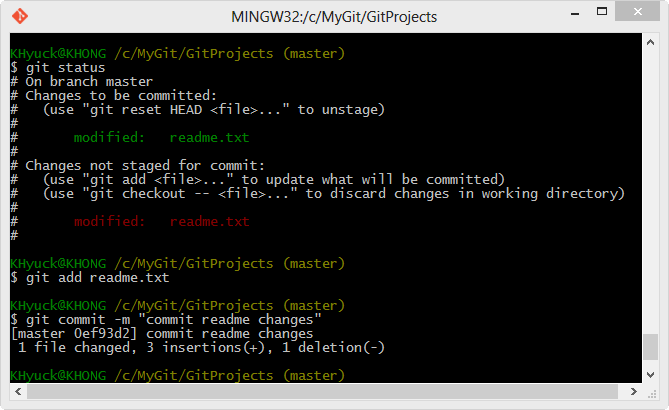
Now that our staging area is set up the way we want it, we can commit our changes. Remember that anything that is still **unstaged** - any files we have created or modified but we haven't run **git add** after we edited them - won't go into this commit. They will stay as modified files on our disk.



When we run **git status**, if we see that everything has been staged, then we're ready to commit our changes. The simplest way to commit is to type **git commit**:

**$ git commit**

The commit gives us some output about itself: which branch we committed to (master), what SHA-1 checksum the commit has (463dc4f), how many files were changed, and statistics about lines added and removed in the commit.



Remember that the **commit** records the **snapshot** we set up in our **staging area**. Anything we didn't stage is still sitting there modified; we can do another commit to add it to our history. Every time we perform a **commit**, we're recording a snapshot of our project that we can revert to or compare to later.

Deleting files

Deleting files from our repo is simple, delete them, and then commit.

We have 4 files in repository: Book1, Book2, Book3, and OldBook.

**k@laptop:~/GitDemo$ ls**

**Book1.rtf Book2.rtf Book3.rtf OldBook.rtf**

**k@laptop:~/GitDemo$ git status**

**On branch master**

**nothing to commit, working directory clean**

**k@laptop:~/GitDemo$**

Now, we want to remove "OldBook" which is already in our repository.

**k@laptop:~/GitDemo$ git rm OldBook.rtf**

**k@laptop:~/GitDemo$ git status**

**On branch master**

**Changes not staged for commit:**

**(use "git add/rm ..." to update what will be committed)**

**(use "git checkout -- ..." to distestbranchd changes in working directory)**

**deleted: OldBook.rtf**

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

But our repository still has the 'OldBook'. So, we need to do commit the deleted file:

**k@laptop:~/GitDemo$ git commit -am "deleted OldBook"**

**[master 7a4caa8] deleted OldBook**

**1 file changed, 9 deletions(-)**

**delete mode 100644 OldBook.rtf**

Now, it's gone and the 'OldBook' became a history. So, deleting a file from a repo is straight forward: delete and commit.

Renaming files

We have two ways of renaming a file in our repository.

1. Renaming a file in a working folder.

Now we want to rename the 'Book3' to 'Appendix'.

First, we directly rename the file in our working directory:

**k@laptop:~/GitDemo$ mv Book3.rtf Appendix.rtf**

**k@laptop:~/GitDemo$ ls**

**Appendix.rtf Book1.rtf Book2.rtf**

If we check the git status:

**k@laptop:~/GitDemo$ git status**

**On branch master**

**Changes not staged for commit:**

**(use "git add/rm ..." to update what will be committed)**

**(use "git checkout -- ..." to distestbranchd changes in working directory)**

**deleted: Book3.rtf**

**Untracked files:**

**(use "git add ..." to include in what will be committed)**

**Appendix.rtf**

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

The git thinks the 'Book3' is removed and 'Appendix' is added. Now we should remove 'Book3' and add 'Appendix' on repo:

**k@laptop:~/GitDemo$ git rm Book3.rtf**

**rm 'Book3.rtf'**

**k@laptop:~/GitDemo$ git add Appendix.rtf**

Then, check the status again:

**k@laptop:~/GitDemo$ git status**

**On branch master**

**Changes to be committed:**

**(use "git reset HEAD ..." to unstage)**

**renamed: Book3.rtf -> Appendix.rtf**

Notice that Git is smart enough to realize what we did was just renaming the file!

All we have to do now is to commit:

**k@laptop:~/GitDemo$ git commit -m "renamed Book3 as Appendix"**

**[master 8d7d7f1] renamed Book3 as Appendix**

**1 file changed, 0 insertions(+), 0 deletions(-)**

**rename Book3.rtf => Appendix.rtf (100%)**

1. Using "git mv" command

This time we want to rename 'Book2' to "Introduction".

**k@laptop:~/GitDemo$ git mv Book2.rtf Introduction.rtf**

Now git immediately recognizes we renamed it:

**k@laptop:~/GitDemo$ git status**

**On branch master**

**Changes to be committed:**

**(use "git reset HEAD ..." to unstage)**

**renamed: Book2.rtf -> Introduction.rtf**

All we have to do now is doing commit:

**k@laptop:~/GitDemo$ git commit -m "Book2->Introduction"**

**[master e1ec728] Book2->Introduction**

**1 file changed, 0 insertions(+), 0 deletions(-)**

**rename Book2.rtf => Introduction.rtf (100%)**

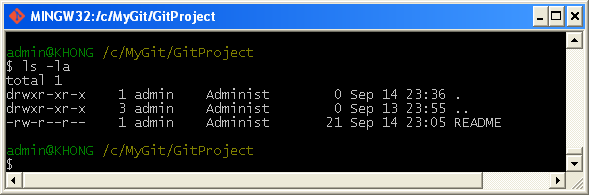
**k@laptop:~/GitDemo$ git status**

**On branch master**

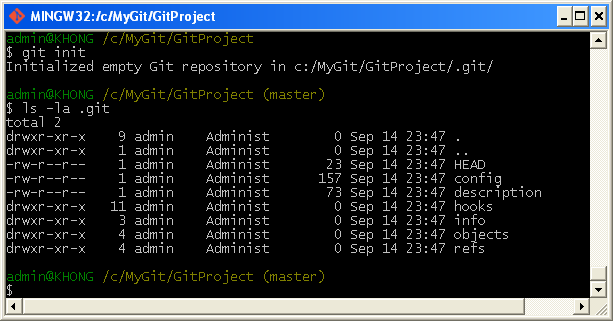
**nothing to commit, working directory clean**

git init & git config

We have "README" in our GitProject directory:

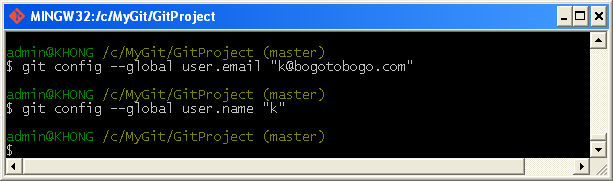


Then we need to issue **git init** command:



As we can see the **git init** command creates an empty Git repository - basically a **.git** directory with subdirectories for **objects**, **refs/heads**, **refs/tags**, and **template** files. An initial HEAD file that references the HEAD of the master branch is also created.

We may also want to do email and name configuration using **git config** command:



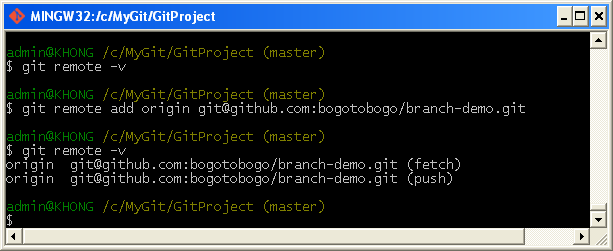
Add & Commit

Now we want to put our "README" to staging area and then do commit (take snapshot).

# git_add_commit.png

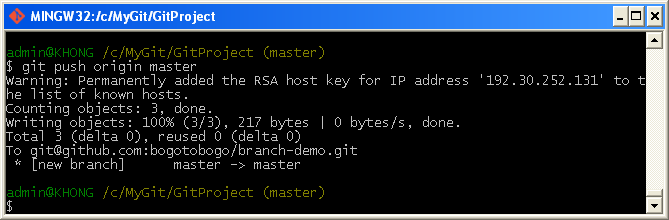
adding remote repo

Up to this point, we've been working on local repo, but now it's time to use remote repo which is GitHub. To use remote repo, we need to tell git what remote repo we're going to use via **git remote add** command:

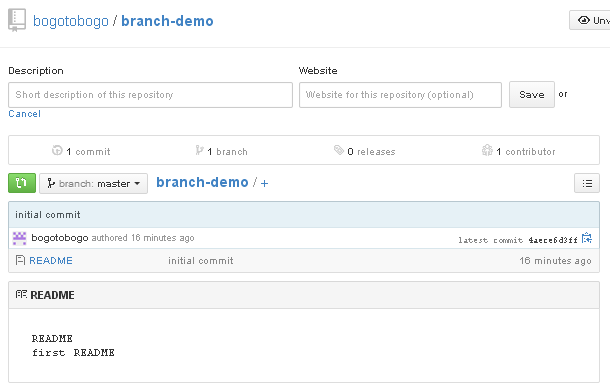


git push

Since we can use remote repo name as **origin**, let upload our README to our GitHub:



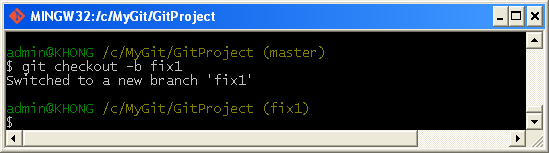
We can check whether the README file is really in GitHub:



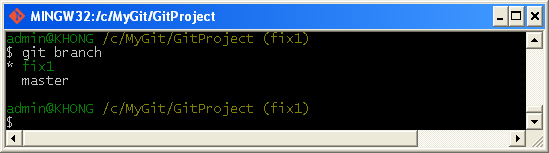
branching & merging

git checkout

**git checkout -b <new\_branch>** will create a new branch (the "-b" causes a new branch to be created) with the name of "new\_branch":



We can check which branch we're now by using **git branch**:



As expected, we're on the new branch, 'fix1'. Now, we want to modify the README like this:

**README**

**first README fix1**

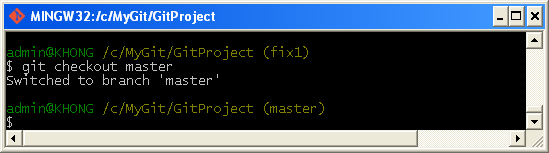
To add the change and commit:

**$ git commit -a -m "Added branch fix1"**

**1 file changed, 1 insertion(+), 1 deletion(-)**

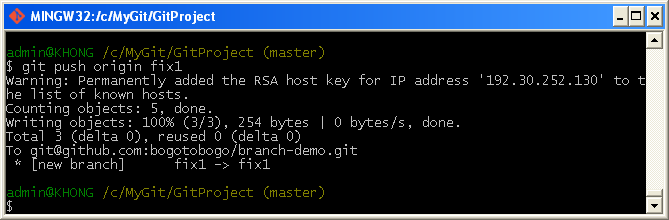
Note that we committed the change only to the branch not to the master.

Now we can go back (switch) to the master:

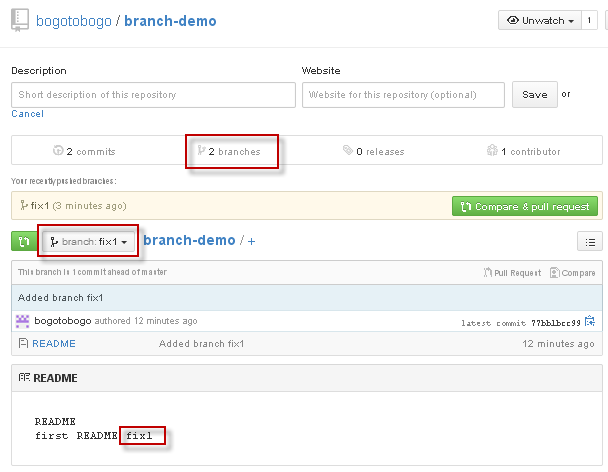


branch git push

We can push our branch to GitHub:



If we check our GitHub, it now has two branches and the newly added fix1 branch shows our change:



listing branches

To see which branch we're now on:

**admin@JRAYA /c/MyGit/GitProject (master)**

**$ git branch**

**fix1**

**\* master**

**admin@JRAYA /c/MyGit/GitProject (master)**

To see the merged branch:

**admin@JRAYA /c/MyGit/GitProject (master)**

**$ git branch --merged**

**\* master**

**merged**

**admin@JRAYA /c/MyGit/GitProject (master)**

To list unmerged branch:

**admin@JRAYA /c/MyGit/GitProject (master)**

**$ git branch --no-merged**

**fix1**

**admin@JRAYA /c/MyGit/GitProject (master)**

To see all the branches and the last commit:

**admin@JRAYA /c/MyGit/GitProject (master)**

**$ git branch -v**

**fix1 77bb1bc Added branch fix1**

**\* master 4aece6d initial commit**

**merged 4aece6d initial commit**

**admin@JRAYA /c/MyGit/GitProject (master)**

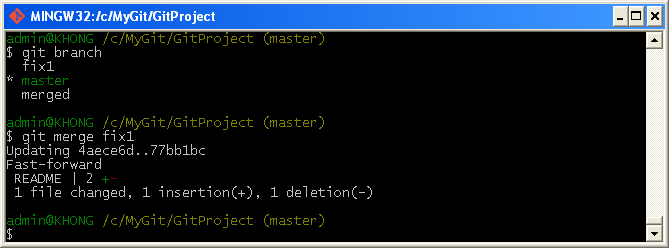
merge

We're now on \*master, and check the README:

**README**

**first README**

To merge with fix1:



Now the README file has been merged on master:

**README**

**first README fix1**

deleting branch

Let's check our branches again:

**admin@JRAYA /c/MyGit/GitProject (master)**

**$ git branch**

**fix1**

**\* master**

**merged**

The fix1 branch is there but we do not need it any more since it's been merged. So, let's remove it:

**admin@JRAYA /c/MyGit/GitProject (master)**

**$ git branch -d fix1**

**Deleted branch fix1 (was 77bb1bc).**

**admin@JRAYA /c/MyGit/GitProject (master)**

**$ git branch**

**\* master**

**merged**

**admin@JRAYA /c/MyGit/GitProject (master)**

If we want to delete unmerged branch, we can use "**git branch -D name\_of\_unmerged\_branch**".

deleting github branch

How we can delete a branch on our GitHub?

**admin@JRAYA /c/MyGit/GitProject (master)**

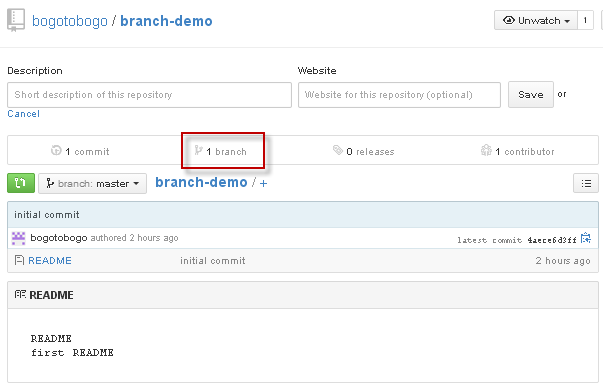
**$ git push origin :fix1**

**To git@github.com:test/branch-demo.git**

**- [deleted] fix1**

**admin@JRAYA /c/MyGit/GitProject (master)**

Let's check how our GitHub looks like:



Our GitHub now has only one branch which is master since another branch fix1 has been deleted.

Now, we need to push our merge local repo to remote repo:

**admin@JRAYA /c/MyGit/GitProject (master)**

**$ git push origin master**

**Counting objects: 5, done.**

**Writing objects: 100% (3/3), 254 bytes | 0 bytes/s, done.**

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

**To git@github.com:test/branch-demo.git**

**4aece6d..77bb1bc master -> master**

**admin@JRAYA /c/MyGit/GitProject (master)**



## **GIT AND GITHUBMERGE CONFLICTS WITH SIMPLE EXAMPLE**

Creating conflicts

In the previous chapter ([Fast-forward merge](http://www.bogotobogo.com/cplusplus/Git/Git_GitHub_Fast-Forward_Merge.php)), the merge was very easy because we did not have any conflicts. In other words, in that case, the master branch (trunk) has no updates since we branched 'testbranch' off the 'master'. All we had to do was simply appending our new updates to the master trunk.

In this chapter, we'll create conflicts by updating our 'master' branch at the same time we update the 'testbranch' branch. So, when we try to merge the 'testbranch' branch, we'll have two different versions of 'Book1', and git has no way to figure out which one to take in.

We are currently on the 'master' branch and we have another branch called 'testbranch':

**k@laptop:~/GitDemo$ ls**

**Appendix Book1 Introduction**

**k@laptop:~/GitDemo$ git branch**

**testbranch**

**\* master**

The log:

**k@laptop:~/GitDemo$ git log --oneline**

**d567a72 Merge branch 'testbranch'**

**4552553 Added 2nd commit to the testbranch branch**

**...**

**b440952 initial commit**

Check if it's clean:

**k@laptop:~/GitDemo$ git status**

**On branch master**

**nothing to commit, working directory clean**

Let's modify 'Book1' on the 'master' by adding the following as its first line:

**Year 2014 - updated master branch**

Of course, we need to commit it:

**k@laptop:~/GitDemo$ git add .**

**k@laptop:~/GitDemo$ git commit -m "updated 1st line 2014 on master branch"**

**[master 79b0888] updated 1st line 2014 on master branch**

**1 file changed, 1 insertion(+)**

Updated log:

**k@laptop:~/GitDemo$ git log --oneline**

**79b0888 updated 1st line 2014 on master branch**

**d567a72 Merge branch 'testbranch'**

**4552553 Added 2nd commit to the testbranch branch**

**a7f55e3 added a line at the beginning**

**...**

**k@laptop:~/GitDemo$ git status**

**On branch master**

**nothing to commit, working directory clean**

**k@laptop:~/GitDemo$ git branch**

**testbranch**

**\* master**

Then, we switch to the 'testbranch' branch.

**k@laptop:~/GitDemo$ git checkout testbranch**

**Switched to branch 'testbranch'**

**k@laptop:~/GitDemo$ git branch**

**\* testbranch**

**master**

**k@laptop:~/GitDemo$ git status**

**On branch testbranch**

**nothing to commit, working directory clean**

Then, add the similar line as we've done on the 'master' as its first line:

**Year 2014 - updated testbranch branch**

Commit:

**k@laptop:~/GitDemo$ git add .**

**k@laptop:~/GitDemo$ git commit -m "updated 1st line 2015 on testbranch branch"**

**[testbranch a428001] updated 1st line 2015 on testbranch branch**

**1 file changed, 1 insertion(+)**

Log for the 'testbranch' branch:

**k@laptop:~/GitDemo$ git log --oneline**

**a428001 updated 1st line 2015 on testbranch branch**

**4552553 Added 2nd commit to the testbranch branch**

**a7f55e3 added a line at the beginning**

**...**

Fixing Merge conflicts

Now we have two different versions of 'Book1' that were committed to each branch, 'master' and 'testbranch'.

We're now on 'master' branch:

**k@laptop:~/GitDemo$ git branch**

**testbranch**

**\* master**

Let's try to merge them;

**k@laptop:~/GitDemo$ git merge testbranch**

**Auto-merging Book1**

**CONFLICT (content): Merge conflict in Book1**

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

**k@laptop:~/GitDemo$**

Basically, git is saying, "I don't know which one is the most up-to-date version. It's your job to fix it".

Note that at this merging phase, we're not on any specific branch but we're somewhere "floating land" which is not 'master' nor 'testbranch'.

But when we open the 'Book1' in our working directory, it shows something like this:

**k@laptop:~/GitDemo$ git merge testbranch**

**Auto-merging Book1**

**CONFLICT (content): Merge conflict in Book1**

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

**k@laptop:~/GitDemo$**

Clearly, it's not the one from 'master' nor from 'testbranch'.

**<<<<<<< HEAD**

**Year 2014 - updated master branch**

**=======**

**Year 2015 - updated testbranch branch**

**>>>>>>> testbranch**

It shows the difference in the two versions of 'Book1'. We're lucky because our sample is simple enough to fix manually, however, in real cases, it could become really messy, and that's the reason why we should minimize the conflicts by commit as often as possible.

Ok, let's combine the conflicts like this:

**Year 2015 - updated by master & testbranch at the same time**

Save it and then commit:

**k@laptop:~/GitDemo$ git status**

**On branch master**

**You have unmerged paths.**

**(fix conflicts and run "git commit")**

**Unmerged paths:**

**(use "git add ..." to mark resolution)**

**both modified: Book1**

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

**k@laptop:~/GitDemo$ git add .**

**k@laptop:~/GitDemo$ git commit -m "fixed the conflict, set year = 2015"**

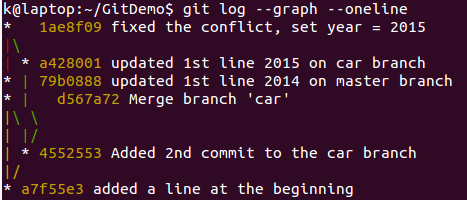
**[master 1ae8f09] fixed the conflict, set year = 2015**

**k@laptop:~/GitDemo$ git status**

**On branch master**

**nothing to commit, working directory clean**

Now we resolved the conflicts. Let's look at the graph log:



Here is the reading from the graph log above:

1. "d567a72 Merge branch 'testbranch'" -   
   Since we branched off the 'testbranch', we did fast-forward merge in the previous chapter ([Fast-forward merge](http://www.bogotobogo.com/cplusplus/Git/Git_GitHub_Fast-Forward_Merge.php))
2. "a428001 updated 1st line 2015 on testbranch branch", "79b0888 updated 1st line 2014 on master branch" -   
   Each branch has its own update.
3. " 1ae8f09 fixed the conflict, set year = 2015" -   
   Then, both branches merged.

## **QUICK COMMAND REFERENCE**

**remote** means remote repository uri,   
for example, https://github.com/User/myrepo.git

Clone & branches

1. To clone a remote git repository with depth = 1 and specifying my project name:
2. **$ git clone --depth=1 remote myproject**
3. To clone a remote git repository with specific branch (eg. master):
4. **$ git clone -b master remote**
5. To see the branches after cloned all branches (cloned w/o specifying any):
6. **$ git branch -a**
7. **\* master**
8. **remotes/origin/HEAD -> origin/master**
9. **remotes/origin/dev**
10. **remotes/origin/master**
11. Switching branches (**checkout -b**):
12. **$ git checkout -b dev**

**Switched to a new branch 'dev'**

About remote repository

1. To setup local repo with a remote repositories.  
   First, we need to use **git init** and then use **git remote add remote-name remote-url**:
2. **$ git init**

This will create **.git** directory into our current working directory. Now we want to add the remote repo:

**$ git remote add origin https://github.com/User/repo.git**

Here the 4th one (**origin**), we can name it whatever we want to. But **origin** is a kind of convention.

1. To see what are the remote repositories:
2. **$ git remote -v**
3. **origin https://github.com/User/repo.git (fetch)**
4. **origin https://github.com/User/repo.git (push)**

If we want to use the same name, we can drop our project name which is the last argument.

1. To switch remote repositories from repo1 to repo2:  
   This is my current remote:
2. **$ git remote -v**
3. **origin https://github.com/User/repo1.git (fetch)**
4. **origin https://github.com/User/repo1.git (push)**

Here it is:

**$ git remote set-url origin https://github.com/repo2.git**

Note that we overwrote (replaced with a new remote) an existing one.   
If we wanted to keep the old one, we could have used **add** instead of **set-url**:

**$ git remote add origin2 https://github.com/repo2.git**

Then, we have two remote repos:

**$ git remote -v**

**origin https://github.com/User/repo1.git (fetch)**

**origin https://github.com/User/repo1.git (push)**

**origin2 https://github.com/User/repo2.git (fetch)**

**origin2 https://github.com/User/repo2.git (push)**

Staging

There are couple of options when we do stage files.

1. Staging only "modified (including deleted)":
2. **$ git add -u**

or

**$ git add --update**

If we want to do commit at one shot:

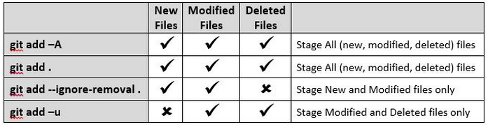
**$ git commit -a**

1. Staging all (modified + new + deleted):  
   "git add -A" is equivalent to "git add --all"
2. **$ git add -A**

or

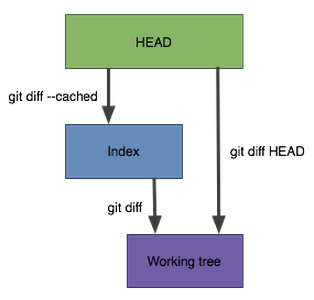
**$ git add --all**

As we can see from the table below, as far as Git 2, there is not difference "git add -A" and "git add ."



Source : [Difference between "git add -A" and "git add ."](http://stackoverflow.com/questions/572549/difference-between-git-add-a-and-git-add/16162511#16162511)

Diff



Just for demonstration purpose, I put my file (**a.txt**) in several stage with a text of that stage name:

1. A file on working directory (working tree) - "My Working directory file"
2. A file in staging area - "My Staged file"
3. A file already committed - "My HEAD file"
4. A file already pushed to remote - "My Remote file"

So, when we issue a diff command, we can see the diff more clearly.

1. After we staged a file, and want to see the diff of the file from the one that's already in the HEAD (local repo):
2. **$ git diff --staged a.txt**
3. **-My HEAD file**
4. **+My Staged file**
5. Worked on a file that's on our working directory (or working tree), and want to see the diff from the one already staged:
6. **$ git diff a.txt**
7. **-My Staged file**
8. **+My Working directory file**
9. To see the diff of my file on working tree with the one already in the HEAD
10. **$ git diff HEAD a.txt**
11. **-My HEAD file**
12. **+My Working directory file**
13. To see the diff of my file on working tree with the remote one.  
    To be more clear, we may want to know how the remote looks like:
14. **$ git remote -v**
15. **origin https://github.com/User/repo.git (fetch)**
16. **origin https://github.com/User/repo.git (push)**

Now, let's do diff:

**$ git diff origin/master:a.txt a.txt**

**-My remote file**

**+My Working directory file**

We can switch the positions

**$ git diff HEAD:a.txt origin/master:a.txt**

**-My HEAD file**

**+My remote file**