Git – A Quick Guide

Table of Contents

The following is a straightforward guide to Git. It covers basic concepts, must-know commands, and examples to get the user started with using Git and GitHub quickly.

What is Git?

Git is a version control system that allows tracking project history and collaboration. It records the changes made to the code in a database called 'repository'. Therefore, when we look at the project history, we can see who has made what changes, when, and why. In case of any error, we can easily revert our project back to an earlier state.

Git is the most popular version control system in the world because it is:

- Free
- Open source
- Fast
- Scalable
- Branching and Merging are computationally much less expensive than in other version control systems

How to Use Git?

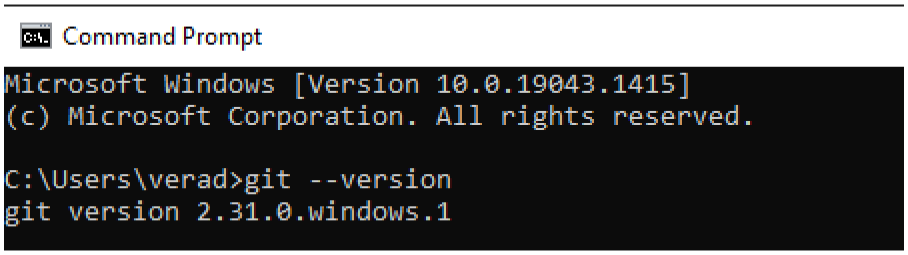
- Command line: fastest and easiest
- IDEs and code editors: In VS code for example, the basic Git features exist
- Graphical user interfaces: These tools are available on the Git website. The 2 most popular tools are GitKraken (cross-platform and compatible with GitKraken Boards for issue tracking and GitKraken Timelines for project management, free for open-source projects, requires an annual fee for industrial ones) and Sourcetree (completely free, only for Windows and Mac).

Note that all of these GUI tools have some limitations. Additionally, a command line is always available while a GUI tool might not always be at our disposal (example: If we connect to a server remotely and we don't have the permission to install a GUI tool).

Installation

In order to check if Git is already installed on a machine, open the terminal window and type:

```
git --version
```



```
Microsoft Windows [Version 10.0.19043.1415]
(c) Microsoft Corporation. All rights reserved.

C:\Users\verad>git --version
git version 2.31.0.windows.1
```

Using the latest version of Git is highly recommended.

Head to *git-scm.com/downloads* and follow the instructions to download Git on your machine.

On Windows, once Git is installed, we get the Git BASH application (BASH is for Born Again Shell). This is a command prompt window that emulates Unix or Linux.

Git Environment Customization

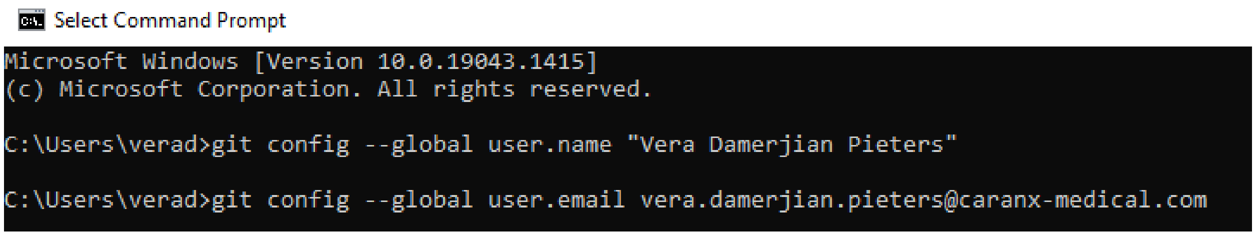
The first time we use Git, we need to specify a few configuration settings:

- Name
- E-mail
- Default editor
- How to handle line endings

These settings can be specified at 3 different levels:

- System: Settings apply to all users of the current computer
- Global: Settings apply to all repositories of the current user
- Local: Settings apply to the current repository (repository in the current folder)

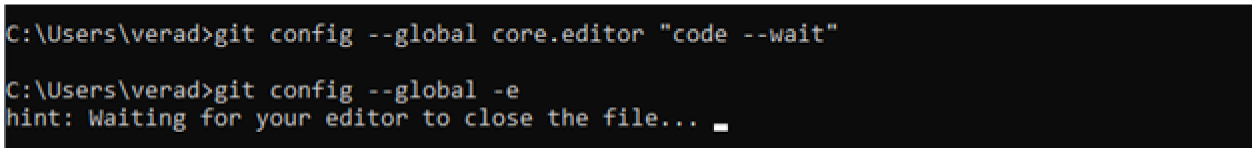
If Visual Studio Code is added to the path, then it can be launched by typing `code` in the command window.



```
Microsoft Windows [Version 10.0.19043.1415]
(c) Microsoft Corporation. All rights reserved.

C:\Users\verad>git config --global user.name "Vera Damerjian Pieters"

C:\Users\verad>git config --global user.email vera.damerjian.pieters@caranx-medical.com
```

We use the `–wait` flag to tell the terminal window to wait until we close the new VS Code instance.

Using the `–e` flag opens the default editor to edit all the global settings. It also waits for the user to close VS Code.



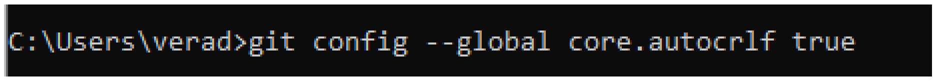
```
C:\Users\verad>git config --global core.editor "code --wait"

C:\Users\verad>git config --global -e
hint: Waiting for your editor to close the file...
```

On Windows, end-of-lines are indicated with 2 special characters: `\r` (carriage return) and `\n` (line feed).

On macOS and Linux systems, end-of-lines are applied using only `\n`.

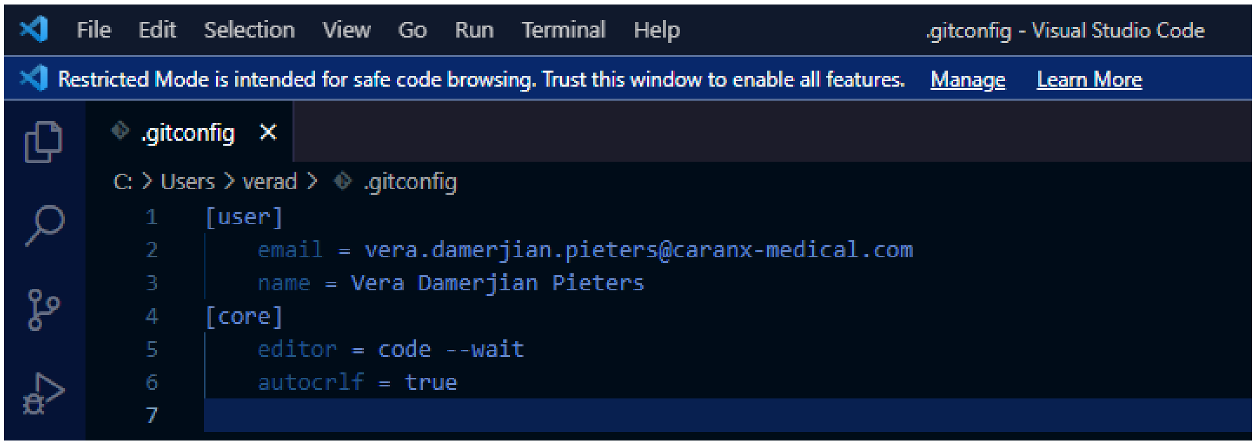
Therefore, managing end-of-lines is essential to prevent issues. This can be configured using `core.autocrlf` (crlf is short for carriage return line feed).

When a Windows user checks in his/her code on Git repository, Git should remove the `\r` character from end-of-lines. Also, when he/she checks out code from the repository, Git should add the `\r` character to the end of lines. This is achieved by setting this property to `true`.



```
C:\Users\verad>git config --global core.autocrlf true
```

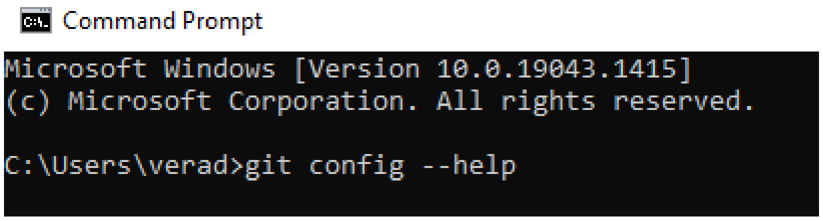
All these configuration settings are stored in a text file which we can edit using our default editor (VS Code in our case).


File
Edit
Selection
View
Go
Run
Terminal
Help
.gitconfig - Visual Studio Code
Restricted Mode is intended for safe code browsing. Trust this window to enable all features.
Manage
Learn More
.gitconfig
C: > Users > verad > .gitconfig

```
[user]
    email = vera.damerjian.pieters@caranx-medical.com
    name = Vera Damerjian Pieters
[core]
    editor = code --wait
    autocrlf = true
```

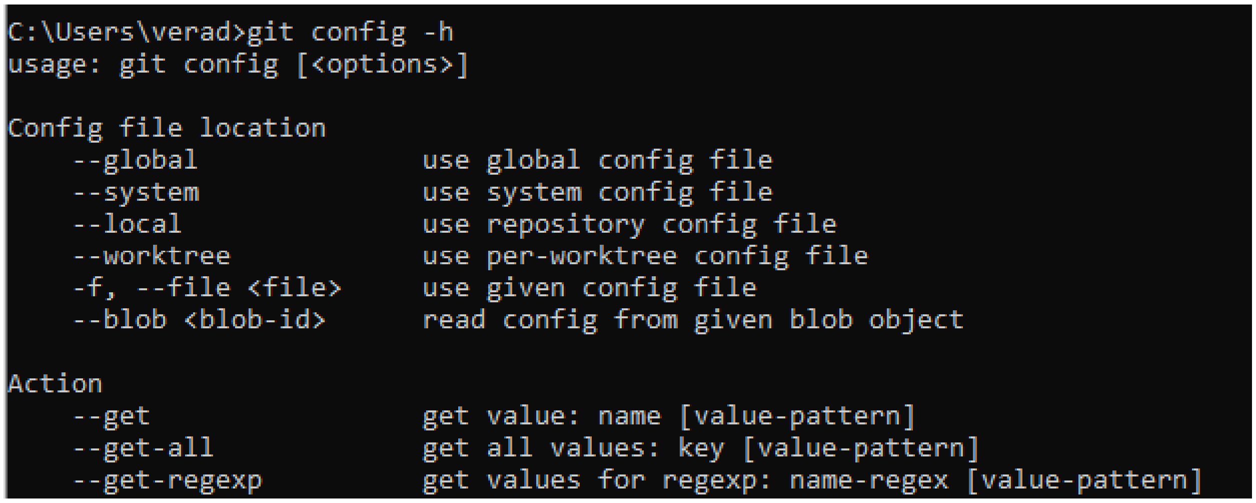
Get Help with Git

If you need help with Git config for example, type the following, and a documentation page will open:



```
Microsoft Windows [Version 10.0.19043.1415]
(c) Microsoft Corporation. All rights reserved.

C:\Users\verad>git config --help
```

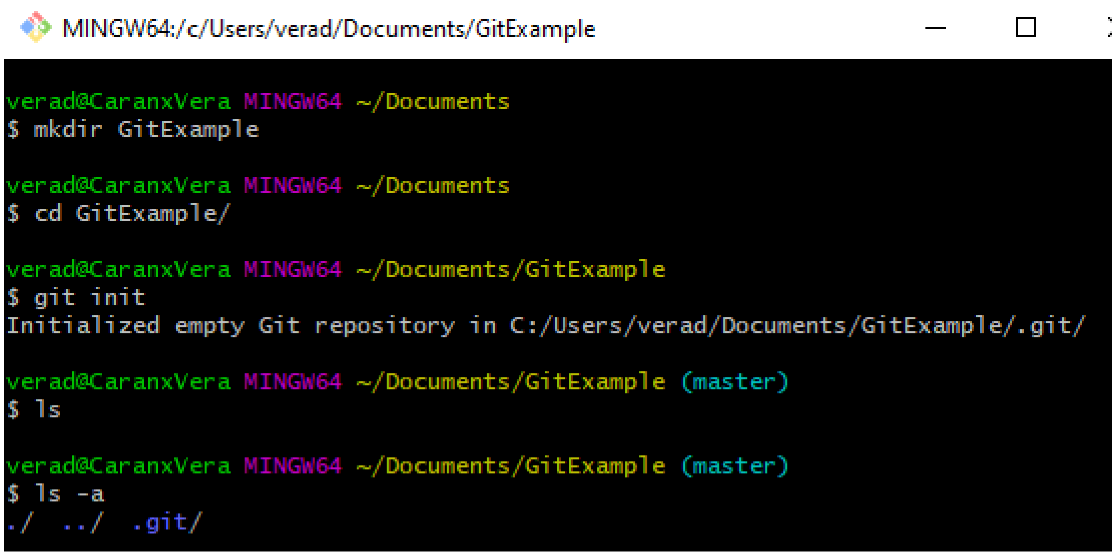
For a small help summary instead of a full page, type:



```
C:\Users\verad>git config -h
usage: git config [<options>]

Config file location
    --global              use global config file
    --system              use system config file
    --local               use repository config file
    --worktree            use per-worktree config file
    -f, --file <file>     use given config file
    --blob <blob-id>      read config from given blob object

Action
    --get                 get value: name [value-pattern]
    --get-all             get all values: key [value-pattern]
    --get-regexp          get values for regexp: name-regex [value-pattern]
```

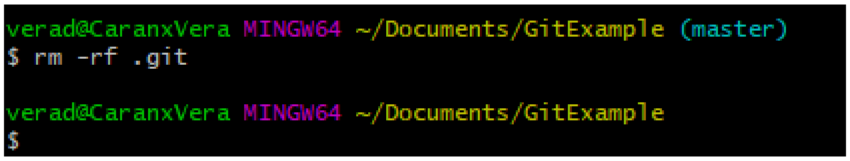
Create a Repository

Once we create our project folder in our computer, we type `git init` to initialize a new and empty directory.
Notice the .git subdirectory created inside the project directory (hidden by default because we are not supposed to touch it). Therefore, if we type `ls`, nothing shows. However, if we type `ls -a` (for all), the hidden .git will be displayed.



```
MINGW64:/c/Users/verad/Documents/GitExample

verad@CaranxVera MINGW64 ~/Documents
$ mkdir GitExample

verad@CaranxVera MINGW64 ~/Documents
$ cd GitExample/

verad@CaranxVera MINGW64 ~/Documents/GitExample
$ git init
Initialized empty Git repository in C:/Users/verad/Documents/GitExample/.git/

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ ls

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ ls -a
./  ../  .git/
```

When we see the word `master` beside a directory path, we know that this is a Git directory. If the .git subdirectory is removed, the project history is lost:



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ rm -rf .git

verad@CaranxVera MINGW64 ~/Documents/GitExample
$
```

`rm` is the remove command. `-rf` is a combination of `-r` (recursive removal: remove directories and subdirectories) and `-f` (force the action).

Workflow Summary

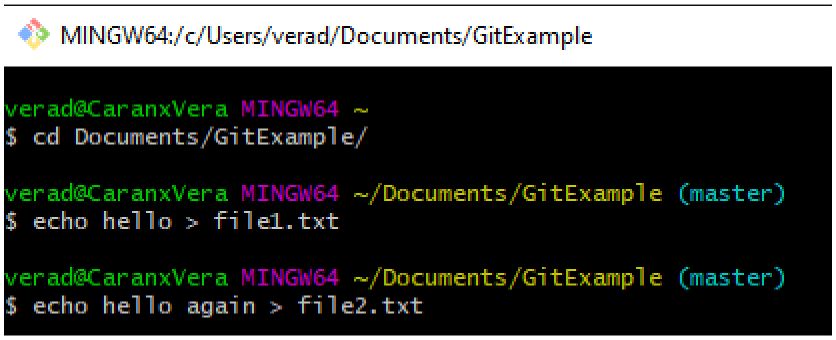
When we make changes to our files inside our project directory, we can commit them to our .git repository. This takes a snapshot of our project at this specific moment.

In Git, there is an intermediate step called the staging area or the index that permits us to review our work before making a commit. When we are done with our file modifications, we add them to the staging area. We can now review our changes, and if everything is OK, we can make a commit. The commit will be saved permanently in our repository. If we change our minds about some of the changes, we can unstage them.

The message we write to commit must make sense in order to have a useful project history.

Note: Making a commit does not empty the staging area which still contains the same version of the committed files.

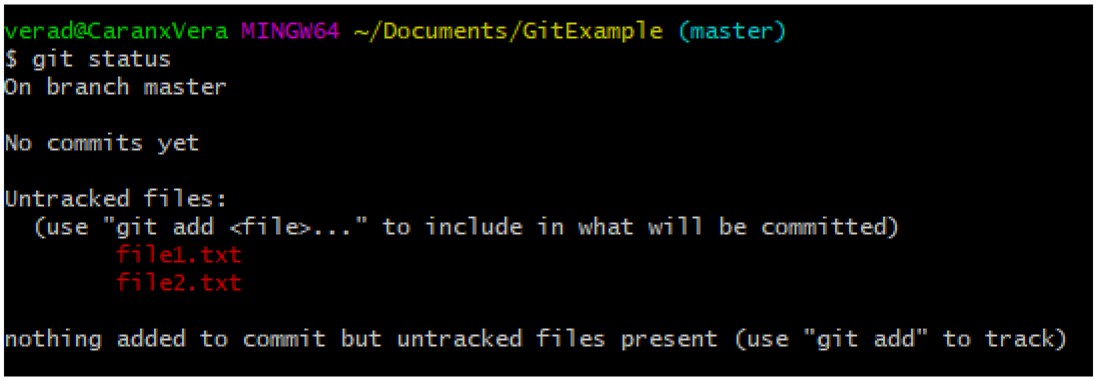
File Staging



```
MINGW64:/c/Users/verad/Documents/GitExample

verad@CaranxVera MINGW64 ~
$ cd Documents/GitExample/

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ echo hello > file1.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ echo hello again > file2.txt
```

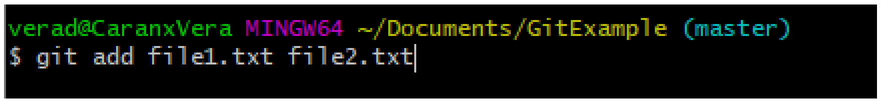
The `echo` command is a standard Unix/Linux command to write to a file. Use > to add and >> to append.

To see the status of the working directory and the staging area, we type `git status`. The file names are indicated in red to tell us that they are not in the staging area yet.



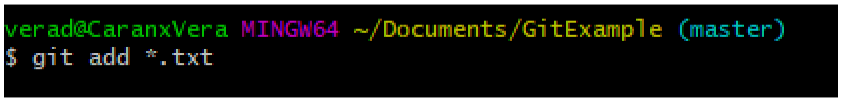
```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.txt
        file2.txt

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

To add these files to the staging area, we use the `git add` command.
To add several files, we can separate the file names by a space.



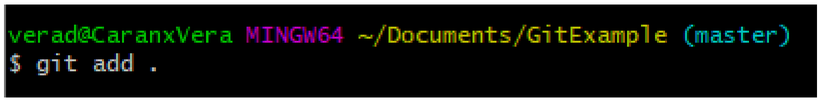
```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git add file1.txt file2.txt
```

To add multiple files, we can use patterns.
`*.txt` means all the files with a .txt extension are to be added:



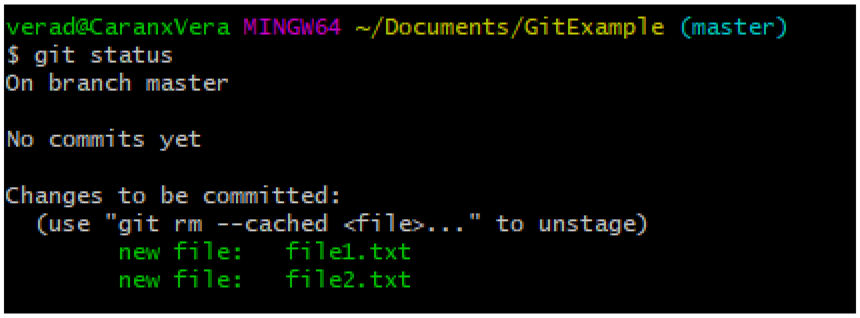
```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git add *.txt
```

To add the whole directory, we can use the `add .` command (with a dot):



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git add .
```

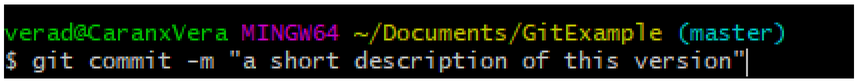
After staging the files, `git status` shows the file names in green:



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
        new file:   file2.txt
```

Committing Changes

The files added to our staging area are ready to be committed to the repository. We type `git commit -m "message to describe the commit"`. `-m` stands for message.



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git commit -m "a short description of this version"
```

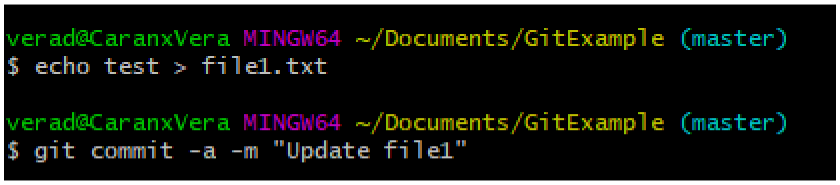
Best committing practices:

- Commits should neither be too big nor too small. We do not have to commit every time we introduce a small update. We should not wait until the whole project is over to commit either!

- Each commit is a representation of a specific change. One commit can contain fixing one bug, another one for another bug, a third one for correcting spelling mistakes, etc.
- Use a convention for commit messages: starting with verbs directly, past tense, present tense, etc.

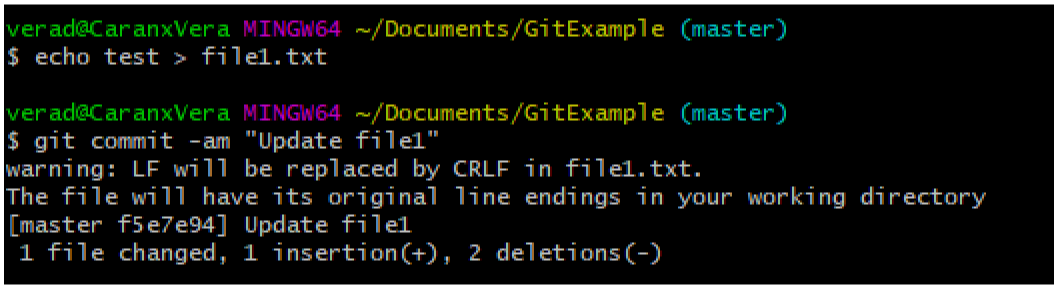
Skipping Staging

If we are sure of our changes, we can skip the staging step and commit directly.



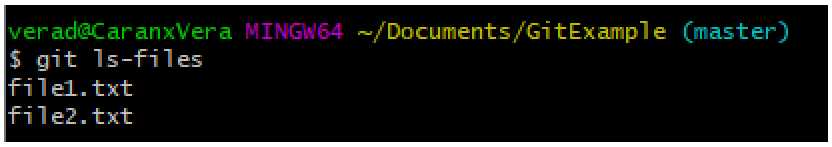
```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ echo test > file1.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git commit -a -m "Update file1"
```

Note: `-a` stands for all. We can merge both flags to `-am`



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ echo test > file1.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git commit -am "Update file1"
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory
[master f5e7e94] Update file1
 1 file changed, 1 insertion(+), 2 deletions(-)
```

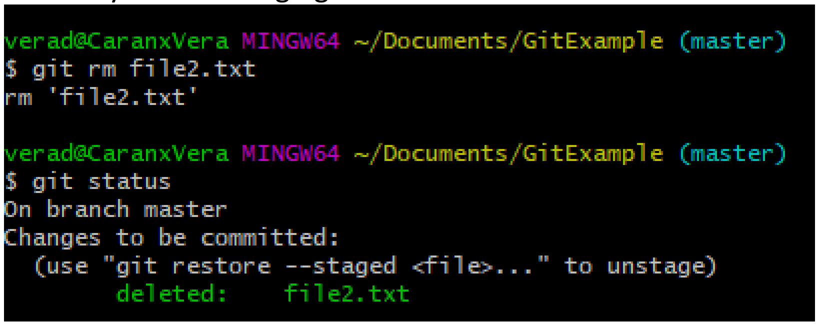
Note: We can check the files in the .git repository (staging area) by typing `git ls-files`



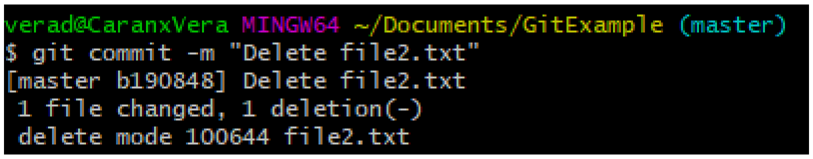
```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git ls-files
file1.txt
file2.txt
```

Removing Files

To remove a file, we can use the `git rm` command, which removes the file from both the working directory and the staging area.



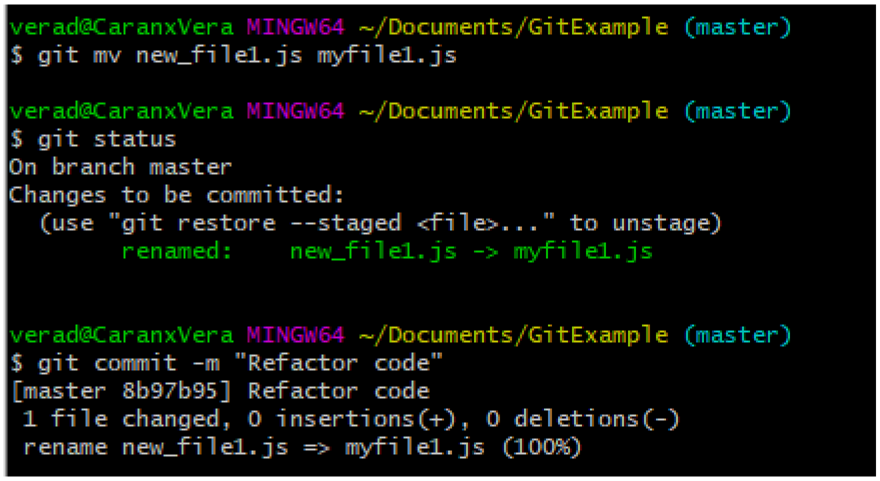
```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git rm file2.txt
rm 'file2.txt'

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    file2.txt
```



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git commit -m "Delete file2.txt"
[master b190848] Delete file2.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 file2.txt
```

Rename and Move Files

To rename or to move a file, we can use the `git mv` command.



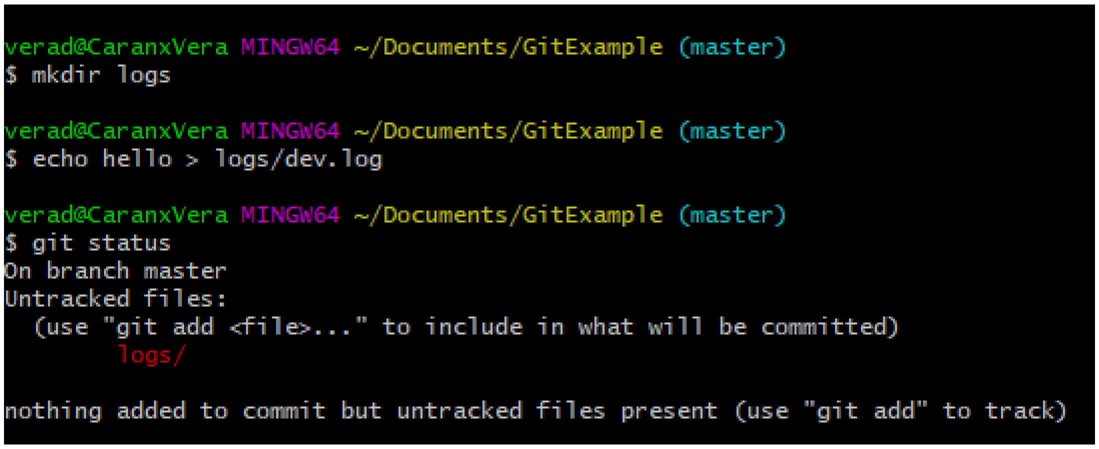
```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git mv new_file1.js myfile1.js

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    new_file1.js -> myfile1.js


verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git commit -m "Refactor code"
[master 8b97b95] Refactor code
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename new_file1.js => myfile1.js (100%)
```

Ignore Files

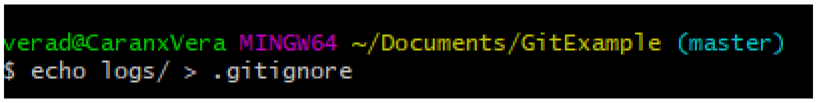
We should ignore certain files or directories that are too large or do not need to be backed up, such as data, log and binary files, etc.

For this purpose, we create a .gitignore file and type the names of the files and directories inside it that we do not wish to track. The .gitignore file can contain as many files and directories as we want. We can also easily ignore files using patterns (*.txt for example).

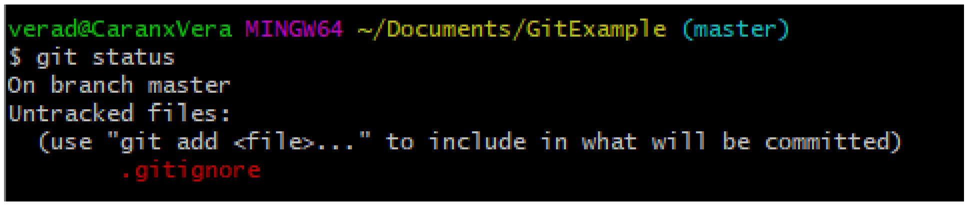
Note: .gitignore has no name, only an extension. We have to place it in the root of the project.



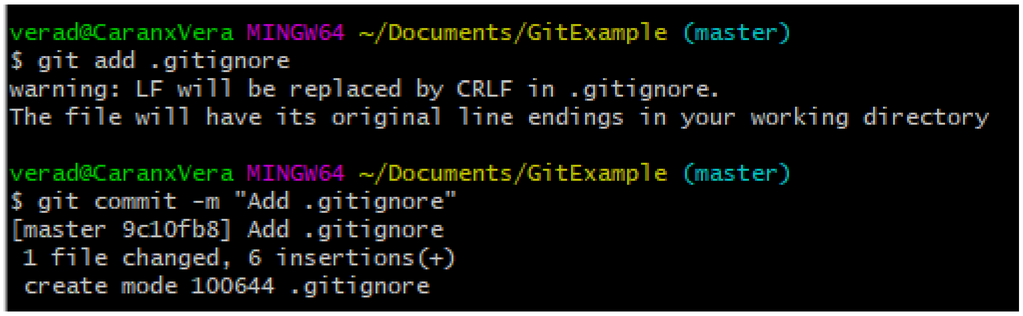
```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ mkdir logs

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ echo hello > logs/dev.log

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        logs/

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



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ echo logs/ > .gitignore
```

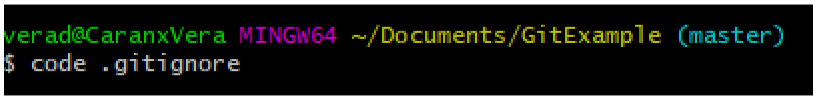


```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
```

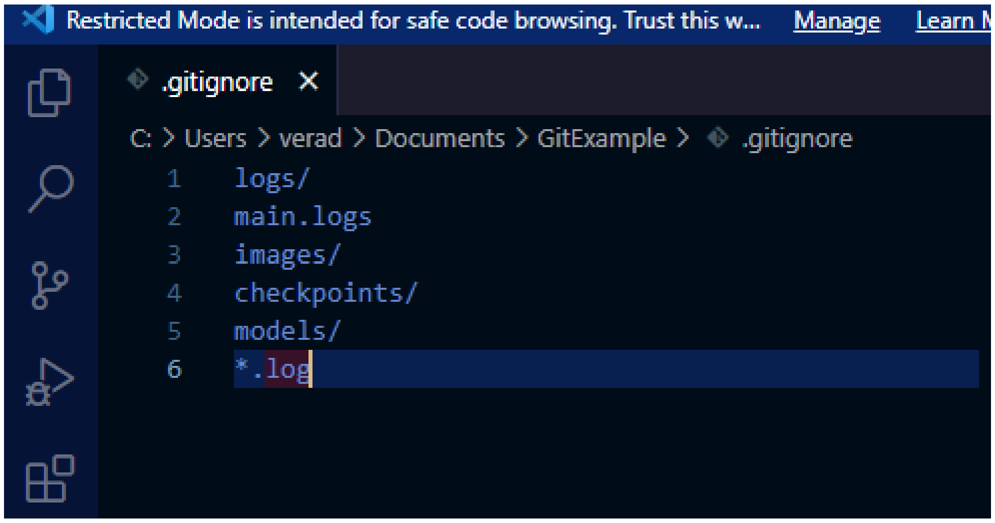


```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git add .gitignore
warning: LF will be replaced by CRLF in .gitignore.
The file will have its original line endings in your working directory

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git commit -m "Add .gitignore"
[master 9c10fb8] Add .gitignore
 1 file changed, 6 insertions(+)
 create mode 100644 .gitignore
```

To edit the .gitignore file by hand:

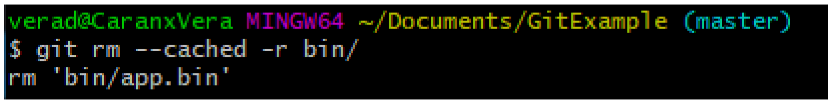


```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ code .gitignore
```

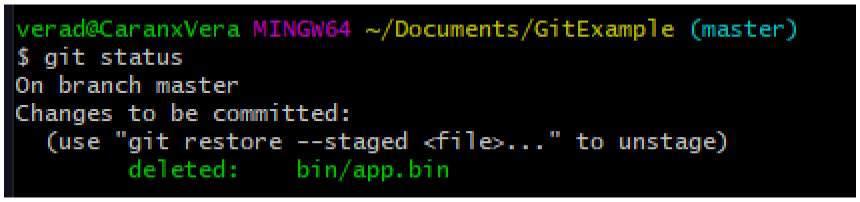

Restricted Mode is intended for safe code browsing. Trust this w...
Manage
Learn M
.gitignore
C: > Users > verad > Documents > GitExample > .gitignore

```
logs/
main.logs
images/
checkpoints/
models/
*.log
```

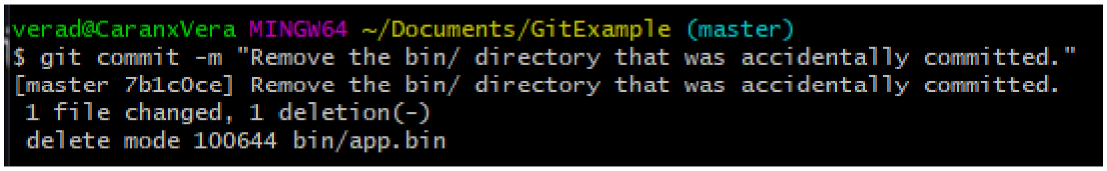
Note: .gitignore only ignores files and folders that are not included in the staging area. If we include a file in the staging area then add it to .gitignore, Git continues tracking that file.
To solve this problem, we need to remove this file or directory from the staging. We can use `git rm` to remove them. Attention is to be paid as this command works for both the working directory and the staging area. To make sure we only remove it from the staging area, we use the `--cached` flag.



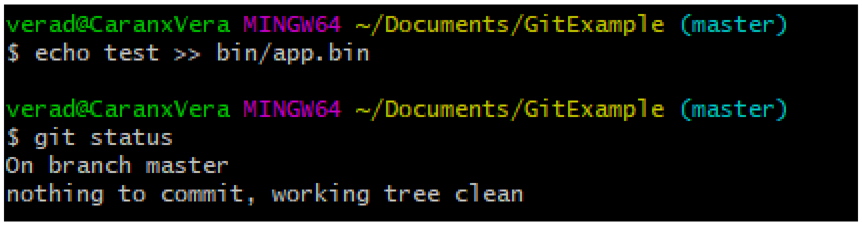
```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git rm --cached -r bin/
rm 'bin/app.bin'
```



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    bin/app.bin
```



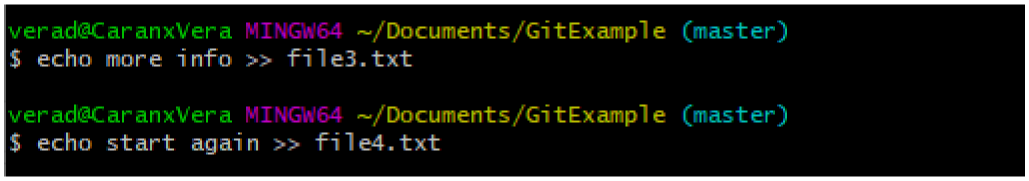
```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git commit -m "Remove the bin/ directory that was accidentally committed."
[master 7b1c0ce] Remove the bin/ directory that was accidentally committed.
 1 file changed, 1 deletion(-)
 delete mode 100644 bin/app.bin
```



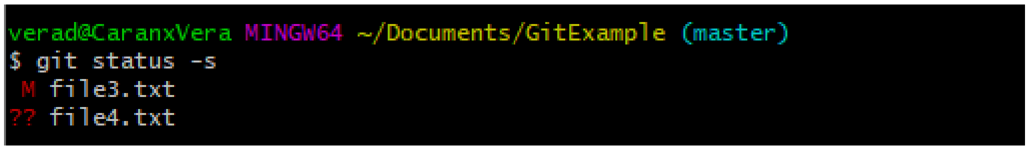
```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ echo test >> bin/app.bin

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Short Status

We can change the `git status` command to be less wordy by using the `-s` or short status flag.

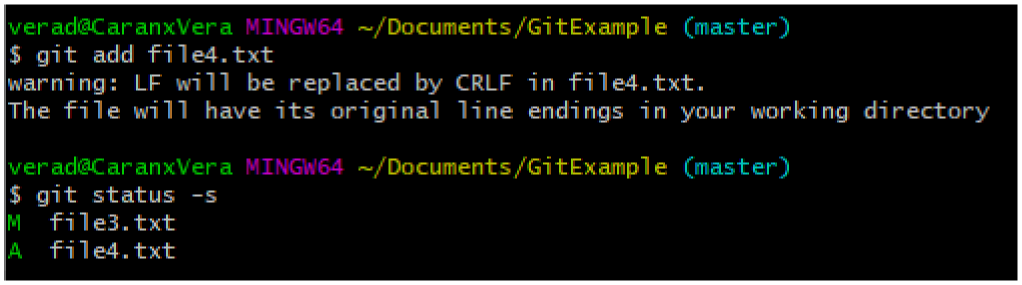


```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ echo more info >> file3.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ echo start again >> file4.txt
```



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status -s
 M file3.txt
?? file4.txt
```

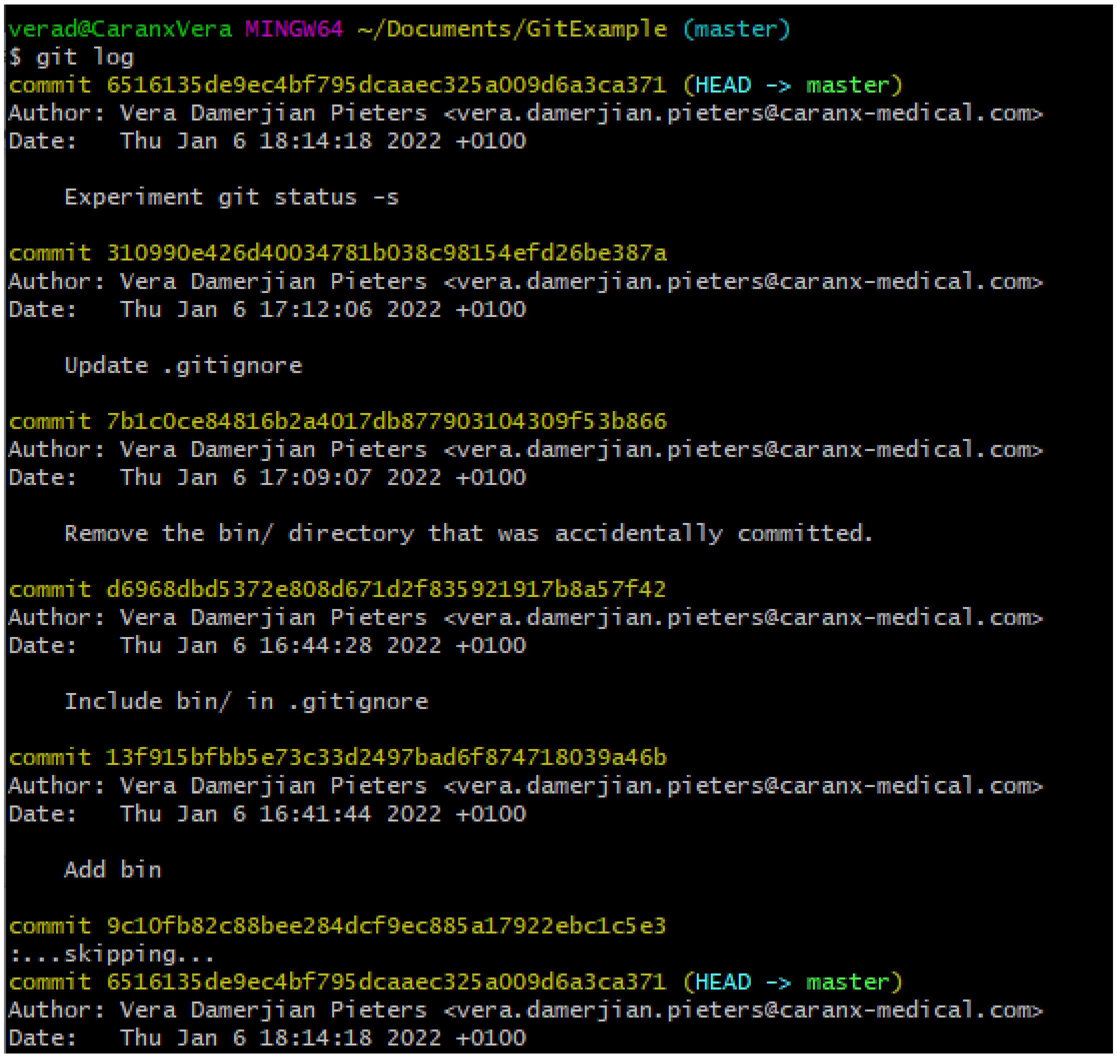
The result shows 2 columns: The left column is for the staging area and the right one is for the working directory. M means unstaged Modifications in file3.txt. `??` represent the newly created file4.txt.
When the file is added to the staging area, M becomes green and a green A depicts the added file.



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git add file4.txt
warning: LF will be replaced by CRLF in file4.txt.
The file will have its original line endings in your working directory

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status -s
M  file3.txt
A  file4.txt
```

View the History

We can use `git log` to display the commits sorted from the latest to the earliest.



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git log
commit 6516135de9ec4bf795dcaaec325a009d6a3ca371 (HEAD -> master)
Author: Vera Damerjian Pieters <vera.damerjian.pieters@caranx-medical.com>
Date:   Thu Jan 6 18:14:18 2022 +0100

    Experiment git status -s

commit 310990e426d40034781b038c98154efd26be387a
Author: Vera Damerjian Pieters <vera.damerjian.pieters@caranx-medical.com>
Date:   Thu Jan 6 17:12:06 2022 +0100

    Update .gitignore

commit 7b1c0ce84816b2a4017db877903104309f53b866
Author: Vera Damerjian Pieters <vera.damerjian.pieters@caranx-medical.com>
Date:   Thu Jan 6 17:09:07 2022 +0100

    Remove the bin/ directory that was accidentally committed.

commit d6968dbd5372e808d671d2f835921917b8a57f42
Author: Vera Damerjian Pieters <vera.damerjian.pieters@caranx-medical.com>
Date:   Thu Jan 6 16:44:28 2022 +0100

    Include bin/ in .gitignore

commit 13f915bfbb5e73c33d2497bad6f874718039a46b
Author: Vera Damerjian Pieters <vera.damerjian.pieters@caranx-medical.com>
Date:   Thu Jan 6 16:41:44 2022 +0100

    Add bin

commit 9c10fb82c88bee284dcf9ec885a17922ebc1c5e3
:...skipping...
commit 6516135de9ec4bf795dcaaec325a009d6a3ca371 (HEAD -> master)
Author: Vera Damerjian Pieters <vera.damerjian.pieters@caranx-medical.com>
Date:   Thu Jan 6 18:14:18 2022 +0100
```

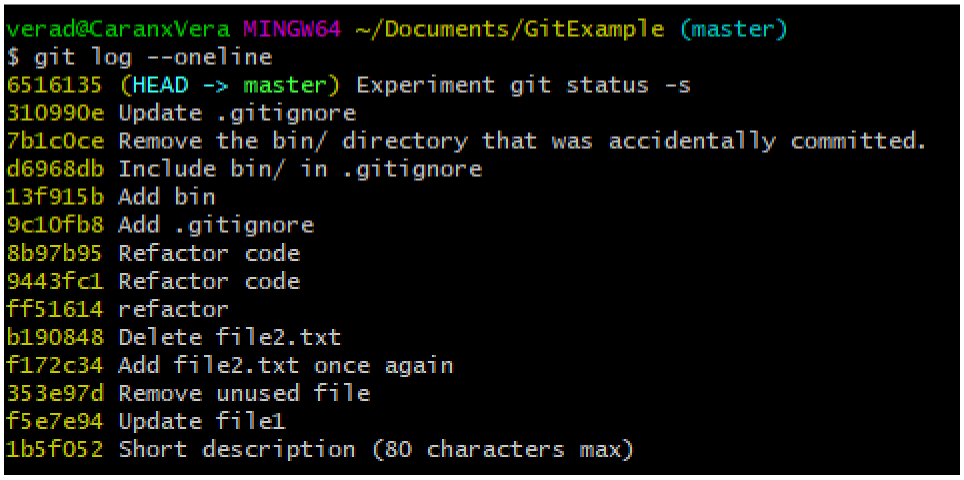
Each commit has a unique ID (a 40-character hexadecimal string) generated automatically by Git. Author information, date and time, and the commit message are also shown.

We can click on space to change pages if the commit list covers several pages. We can use the arrows to go up and down and quit by pressing `q`.

The main working branch in Git is called ‘master’ or ‘main’.

‘HEAD’ is a reference (pointer) to the current branch which tells Git which branch we are currently working on.

To display a short summary of the commits, we use `git log --oneline`.

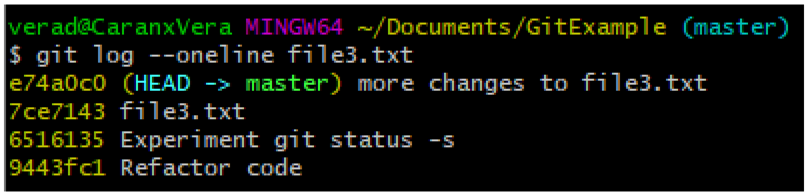


```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git log --oneline
6516135 (HEAD -> master) Experiment git status -s
310990e Update .gitignore
7b1c0ce Remove the bin/ directory that was accidentally committed.
d6968db Include bin/ in .gitignore
13f915b Add bin
9c10fb8 Add .gitignore
8b97b95 Refactor code
9443fc1 Refactor code
ff51614 refactor
b190848 Delete file2.txt
f172c34 Add file2.txt once again
353e97d Remove unused file
f5e7e94 Update file1
1b5f052 Short description (80 characters max)
```

The unique identifier is now shortened to 7 characters, and a one-line commit message is shown per commit.

To see the earliest commits first, we can use the `--reverse` flag.

To see the history of a specific file, we type:



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git log --oneline file3.txt
e74a0c0 (HEAD -> master) more changes to file3.txt
7ce7143 file3.txt
6516135 Experiment git status -s
9443fc1 Refactor code
```

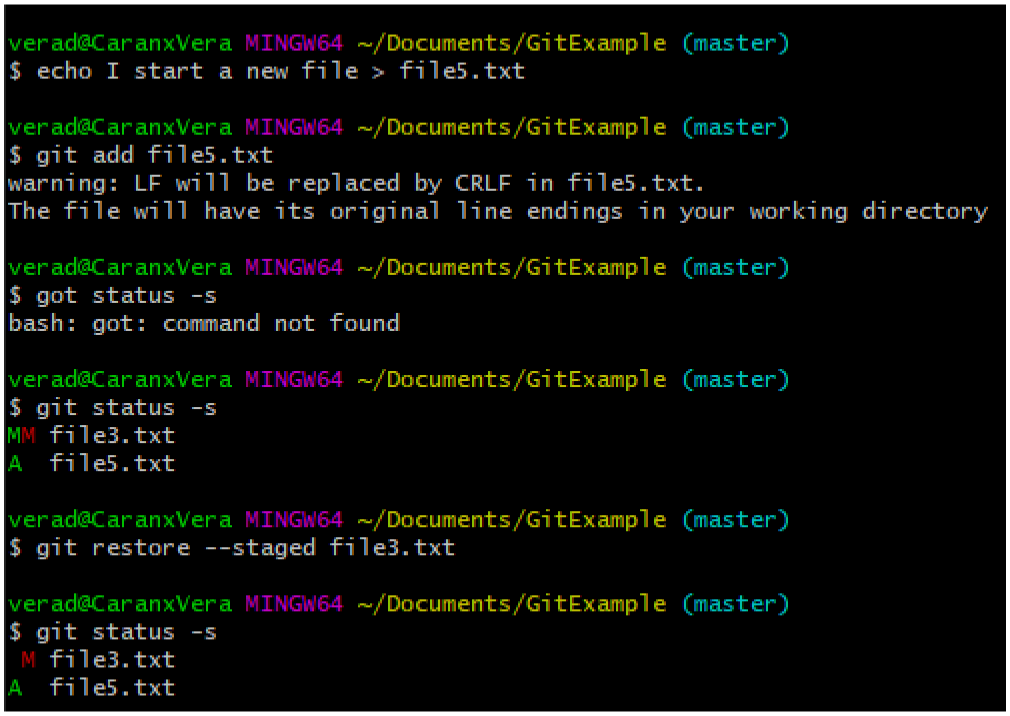
Note: If Git complains about the file name, consider separating this name from the options before it:

```
git log --oneline -- file4.txt
```

The file name has to be at the very end of the statement. All options should go before it.

Unstage Files

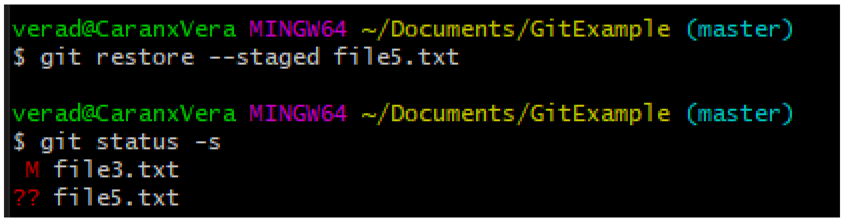
If we wish to unstage some files after adding them because we are unhappy with their revision, we can use `git restore`.



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ echo I start a new file > file5.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git add file5.txt
warning: LF will be replaced by CRLF in file5.txt.
The file will have its original line endings in your working directory

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ got status -s
bash: got: command not found

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status -s
MM file3.txt
A  file5.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git restore --staged file3.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status -s
 M file3.txt
A  file5.txt
```

Notice that file3.txt is not in the staging area because it is shown in red (which means that it's in the working directory.

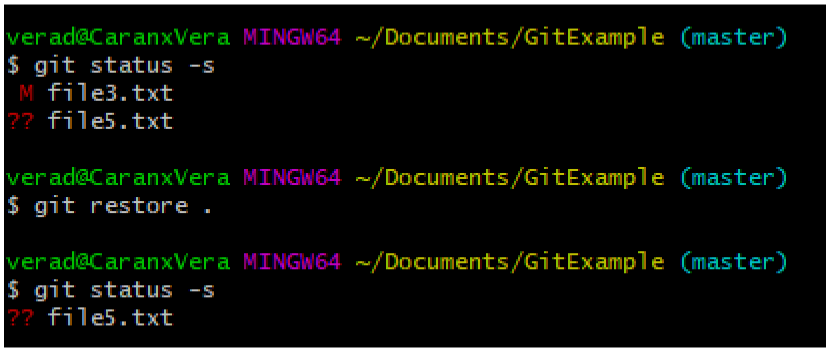
If we restore a file that was newly added to the staging area, Git will take it back to its original untracked state represented by question marks.



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git restore --staged file5.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status -s
 M file3.txt
?? file5.txt
```

Discard Local Changes

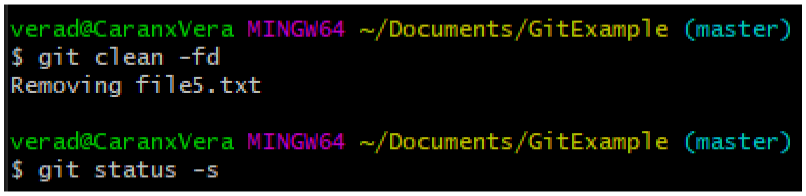
We can choose to discard changes that we made in our local directory by using the `git restore` command. Git then takes these restored files to their previous state.



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status -s
 M file3.txt
?? file5.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git restore .

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status -s
?? file5.txt
```

Notice that when we restore local files, the newly created untracked files will still present because Git does not know what to do with them as they do not have a previous state.

To remove new untracked files, we use `git clean`:

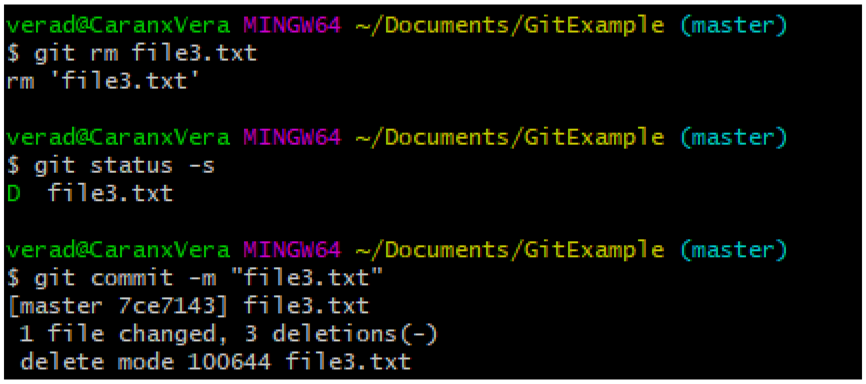


```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git clean -fd
Removing file5.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status -s
```

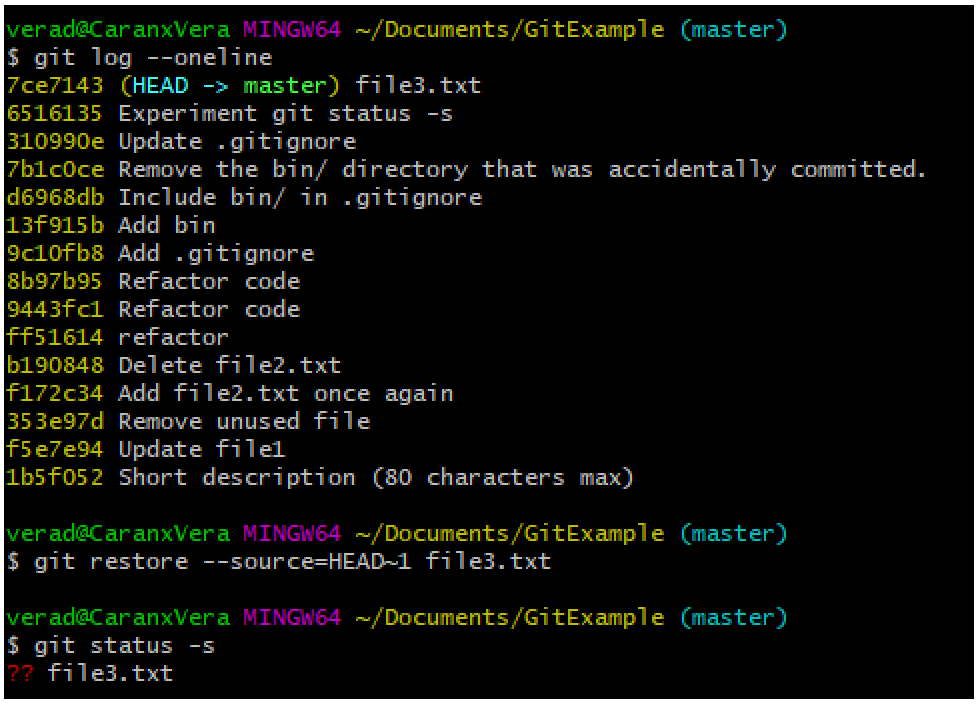
Restore a Deleted File to an Earlier Version

To restore a deleted file that has been committed, we can restore it from a specific commit by using

```
git restore --source=commit_ID file_name.ext
```



```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git rm file3.txt
rm 'file3.txt'

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status -s
D  file3.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git commit -m "file3.txt"
[master 7ce7143] file3.txt
 1 file changed, 3 deletions(-)
 delete mode 100644 file3.txt
```


```
verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git log --oneline
7ce7143 (HEAD -> master) file3.txt
6516135 Experiment git status -s
310990e Update .gitignore
7b1c0ce Remove the bin/ directory that was accidentally committed.
d6968db Include bin/ in .gitignore
13f915b Add bin
9c10fb8 Add .gitignore
8b97b95 Refactor code
9443fc1 Refactor code
ff51614 refactor
b190848 Delete file2.txt
f172c34 Add file2.txt once again
353e97d Remove unused file
f5e7e94 Update file1
1b5f052 Short description (80 characters max)

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git restore --source=HEAD~1 file3.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample (master)
$ git status -s
?? file3.txt
```

A new untracked file now appears in the working directory which corresponds to the restored file from the specified commit.

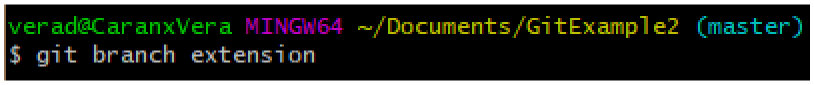
Branches

Branching creates divergence from the main work line, which is the default ‘master’ branch, so that a user can work in isolation. This helps keep the main branch as stable as possible.

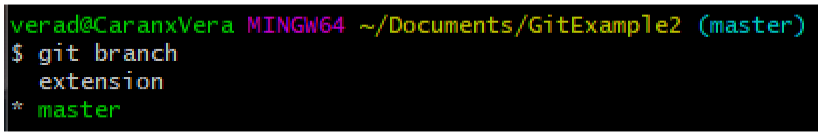
When we create a new branch and switch to it, Git creates a new pointer that moves to the latest commit.

Git knows which branch we are working on by using the HEAD pointer.

Let’s create a new branch called ‘extension’:

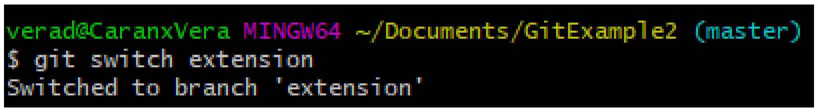

```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (master)
$ git branch extension
```

To check the list of branches:


```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (master)
$ git branch
  extension
* master
```

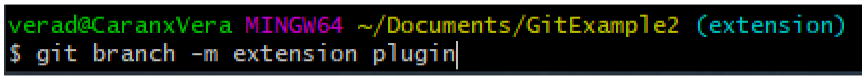
The asterisk before ‘master’ means that we are currently on the master branch.

To switch to a new branch:



```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (master)
$ git switch extension
Switched to branch 'extension'
```

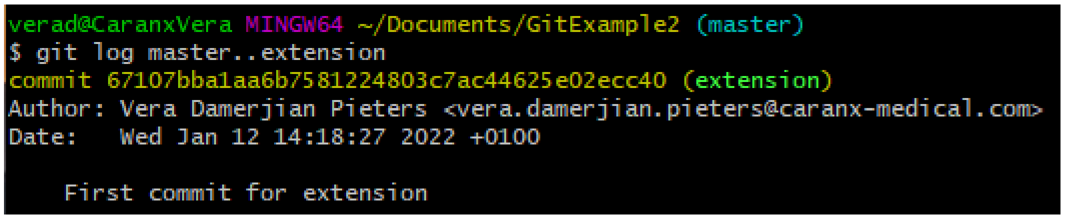
To rename a branch, we use `git branch -m old_name new_name`.



```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (extension)
$ git branch -m extension plugin
```

Note: The changes we made are only visible in the new branch. Therefore, if we switch back to the master branch and display the same file, we will get its old version. This is how we isolate our work with branching.

To show all the commits that are in the 'extension' branch and not in 'master':



```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (master)
$ git log master..extension
commit 67107bba1aa6b7581224803c7ac44625e02ecc40 (extension)
Author: Vera Damerjian Pieters <vera.damerjian.pieters@caranx-medical.com>
Date:   Wed Jan 12 14:18:27 2022 +0100

    First commit for extension
```

Note that because we are currently on 'master', we do not have to rename it every time. Therefore, we can simply use:

```
git log extension
```

Merge Branches

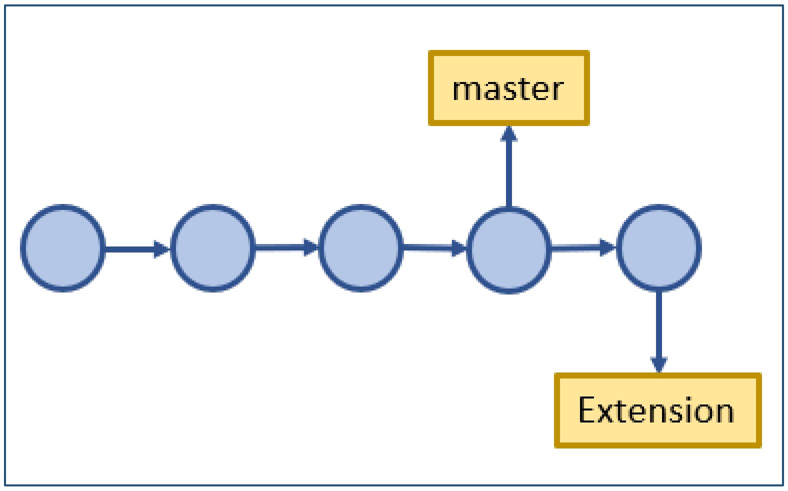
Merging is joining changes from one branch with another.

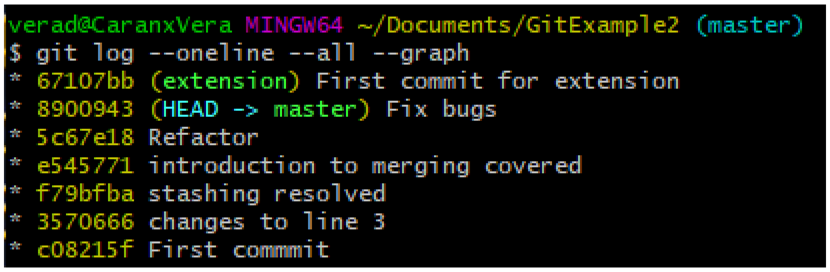
There are 2 types of merging in Git:

- Fast-forward
- Three-way

Note: Always include the `--graph` option while working with branches as it allows a better commit visualization.

1. Fast-Forward Merge:
   This is useful when no changes have been made to the 'master' branch. Git fast-forwards the 'master' pointer to point to the last commit of the new branch.

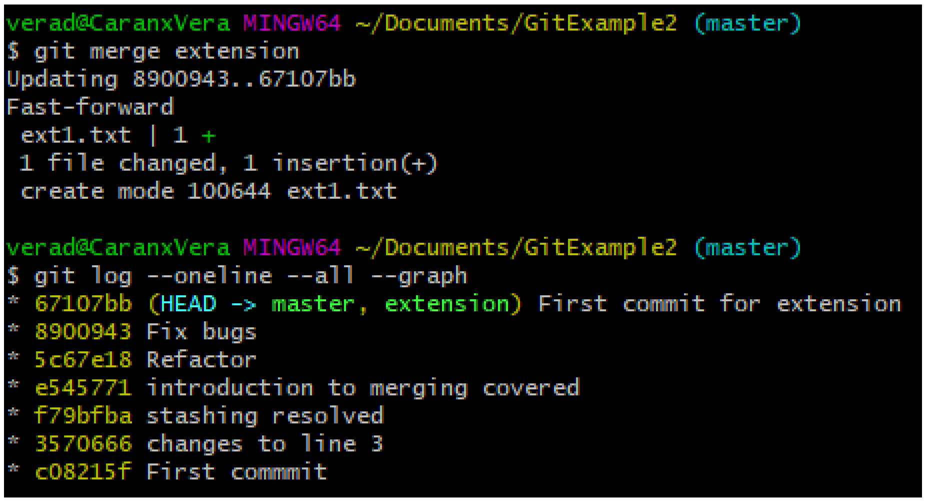




```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (master)
$ git log --oneline --all --graph
* 67107bb (extension) First commit for extension
* 8900943 (HEAD -> master) Fix bugs
* 5c67e18 Refactor
* e545771 introduction to merging covered
* f79bfba stashing resolved
* 3570666 changes to line 3
* c08215f First commmit
```

Notice that the ‘extension’ branch is one commit ahead of master, and no divergence exists between the ‘extension’ and ‘master’ branches. Therefore, when we merge those branches, Git will fast-forward ‘master’.

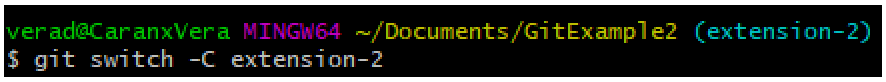
To merge changes from another branch to ‘master’, we need to switch to ‘master’:



```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (master)
$ git merge extension
Updating 8900943..67107bb
Fast-forward
 ext1.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 ext1.txt

verad@CaranxVera MINGW64 ~/Documents/GitExample2 (master)
$ git log --oneline --all --graph
* 67107bb (HEAD -> master, extension) First commit for extension
* 8900943 Fix bugs
* 5c67e18 Refactor
* e545771 introduction to merging covered
* f79bfba stashing resolved
* 3570666 changes to line 3
* c08215f First commmit
```

Notice that both branches are now pointing to the same commit.

Instead of creating a new branch then switching to it, we can do both in one step using the `-C` option for create:



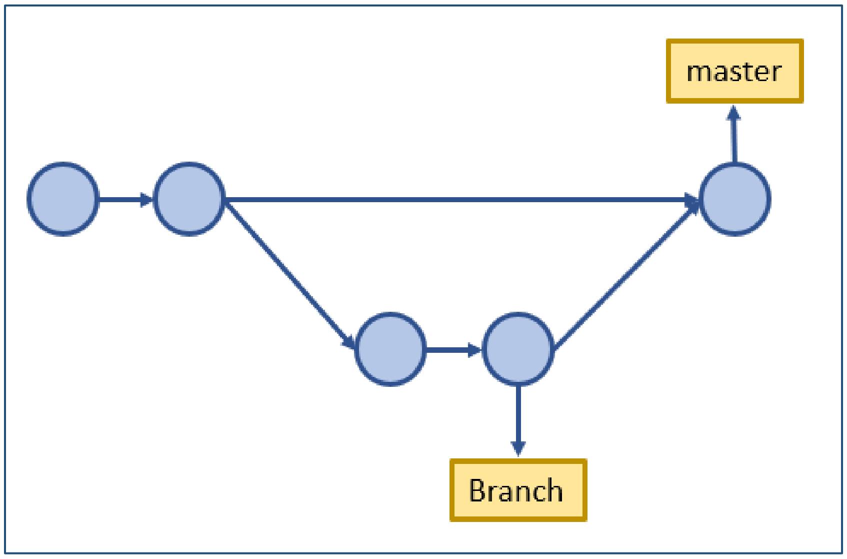
```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (extension-2)
$ git switch -C extension-2
```

We can also merge branches by disabling the fast-forward merging. As a result, even if it is possible to do fast-forward merging, Git will not do it. It will create a merge commit that will combine the changes in both branches instead.

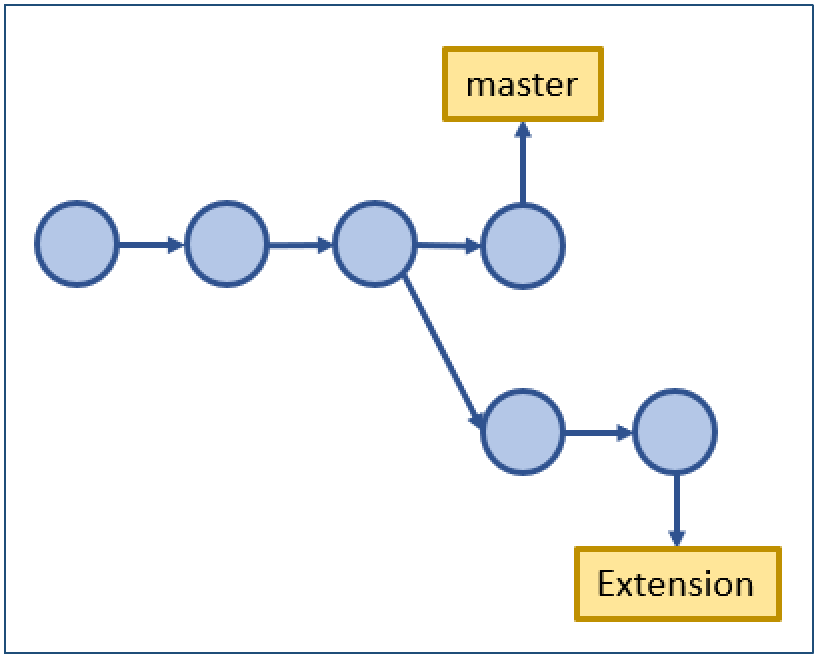
To do this type of merge, first switch to master then type:

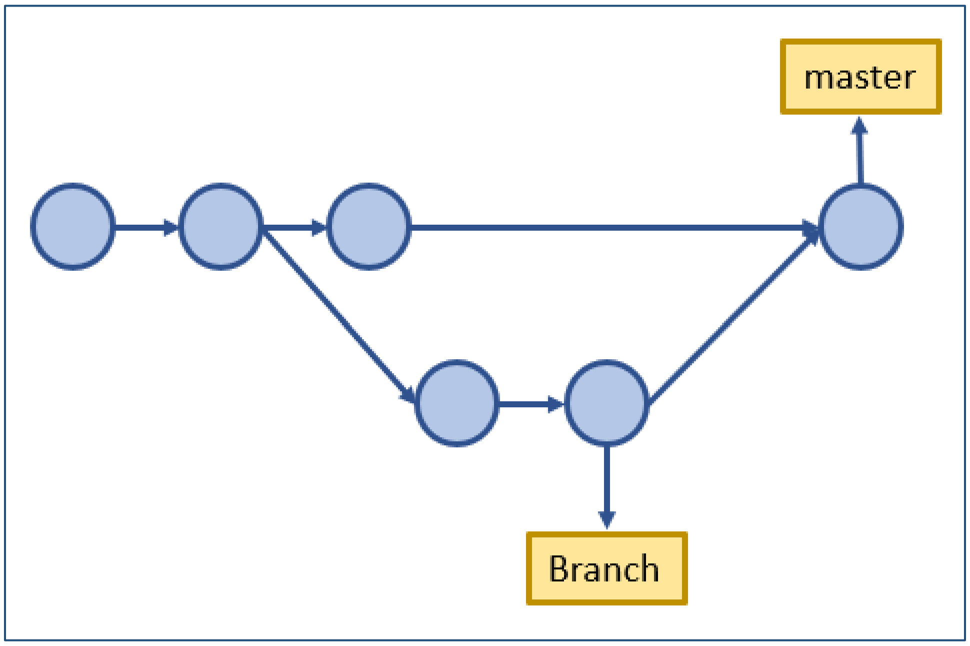
```
git merge -no-ff extension-2
```

No-ff commits make it easier to revert or undo a feature because they create a single commit to group the changes. Therefore, to undo a commit, we would only need to revert the merge commit, whereas in plain fast-forward merging, we would have to revert more commits.

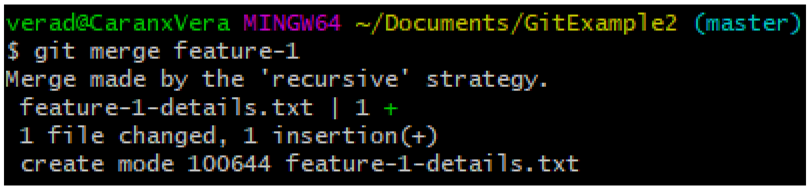


2. Three-Way Merge:
   This is used when changes have been made to both branches. A new commit is therefore created that combines the changes from both branches.





When there is a divergence between branches, Git does a three-way merge. It checks the tips of both branches and compares them to the common ancestor branch. It notices their differences and decides to do three-way merge by creating a new merge commit to group all changes from both branches.

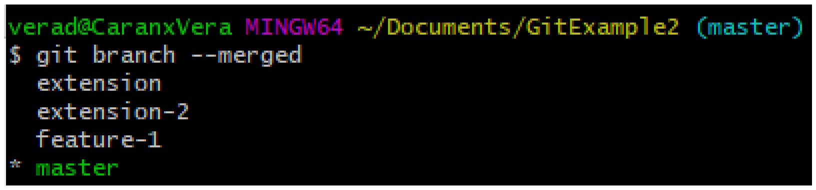


```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (master)
$ git merge feature-1
Merge made by the 'recursive' strategy.
 feature-1-details.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 feature-1-details.txt
```

Delete Branches

After we are done working on a branch, we should not forget to merge it with master and delete it.

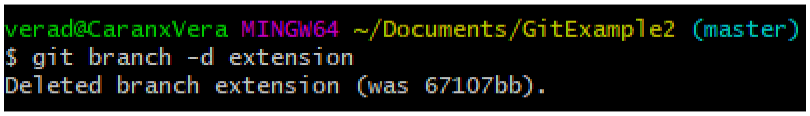
To view the list of branches that we have already merged to the 'master' branch:



```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (master)
$ git branch --merged
  extension
  extension-2
  feature-1
* master
```

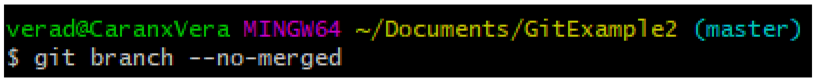
It is safe to delete these branches if our work is done.

To delete a branch:



```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (master)
$ git branch -d extension
Deleted branch extension (was 67107bb).
```

To show the unmerged branches:



```
verad@CaranxVera MINGW64 ~/Documents/GitExample2 (master)
$ git branch --no-merged
```

Collaboration

Git is a distributed version control system. This means that every developer has a local repository to work offline independently. To collaborate, developers can use a central repository to synchronize all the work. This can be a server on a private company network or on a Git hosting service (cloud) such as GitHub, GitLab, BitBucket, etc. We can set this cloud repository as private to ensure accessibility to team members only.

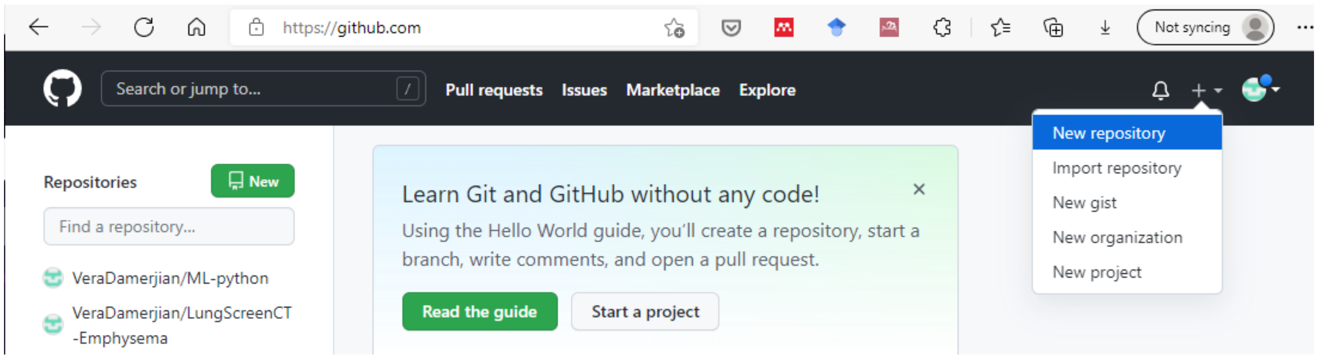
There are 2 types of workflows:

- Centralized workflows: Used for private projects
- Integration-manager workflows: Used in open-source projects

GitHub

GitHub is the most popular Git hosting platform, used to host the central repository of a team on the cloud.

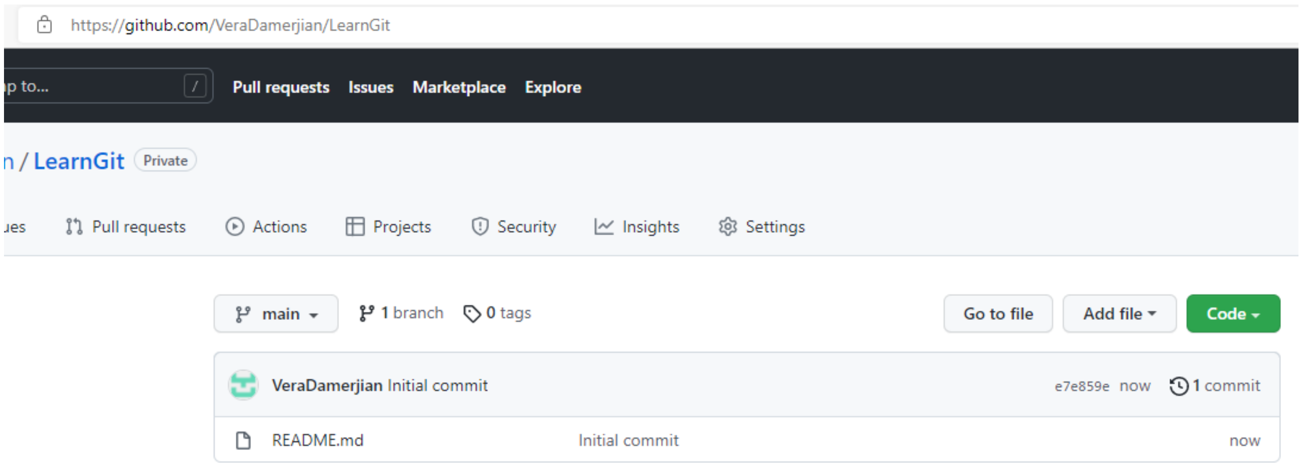
We create an account on *github.com* then make a new repository:



We choose a name for our repository.

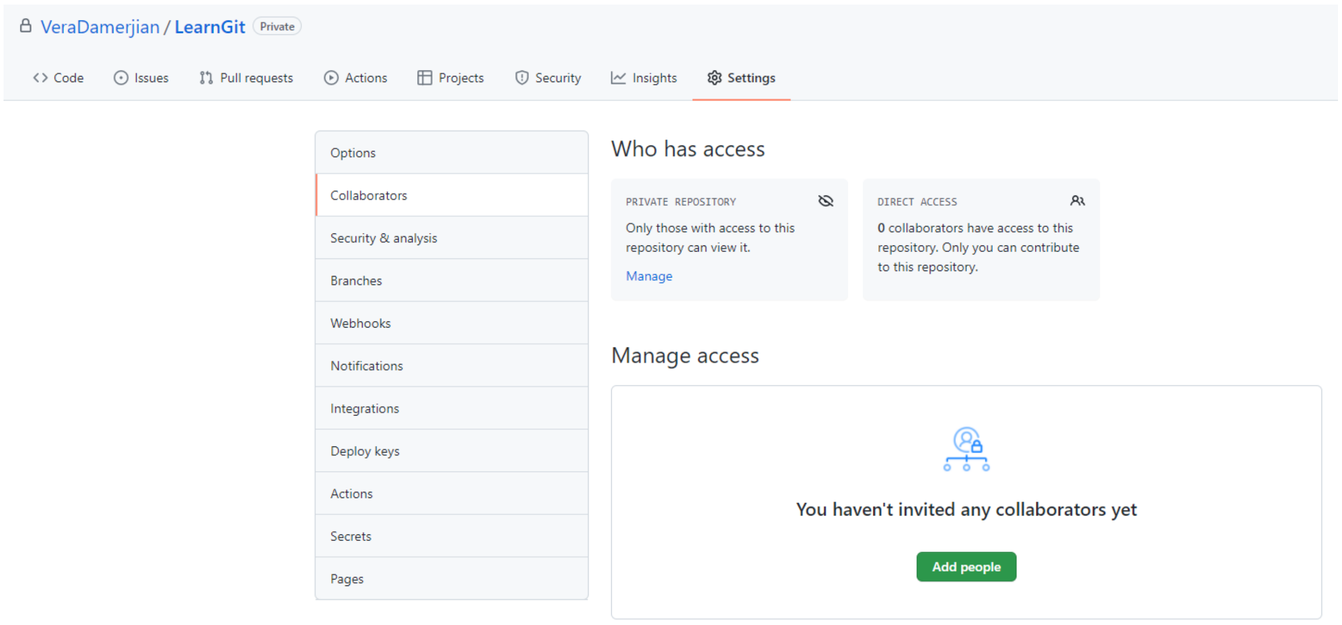
We decide if our repository is private or public. Public repositories can be seen by everyone on the internet, so they are useful for open-source projects. For private repositories, we can add users of our choice to visualize and commit to this repository.

If we choose to initialize the repository with a README file, GitHub creates this file and sends the first commit.



Add Collaborators

We can add team members and give them push access to the repository. This sends invitations to the specified e-mail addresses to gain access to the repository.



Who has access

PRIVATE REPOSITORY

Only those with access to this repository can view it.

Manage

DIRECT ACCESS

0 collaborators have access to this repository. Only you can contribute to this repository.

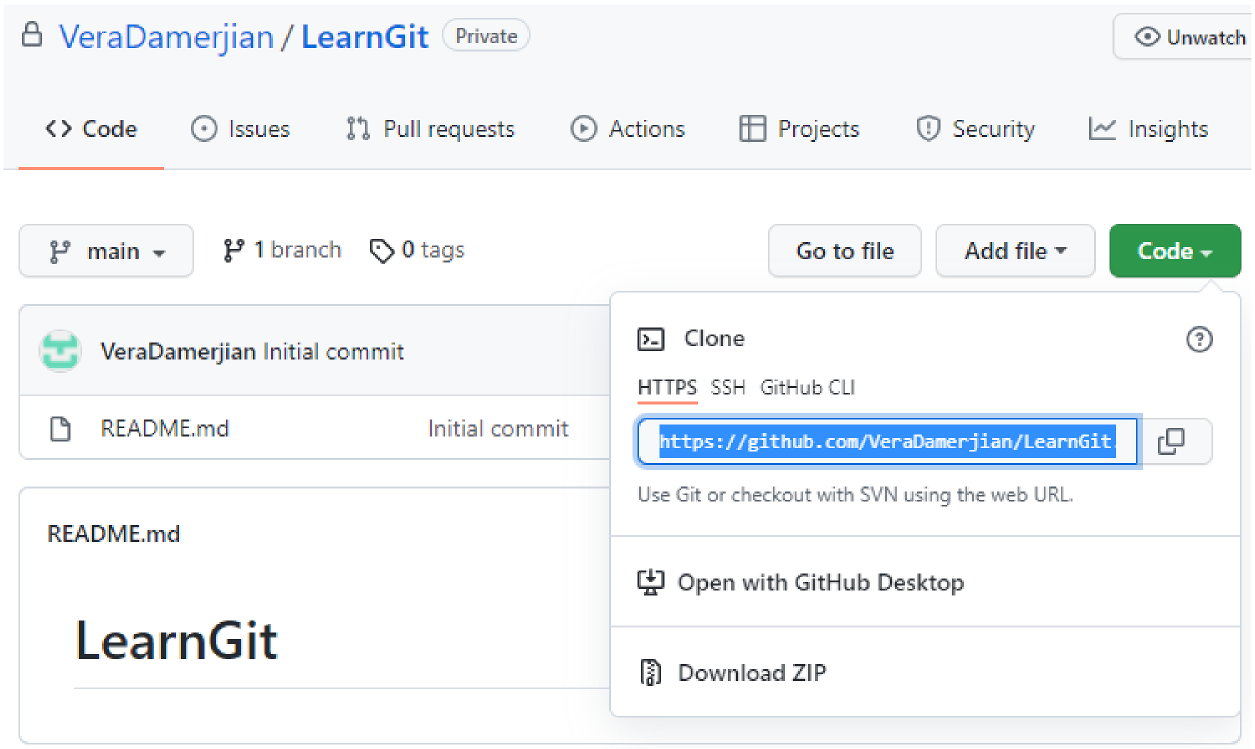
Manage access

You haven't invited any collaborators yet

Cloning a Repository

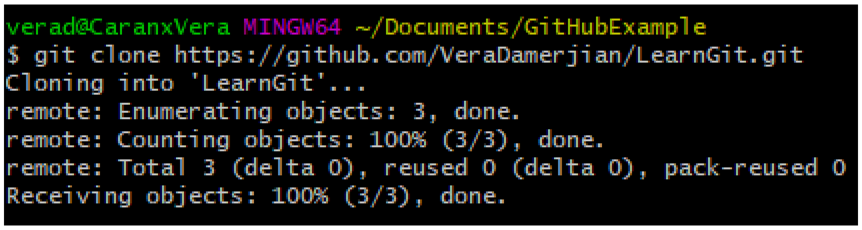
Every member must have a copy of the project available on GitHub. This is possible by cloning.

To clone a repository, we click on the 'code' button shown in green and copy the link provided:



```
https://github.com/VeraDamerjian/LearnGit
```

Use Git or checkout with SVN using the web URL.

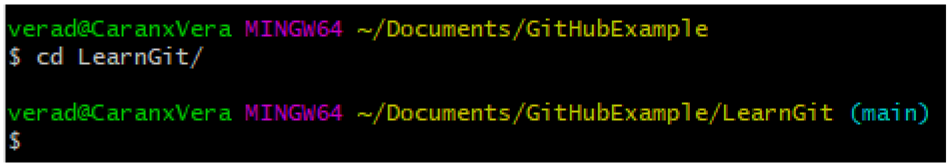


```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample
$ git clone https://github.com/VeraDamerjian/LearnGit.git
Cloning into 'LearnGit'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

Now Git creates a directory in the local repository with the same name and copies all the files on the cloud in it.

Users can now work locally in isolation and push their changes back to GitHub when they wish.

We go to the cloned repository. Note the word ‘main’ in blue indicating that this is a Git repository:



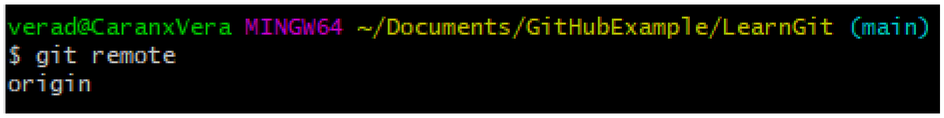
```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample
$ cd LearnGit/

verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (main)
$
```

When we clone a repository, Git names the cloud repository ‘origin’. Therefore, ‘origin/master’ indicates where the master pointer is in the remote GitHub repository.

Similarly, ‘origin/HEAD’ indicates where the HEAD pointer is in the origin repository.

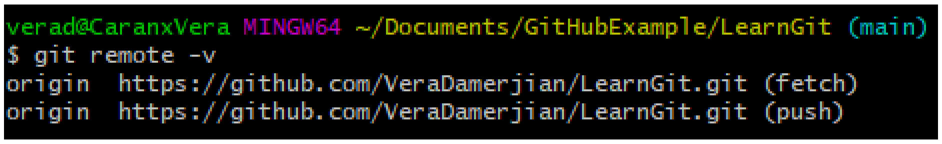
Origin/master and origin/HEAD are called remote tracking branches.

To show the list of remote repositories:



```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (main)
$ git remote
origin
```

For more details, we use the −v option for verbose:



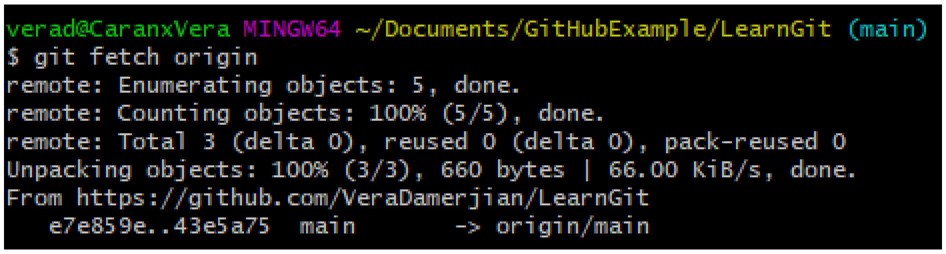
```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (main)
$ git remote -v
origin  https://github.com/VeraDamerjian/LearnGit.git (fetch)
origin  https://github.com/VeraDamerjian/LearnGit.git (push)
```

This displays the URL links that Git uses when it communicates with the origin (remote) repository.

Fetch and Merge

When new commits are made in the remote repository, the local repository is not aware of the changes automatically.

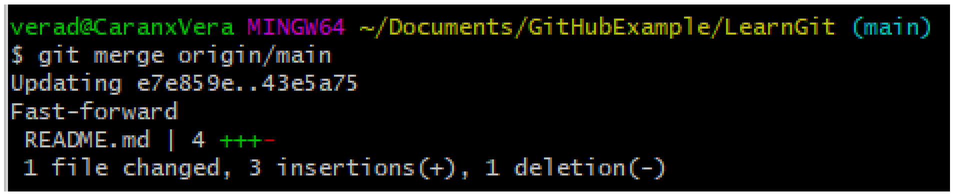
We must use the `fetch` command to download the new commit.



```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (main)
$ git fetch origin
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 660 bytes | 66.00 KiB/s, done.
From https://github.com/VeraDamerjian/LearnGit
   e7e859e..43e5a75  main       -> origin/main
```

Now that the new commit is downloaded, we need to update our local working directory. Therefore, we make sure that we are on the main (master) branch, and we type:

```
git merge origin/master
```



```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (main)
$ git merge origin/main
Updating e7e859e..43e5a75
Fast-forward
 README.md | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Pull

Pull = Fetch + Merge

When we apply `git pull`, Git downloads the new commit from the remote GitHub repository and applies a three-way merge to update the local main with the new changes.

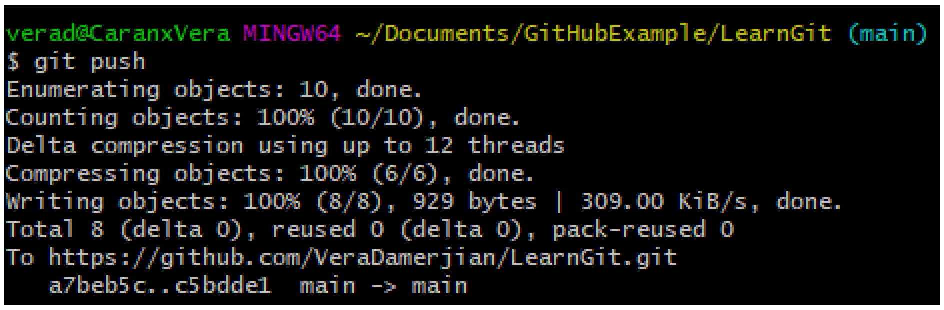
Push

The `push` command is used to update the remote repository for it to look like our local working directory:

```
git push remote_repo current_repo
```

Since we are already on the main branch, we do not have to specify main as the local branch.

We can also remove the word ‘origin’ as it will be assumed by default.


```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (main)
$ git push
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 12 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (8/8), 929 bytes | 309.00 KiB/s, done.
Total 8 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/VeraDamerjian/LearnGit.git
   a7beb5c..c5bdde1  main -> main
```

Now, if we go to our GitHub repository, we can see the commit.

Note: When the remote and the local histories evolve differently, the `push` command will be rejected.

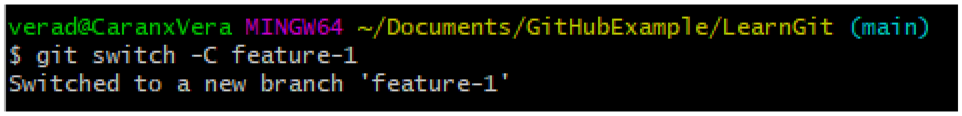
Using the `-f` (force) option will drop someone else's work on the remote repository to add ours. Therefore, we should NEVER force-push unless we have an extremely good reason.

Note: When histories evolve differently, we first need to `pull` the remote repository to update the local working directory. We then resolve any conflicts. Now we are ready to do a `push`.

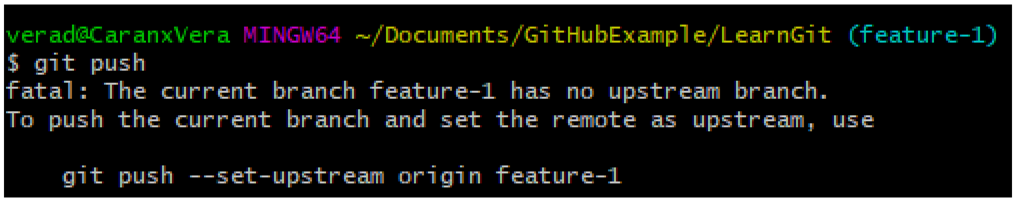
Share Branches

Branches are not automatically pushed to the remote repository. If we want to share them with other users, we have to push them on their own.

We create a new branch locally:

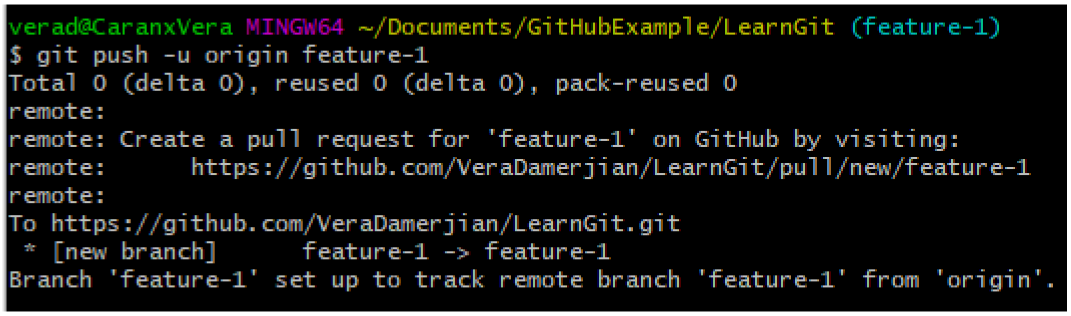

```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (main)
$ git switch -C feature-1
Switched to a new branch 'feature-1'
```

When we try to push it, Git tells us that it is not linked to a branch in origin.


```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (feature-1)
$ git push
fatal: The current branch feature-1 has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature-1
```

The main branch is linked to the origin/master branch, but the other branch is not linked to any branches remotely.

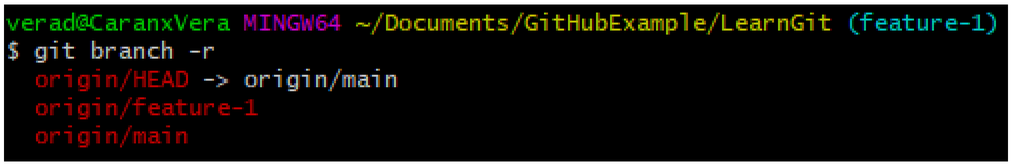
To link the local branch to a remote branch:



```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (feature-1)
$ git push -u origin feature-1
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'feature-1' on GitHub by visiting:
remote:      https://github.com/VeraDamerjian/LearnGit/pull/new/feature-1
remote:
To https://github.com/VeraDamerjian/LearnGit.git
 * [new branch]      feature-1 -> feature-1
Branch 'feature-1' set up to track remote branch 'feature-1' from 'origin'.
```

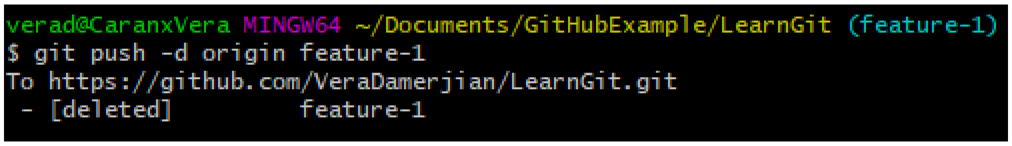
Note: `-u` stands for upstream. It is used the first time we push a local branch to upstream.

To see all the remote branches:



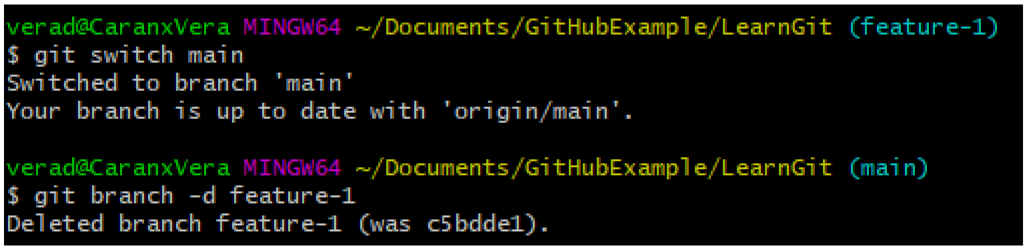
```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (feature-1)
$ git branch -r
  origin/HEAD -> origin/main
  origin/feature-1
  origin/main
```

After working with the branch, we would need to remove it. To remove it remotely:



```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (feature-1)
$ git push -d origin feature-1
To https://github.com/VeraDamerjian/LearnGit.git
 - [deleted]         feature-1
```

Even if we remove the branch from origin, the corresponding local branch is still present. To remove it, we switch to main and delete the branch locally:



```
verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (feature-1)
$ git switch main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

verad@CaranxVera MINGW64 ~/Documents/GitHubExample/LearnGit (main)
$ git branch -d feature-1
Deleted branch feature-1 (was c5bdde1).
```