Git Lab 1
Creating and tracking changes

1 Commands covered

Configuration	Initializing a repo

`git config --global user.name`
`git config --global user.email`
`git config --global color.ui auto`
`git config --list --show-origin`
`git config --global core.editor`
 	
Atlassian config command
 	
TheServerSide
	`git init`
	
	Atlassian git command
	
	Atlassian repo tutorial

Saving changes	Tracking changes

`git add`
`git add --all`
 	
Atlassian saving changes tutorial
 	
CareerKarma
	`git status`
	
	Atlassian inspecting repository
	
	git status tutorial

`git commit -m`
`git commit -a`
`git commit -am`
	`git log`
	
	FreecodeCamp git log command

Atlassian git commit command BitDegree git commit command	Atlassian git log command

`git rm`
`git mv`
 	
Atlassian git rm tutorial
 	
GitTower git rm tutorial
 	
GitHowTo moving files
 	
More details on git mv	

2 Lab setup

Make sure you have a suitable text editor installed for your OS.

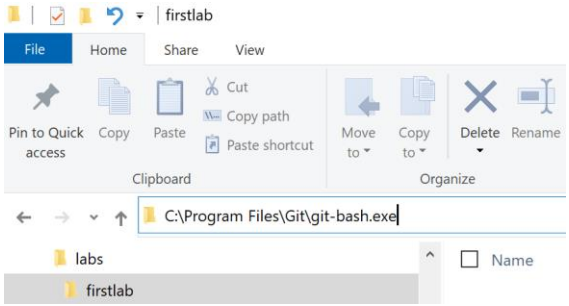
Create a new empty directory with the name `labs` in a location that you have read/write access to (for e.g. if you are using a company desktop / laptop for this workshop, you probably only have read/write access within your home directory while you are logged into your account). If so, you could create a directory like this (`Desktop\git-training\labs`)

This `labs` directory will serve as the top-level directory to hold the various Git project folders that we will be working with in the upcoming labs. Create a subdirectory within this main `labs` directory with the name `firstlab`. We will be initializing our first Git project in the `firstlab` subdirectory.

IMPORTANT: If you are new to Git, **PLEASE FOLLOW** all the folder names suggested above to avoid confusion with folder names at a later point in a lab.

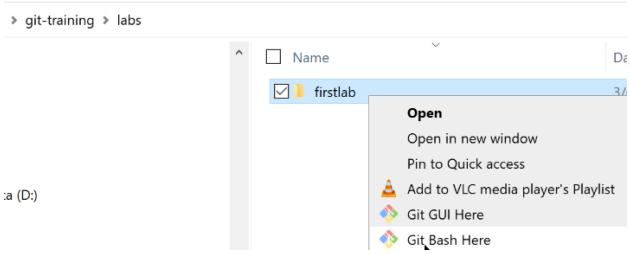
If you are working with Windows, we will be typing in the commands via Git Bash. To open the Git Bash shell in the `firstlab` folder, there are 2 approaches:

Approach 1:
With the File Explorer open in the `firstlab` folder, type
`C:\Program Files\Git\git-bash.exe`
into the File Explorer address bar, and press Enter



If Git Bash is installed properly at the default location indicated above, it should start up at that given location.

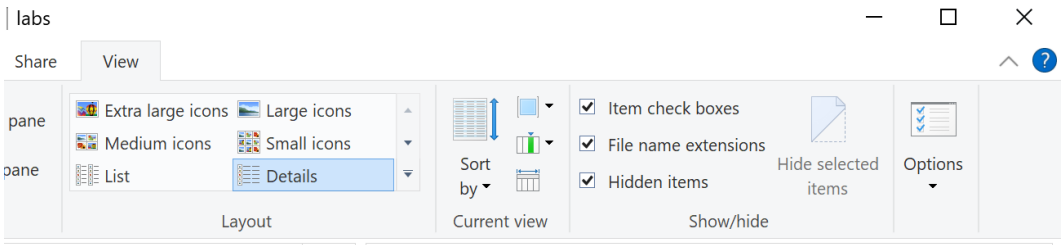
Approach 2:
Right click on the directory and select `Git Bash Here` from the context menu to open the Bash shell.



If you are working with Linux / MacOS, you will type your Git commands into the standard shell that you typically use for your Linux distro / MacOS. You will need to use the **cd** command to change to the designated folder that you wish to initialize the Git repository in.

The Git Bash shell prompt should now look some thing like this:

If you are using Windows, make sure that you have checked the File name extensions and Hidden items checkboxes in your File Explorer in order for us to be able to directly manipulate the file extensions as well as view the hidden `.git` folder that will be created in a later lab session.



In the lab sessions to follow, the Git commands to type are listed after the $ prompt and their output (or relevant parts of their output) will be shown as well. For this workshop, it will be easier and less prone to error to simply copy and paste these commands from this manual into the Git Bash shell.

3 Basic Configuration

Git has a wide variety of configuration options that allow us to customize how it performs specific commands and operations. The configuration can be done at 3 different levels and the configuration information for these 3 levels are stored in different corresponding configuration files in different locations:

Level	Scope of application	Location
System (`--system`)	To every user in the system and their repos	Windows: `C:\Program Files\Git\etc\gitconfig` Linux: `~/etc/gitconfig`
Global (`--global`)	To the current logged in user	Windows: `C:\Users\`*`username`*`\.gitconfig` Linux: `~/home/`*`username`*`/.gitconfig`
Local (`--local`)	To a single specific repo	Windows and Linux: `.git\config` in the working directory

For the majority of cases, we will typically set configuration values at the global level. If a particular configuration variable is set with different values at two or more different levels, the value at the most specific level takes effect. For e.g. local is more specific than global which is in turn more specific than system.

There some basic values that need to be set before we start working on a Git project.
We will need to set the username and email address because every Git commit that you create will include this information. In addition, we will enable coloring to make the command outputs easier to view and interpret.

Set these configuration variable values using these commands. **NOTE:** If you already have a BitBucket account which you are regularly interacting with from the Git Bash shell or Git GUI client, the `user.name` and `user.email` values will already have been set and you can skip the commands to do that below.

```
$ git config --global user.name "Peter Parker"
$ git config --global user.email "spiderman@gmail.com"
$ git config --global color.ui auto
```

After you have done this open the corresponding configuration file at the global level to verify that the values have been set correctly.

To obtain the values for a particular configuration variable, use `git config` followed by the configuration variable, for e.g.

```
$ git config user.name
Peter Parker

$ git config user.email
spiderman@gmail.com
```

In addition to user name and email, you may also wish to configure the default editor that Git uses in order to create commit messages when these are not explicitly specified. To check on the current configured editor, type:

```
$ git config core.editor
```

On Windows, if you have already installed Visual Studio Code, this is most likely to be the default configured editor, in which case you will see a setting similar to this:

```
'C:\Users\UserAccount\AppData\Local\Programs\Microsoft VS
Code\Code.exe' --wait
```

Here, *UserAccount* will be the name of the current user account that you are logged into on your Windows machine.
If you do not see this, or if you see another editor configured and you wish to use VS Code instead, you can configure it in a similar way with the user name and email (make sure that the entire command below is entered on a single line):

```
$ git config --global core.editor
 "'C:\Users\UserAccount\AppData\Local\Programs\Microsoft VS
Code\Code.exe' --wait"
```

Remember, *UserAccount* will be the name of the current user account that you are logged into on your Windows machine.

NOTE: At the moment, there is no support for setting Visual Studio as the default editor: the only support is for Visual Studio Code.

If you wish to use NotePad++ instead as the default editor, then type (make sure that the entire command below is entered on a single line):

```
$ git config --global core.editor
"'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar
-nosession -noPlugin"
```

On Linux/MacOS, you should see either Vim or Emacs as the default configured editor. If you wish to use another editor (for e.g. Nano), then type:

```
$ git config --global core.editor "nano"
```

To view the complete list of all configuration variables, their values as well as the config files that they are found in, type:

```
$ git config --list --show-origin
```

A long list will appear. You can use the arrow keys to scroll up and down through the list and press Q to exit the list scrolling. This will be the case as well for any other Git command that produces a long list of output that cannot be displayed on a single screen.

Alternatively, you can redirect the console output to a file for later viewing/editing.

```
$ git config --list --show-origin > configuration.txt
```

Open `configuration.txt` to verify and delete it when done.

4 Creating initial directory content

We will assume that this directory (`firstlab`) is the root folder of your project. In a real software project, this will contain source code files along and other relevant build artifacts. In this and the subsequent labs, we will use simple text files to make it easier to illustrate the operation of the various Git commands but keep in mind that Git is typically used to perform version control for a software project which involves source code files in standard programming languages like C#, Java, Python, C++, etc.

Create 3 files named as below and populate them with a single line each as shown using a text editor. Make sure you include a new line after the end of the single line

```
1: developer
```

`humans.txt`

```
1: cat
```

`animals.txt`

```
1: honda
```

`cars.txt`

5 Initializing a local Git repository

There are two ways to create a Git repo:

a) Initialize a local repo in the root folder of your project. This root folder and its contents now become the working directory or working tree for the Git project.

b) Clone a local repo from the contents of a remote repo

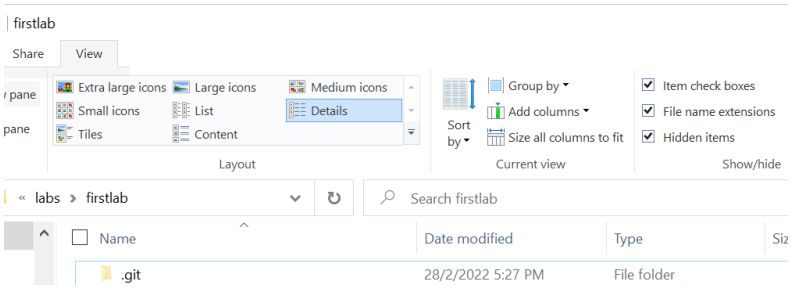
We will proceed with the first approach a). Make sure that your Git Bash shell is open in `firstlab` (this will now be considered the root folder of your project) before proceeding


```
$ git init

Initialized empty Git repository in G:/labs/firstlab/.git/

User@computer MINGW64 /g/labs/firstlab (master)
```

The Git repo is stored in the `.git` folder in the root folder of your project. This is a hidden folder by default to protect it from accidental manipulation. To able to view and access this folder in Windows, you will need to check the hidden items check box in the View tab of the File Explorer.



You should not manually manipulate the contents of this folder by yourself unless you are very sure about what you are trying to do: making an error in here could potentially damage your repo.

Notice that the name of the prompt in the Git Bash shell now includes `master` at the end. This is the default branch for every new locally initialized Git project. Remember that a branch is a pointer to a commit. At the moment, we do not have any commits yet in the project.

A project should **ONLY** have one `.git` folder (which holds the Git repository) and this should be located in the root folder of the project (right now this is `firstlab`). This means that any subdirectories with the root project folder (for e.g. `firstlab/src`, `firstlab/dist`, `firstlab/lib`, etc) cannot have a `.git` folder: you should therefore **NEVER RUN** the `git init` command again in a subdirectory of the root folder that already has a `.git` folder. It is possible to have Git repositories nested within other Git repositories (this is known as submodules), but this is an advanced concept. For now, always ensure that all your projects have only one `.git` folder which is always in the root folder of that project.

Since the Git repository holds all the information related to version control for the project, if it is deleted for whatever reason, you will lose all the versioning information related to that project. Also, if you accidentally move the `.git` folder to another location after its creation, all the versioning control information it holds becomes invalid. The `.git` folder will create versioning information that is specific to its relative location in the project structure, which is in the root folder of the project. The sensitivity of the `.git` folder is the reason why it is made a hidden file by default, and the only reason you are able to see it is because you have enabled your File Explorer to view hidden files.

6 Staging and committing changes

To check the status of the project, type:

```
$ git status
On branch master

No commits yet

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

```
        animals.txt
        cars.txt
        humans.txt

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

The list of your original 3 files are highlighted in red and categorized as untracked files. Remember that all files in your working directory are either tracked or untracked. Newly added files in the working directory are initially considered untracked. To change these files to being tracked, we can stage them. Lets stage one file first:

```
$ git add animals.txt
$ git status

On branch master

No commits yet

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

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

Notice now that the specified file has been highlighted in green and listed under changes to be committed.

We can use a wildcard pattern to stage the two other remaining files:

```
$ git add *.txt
$ git status

On branch master

No commits yet

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

Before we create out first commit in the project to contain all these changes, we will create a `.gitignore` file that lists all the files that we explicitly want Git to ignore during the course of the project. This means Git will explicitly not track these files or any changes that happen in them throughout the duration of the project.

For e.g. if you are working with a Mac, you probably want to ignore all the `.DS_Store` files (these are systems files created by the Finder app). For a software project, this could be automatically generated files such as log files or artifacts produced by your build system such as binary executables.

Create a file name `.gitignore` and fill it with a two file listing patterns: the first to match all files that end with the extension `obj` and the second to tell Git to ignore the `.gitignore` file as well.

```
*.obj
.gitignore
```

We can now commit all the staged changes with:

```
$ git commit -m "Yay ! My first commit"
[master (root-commit) 1867882] Yay ! My first commit
 3 files changed, 3 insertions(+)
 create mode 100644 animals.txt
 create mode 100644 cars.txt
 create mode 100644 humans.txt
```

All commits must have a message associated with it, and here we specify it after the option `-m`.
The `root-commit` term indicates that this is the first commit in the project. The number sequence `1867882` is the first couple of digits of the 40-character SHA-1 hash of this newly created commit.

The SHA-1 hash value will be different for your example, since it is computed on the commit contents which include metadata such as author, date of creation, etc. Even if you are using the exact same username as me, the date of creation of this commit will be different for you.

The numbers `100644` after `create mode` are Linux file permission sequences, and here they indicate that these 3 files are normal files.

7 Tracking changes

To view the commit history, type:

```
$ git log
commit 18678822f5077682573f2dc04ba558398a4612c1 (HEAD -> master)
Author: Peter Parker <spiderman@gmail.com>
Date:   Wed Jul 7 17:31:13 2021 +0800

    Yay ! My first commit
```

This command shows the commit history starting from the commit pointed to by HEAD all the way back to the root commit. Here, we only have one commit (the root commit) which has a 40-character SHA-1 hash that is used to identify it (`1867…2c1`). It includes the author and email values that we configured earlier as well as the time stamp for commit creation and the commit message we specified.

The `(HEAD -> master)` indicates that the `master` branch is pointing to this commit and HEAD is pointing to `master`. Typically, a branch pointer will always point to the most recent commit in

sequence of commits, while the HEAD pointer will always point to a branch pointer. Right now, we only have one branch pointer so HEAD will always point to it.

Let's add in two more files in the working directory with the following contents:

```
some stuff here
```

program.obj

```
just some changes
```

useless.txt

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        useless.txt

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

Notice now that `program.obj` does not appear in the list of untracked files even though it is a newly created file. This is because it matches the `*.obj` file listing in `.gitignore`, as mentioned earlier.

Make some random changes to `useless.txt` and type:

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        useless.txt

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

Notice that no matter how many changes are made and saved to `useless.txt` it still shows up as untracked in the status output. We can stage all the changes to this file and commit it:

```
$ git add useless.txt
$ git commit -m "Added the useless file"

[master 4d6b121] Added the useless file
 1 file changed, 8 insertions(+)
 create mode 100644 useless.txt
```

We check the commit history again with:

```
$ git log
commit 4d6b121d17ed96d134a40adf7f682c70e86df299 (HEAD -> master)
Author: Peter Parker <spiderman@gmail.com>
Date:   Wed Jul 7 19:53:29 2021 +0800
```

```
    Added the useless file

commit 18678822f5077682573f2dc04ba558398a4612c1
Author: Peter Parker <spiderman@gmail.com>
Date:   Wed Jul 7 17:31:13 2021 +0800

    Yay ! My first commit
```

Notice we now have two commits and both `HEAD` and `master` have been advanced to point to the latest commit. The commit history is listed in reverse chronological order with the most recent commit first followed by the ancestor commits all the way back to the root commit. Again, each commit is shown with its SHA-1 hash, the author's name and email, the date / time of its creation and the commit message

Add these additional lines to the end of the following 3 files and save, ensuring that you include a new line at the end:

```
2: project manager
```

humans.txt

```
2: dog
```

animals.txt

```
2: mercedes
```

cars.txt

As usual, we verify that the files have been modified but not yet staged with:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
        modified:   animals.txt
        modified:   cars.txt
        modified:   humans.txt

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

So far, we have staged the files with `git add` and committed the staged changes with `git commit`. We can use a shortcut to both stage and commit changes in the modified files with a single command:

```
$ git commit -am "Added a second line to animals, cars and humans"
[master 60aea14] Added a second line to animals, cars and humans
 3 files changed, 5 insertions(+)
```

We check the status again with:

```
$ git status
On branch master
nothing to commit, working tree clean
```

The term `working tree clean` indicates that there is neither staged or unstaged changes at this current point in the working directory.

8 Deleting and renaming tracked files

We can delete a file that has already been staged and committed if we wish to. However, it has to be done via a Git command instead of just directly deleting the file using the file manager utility of the OS. Let's do that now for `useless.txt`

```
$ git rm useless.txt
rm 'useless.txt'
```

If you check the project directory or the text editor where `useless.txt` is open in or, you will notice that it is now deleted.

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    useless.txt
```

Notice as well that `useless.txt` has been staged for deletion. The deletion of a tracked file is considered a change that needs to be committed, just like addition of new files or modification of existing files.

We will remove it with a suitable commit message:

```
$ git commit -m "Removed useless"
[master b26f5b5] Removed useless
 1 file changed, 6 deletions(-)
 delete mode 100644 useless.txt
```

We can check the commit history again with:

```
$ git log
commit b26f5b5ef121a1f2f52b89b42383fc62ed836341 (HEAD -> master)
Author: Peter Parker <spiderman@gmail.com>
Date:   Sun Feb 27 21:32:35 2022 +0800

    Removed useless

commit 60aea14b44e0c92b57e3cacb3f5ace9de8d5e404
Author: Peter Parker <spiderman@gmail.com>
Date:   Sun Feb 27 21:11:03 2022 +0800

    Added a second line to animals, cars and humans
….
….
….
```

If the listing of commits does not fit within the display area of the Git bash terminal, you can use the up and down arrow keys to scroll through the listing and also press the Q key to escape and return to the shell.

Occasionally during a development workflow, you may need to rename existing files. This operation will also need to be staged and committed in the same manner as a removal.
Let's rename `humans.txt` to `people.txt` with:

```
$ git mv humans.txt people.txt
```

Then check the status in the usual manner with:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    humans.txt -> people.txt
```

Finally, commit these changes with a message:

```
$ git commit -m "Renamed humans to people"
[master 438a38c] Renamed humans to people
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename humans.txt => people.txt (100%)
```

The 100% above is a comparison difference that Git runs in the background whenever a `git diff` (to be covered in upcoming lab) or `git mv` operation has been executed. Here, we are only performing a rename operation, which means that the content of `people.txt` (new file) is 100% similar to `humans.txt` (old file).

Let's restore back the file name to `humans.txt` and at the same time also add in a new line to the 3 text files. First, we will add in the new line to the 3 existing files, making sure to include a newline at the end:

```
3: CEO
```

people.txt

```
3: elephant
```

animals.txt

```
3: proton
```

cars.txt

Next, we rename the file `people.txt` back to the original name of `humans.txt`

```
$ git mv people.txt humans.txt
```

Then check the status in the usual manner with:

```
$ git status
On branch master
```

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    people.txt -> humans.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
        modified:   animals.txt
        modified:   cars.txt
        modified:   humans.txt
```

Here, the file modifications are listed as unstaged changes, while the file renaming operation is already staged. You can make as many types of modifications as you want before creating a proper commit: this includes adding new files, modifying, renaming or deleting existing files. It is not necessary to commit the modifications in stages: i.e. perform modification changes then commit, perform deletion changes then commit, etc, etc. The key point is that the commit should represent a meaningful milestone in the evolution of the code base.

Let's create a new commit again that includes all our recent changes with:

```
$ git commit -am "Renamed people to humans and added 3rd line to all
files"
[master a8fe9ef] Renamed people to humans and added 3rd line to all
files
3 files changed, 4 insertions(+)
rename people.txt => humans.txt (82%)
```

Here, we see there is similarity of 82% (instead of 100% as previously) because we made a slight change to the content of `people.txt` before performing the rename to `humans.txt`