Lab 2 – Basic Git Commands

Contents

[1 Introduction 1](#_Toc103163533)

[2 Lab 2](#_Toc103163534)

# Introduction

In this lab you will learn how to create a Git repository and track, stage, and commit files. Note:

* The labs will ask you to add specific files with specific text and the screen prints in this lab will reflect that. It is not imperative that you type exactly the text I specify. Also, you may want to try commands again and/or experiment and that is fine also.
* Labs 2-5 build on one another. When you finish a lab, I recommend copying the folder/repository and give it a new name. For example, in this tutorial, you will create and use a folder/repository named, *gitex.* So, at the conclusion, I recommend you make a copy and name it *getex\_2.* Then you will start tutorial 3 with *gitex* and if things go wrong, you can delete it, and copy *gitex\_2,* give it the new name, *gitex,* and start tutorial 3 over.

Read this:

|  |
| --- |
| Copied from: <https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>  There are three main sections of a Git project: the Git directory, the working tree, and the staging area. The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.  The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.  Working tree, staging area, and Git directory.The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.  The basic Git workflow goes something like this:   1. You modify files in your working tree. 2. You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area. 3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.   If a particular version of a file is in the Git directory, it’s considered committed. If it has been modified and was added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified. In Git Basics, you’ll learn more about these states and how you can either take advantage of them or skip the staged part entirely. |

# Lab

1. Run Cmder
2. Create a folder. I will use *E:\gitex.*

**λ**  e:

**λ**  mkdir gitex

Type

**λ**  dir

And you should see *gitex* listed. On linux, you can use *ls* or *ls –l* to achieve the same.

1. Navigate into the *gitex* folder

**λ**  cd gitex

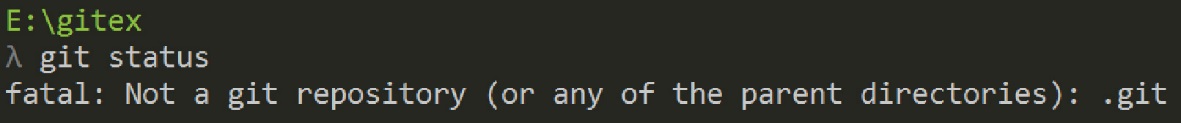
1. Create a text file, *foo.txt* in the *gitex* folder and type “Hello World” in the file and save. We will use Cmder to launch Notepad to create and edit the file. **Note: if you are not on Windows, or don’t have notepad, simply create the file in the gitex folder some other way (Word, *etc*).**

**λ**  notepad foo.txt

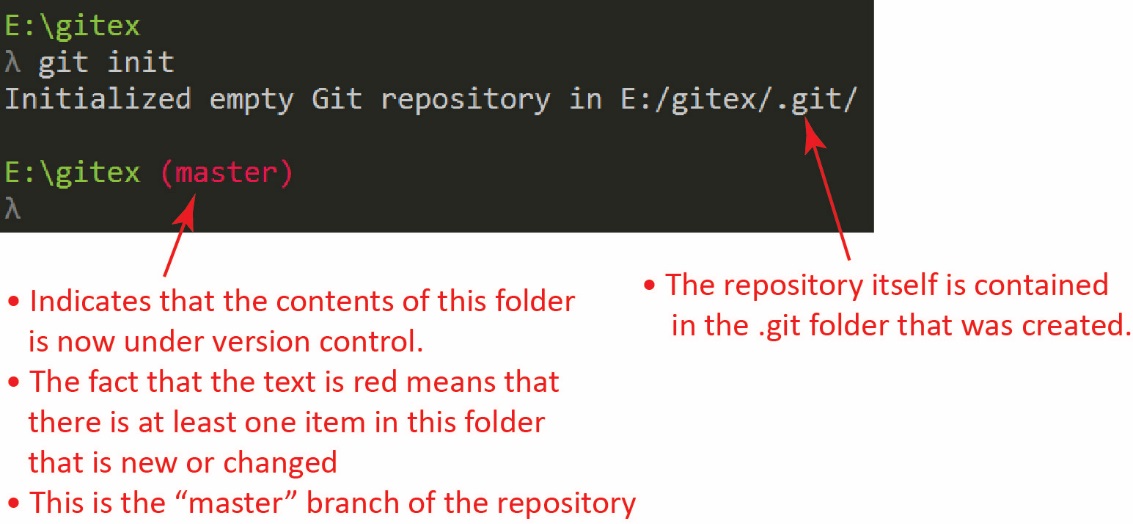
1. Issue the status command:

**λ**  git status

The result is shown in the figure below. Note that the result indicates that this is not a Git repository. We are about to make it a local Git repository.



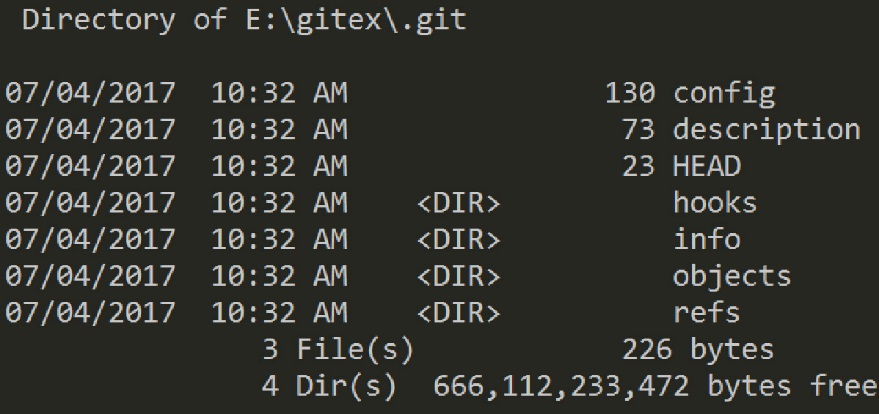
1. (Read, no action required). Getting help:
2. **λ**  git help <verb> //- displays man page in browser from local copy
3. <https://git-scm.com/docs> - Git reference manual (all commands)
4. Google and/or Stackoverflow
5. Initialize a local *Git* repository. Type:

**λ**  git init

The result is shown on the right. Note the indicated items.

**Note: As of Summer 21, it is gold instead of red**

1. (Read, no action required) You can create a Git repository in two ways:
2. As we did above, turn a local directory into a Git repository
3. Clone an existing repository from (usually) a remote location which results in local Git repository. We consider this in Lab 9.
4. Navigate into the repository and examine its contents. Type:

**λ**  cd .git

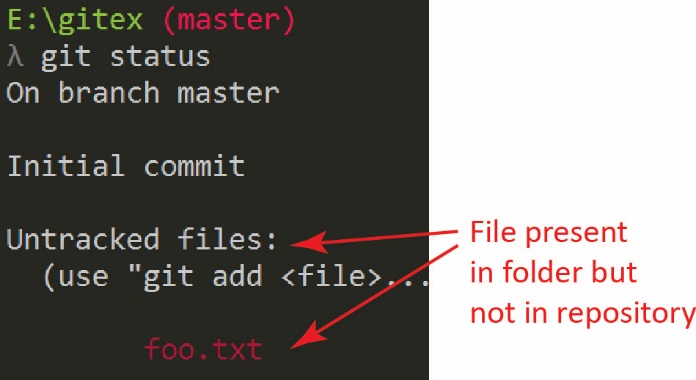
**λ**  dir

The result is shown in the figure on the right. We would generally never have any reason to directly use any of these files/folders.

1. Return to the *gitex* folder. Type:

**λ**  cd ..

1. Check the status of the repository. Type:



**λ**  git status

The result is shown on the right. The file(s) shown in red are being recognized as present in the *gitex* folder, but not actually in the Git repository. Adding files to a repository is a two-step process. First they are *staged* and then they are *committed*, which we will do next.

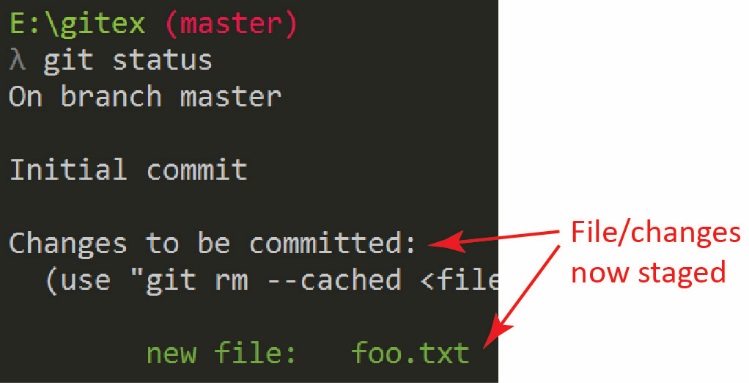
1. *Stage* the file. Type

**λ**  git add foo.txt

No response is shown unless there is an error. Note that we can use the “-A” flag to stage all files: git add –A.

This also works (with a “.” after the add): git add .

|  |
| --- |
| Note: if you goof something up, you can start over by removing the gitex folder:  **λ**  cd ..  **λ**  rmdir /s gitex  But, be careful and don’t delete all the stuff on your harddrive. This is not reversible. Just make sure you are deleting the gitex folder only. |

1. Check the status of the repository. Type:

**λ**  git status

The result is shown on the right. The file(s) shown in green are now staged, but not actually committed to the repository.

1. (Read, no action required). The *commit* command has this syntax:

**λ**  git commit -m "Commit message title" -m "Description of changes" [file1 file2 …]

The first argument is the commitmessage title briefly stating what was changed. The second argument provides more detail on what was changed. Sometimes in these tutorials we will just supply the title and not a detailed description of changes. Any time we make a change to files in a repository, or add new ones, we supply a commit message. Finally, you can optionally supply a list of file names. If no files are listed, all files that have been staged are committed.

1. *Commit* the file to the repository. The *commit* command commits all files/changes that have been staged. Type

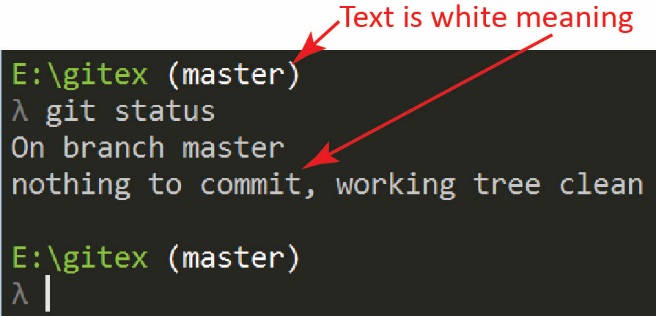
**λ**  git commit -m "new file added" -m "this file is used to let the user know about all the new hello world options that are available"

(Read, no action required) The commit command commits all staged files. Right now we just had one. However, immediately after you commit, you might realize you forgot a file. You can amend a commit this way:

**λ**  git commit -m "Initial commit"

**λ**  git add forgotten\_file

**λ**  git commit --amend

1. Check the status of the repository. Type:

**λ**  git status

The result is shown on the right.

1. List all c*ommits* that have been made to the repository. In this case, there is only one. Type:

**λ**  git log

The result is shown below:



1. (Read, no action required). A file can be in one of these four states:

|  |  |
| --- | --- |
| File State | Description |
| Untracked | File in working folder but not a part of the Git repository. For a file to move from *untracked* to *staged* you typically use: **λ**  git add *filename*. |
| Modified (Tracked) | A file in the repository has been changed. For a file to move from *modified* to *staged* you typically use: **λ**  git add *filename* |
| Staged | A changed file has been marked to be included in next commit. For a file to move from *staged* to *committed* you typically use:  **λ**  git commit –m “Commit message…" |
| Committed | In repository with no changes. |

The figure below shows these states. Note, that “Unmodified” in the figure is “Committed” in the table above.



1. (Optional Reading). When you commit, Git stores a snapshot of all your files which you can think of as a version (see image below). If a file has not changed it does not save a copy with the new version; it simply points back to the last saved version. From [Getting-Started-Git-Basics](https://git-scm.com/book/en/v2/Getting-Started-Git-Basics):

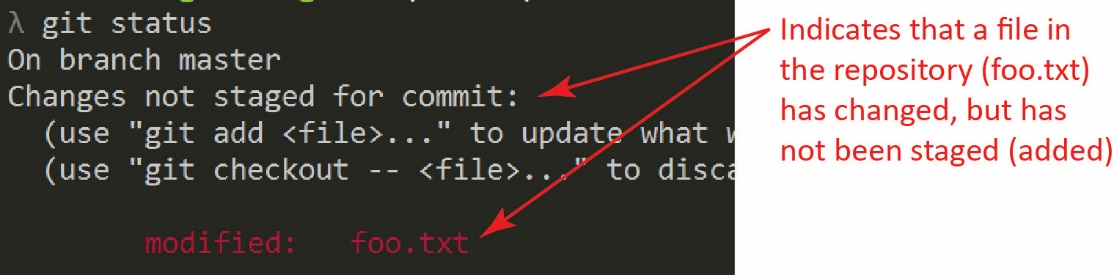
Git doesn’t think of or store its data this way. Instead, Git thinks of its data more like a series of snapshots of a miniature filesystem. With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots.



1. Add this text: “Glad you could join us.” to *foo.txt*.

**λ**  notepad foo.txt

1. Check the status of the repository. Type:

**λ**  git status

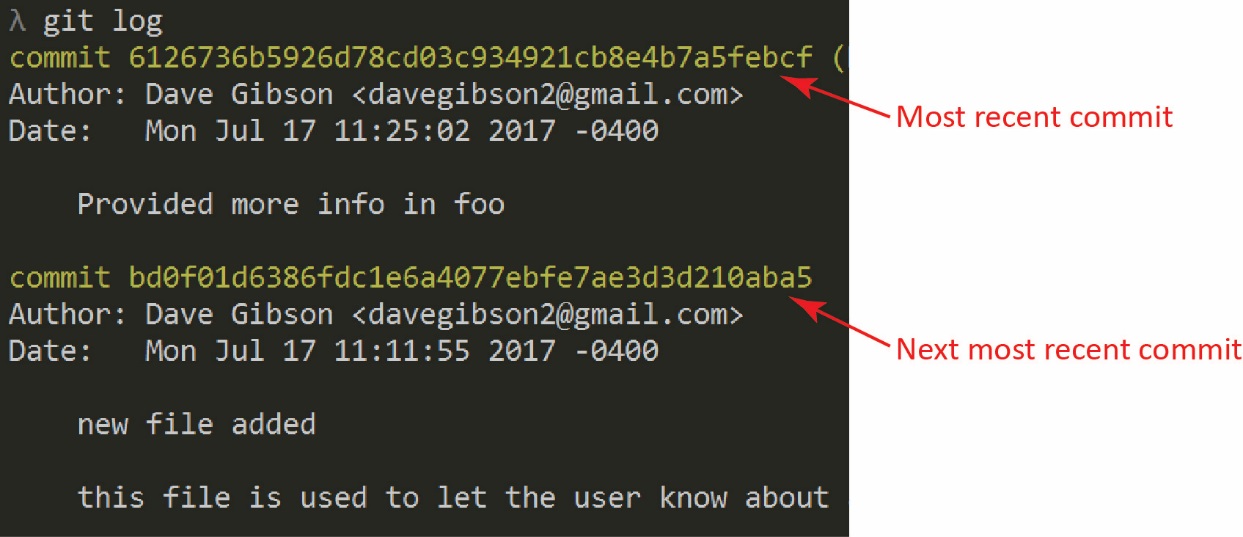
The result is shown on the right. Note that Git recognizes that the file has been changed and that it is not staged. The file is in the *modified* state.

1. Stage and commit these changes by issuing these commands. This time we will just supply the commit title. Also notice that we are using the “.”on the *add* which specifies to add all changed files to the index. Type:

**λ**  git add .

**λ**  git commit –m “Provided more info in foo"

1. Display all commits, most recent first by issuing this command:

**λ**  git log

The result is shown on the right. Notice that all commits are shown with the most recent shown first.

|  |
| --- |
| Note: if the output from a git command will not fit in the window, you will see a “:” in the lower left portion of the window. Press space to see the next page, or “q” to return to the prompt. |

1. When you delete a file from your workspace it is not automatically deleted from the repository. To delete it from the repository, you must stage and commit it. Do the following:
2. Create a new file, add some text, and save.

**λ**  notepad fee.txt

1. State and commit

**λ**  git add fee.txt

**λ**  git commit –m “Added to make more robust”

1. Delete file from workspace

**λ**  git rm fee.txt

1. Check the status. Note that *fee.txt* is in the staged state.

**λ**  git status

1. Commit the deletion.

**λ**  git commit –m “Decided it didn’t add much”

**λ**  git status

(Optional – Read, no action required) Note, if you use a DOS rm (not git rm), the file will not be staged. You have to do that yourself and then commit. For reference:

**λ**  rm fee.txt

**λ**  git add fee.txt

**λ**  git commit –m “Decided it didn’t add much”

1. You can remove a file from the staging area with: git reset filename. This does not discard changes to the file, it simply changes its state from staged to modified. Do the following.
2. Add some text to the end of *foo.txt*, stage, and check the status:

**λ**  notepad foo.txt

**λ**  git add foo.txt

**λ**  git status

1. Remove *foo* from the staging area and check the status. Notice that the file is now in the modified state. Finally, verify that the change you made is still present:

**λ**  git reset foo.txt

**λ**  git status

**λ**  type foo.txt

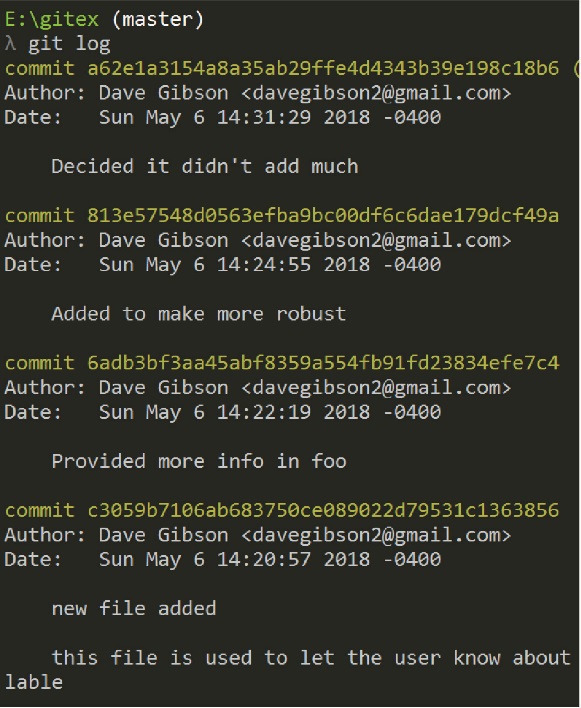
|  |
| --- |
| Note: if the *type* command is not recognized (probably only on a mac or linux), the *cat* command will probably work (it works on windows also). Regardless, the point is simply to display the contents of *foo.txt* and you can do that on your computer in any number of ways. |

1. You can discard the changes by checking out the file from the repository and check the status. Notice that the working tree is clean, there is nothing to commit. Finally, verify that the changes are discarded and the original is restored.

**λ**  git checkout foo.txt

**λ**  git status

**λ**  type foo.txt

1. **Do the following:**
2. Display the log of commits. The result will be similar to the image on the right.

**λ**  git log

1. Make a screen shot similar to the one on the right. Your name and date should be displayed.
2. Place the image in the *HW VCS* document in the appropriate place.
3. The image should easily readable without zooming in or out.

Good idea: make a backup copy of your *gitex* folder with the name: *gitex\_2*. Then, you’ll start Lab 3 using the *gitex* repo. If something goes wrong in lab 3, you can delete the *gitex* folder, and then copy *gitex\_2* and give it the new name, *gitex*.