**001. Git Basics**

Git is different from all other VCS in its way it operates. Nearly every operation in Git needs only local files and resources. For example, to browse the history of the project, Git doesn't need to go to the server to get the history and display it for you – it simply reads it directly from your local database. There really is very little you can't do even if you're offline, like you go on vacation or train and you want to do a little work, you can commit happily (to your local repository, remember?) until you get to a network connection to upload. If you are home and the internet is down or you can't get the VPN client working properly, you can still work. In many other systems, doing so is either impossible or painful.

Everything in Git is check summed before it is stored and is then referred to by that checksum. The exact mechanism produces SHA-1 hash, which is a 40-character string, composed of hexadecimal characters (0-9 and a-f) and calculated based on the contents of a file or directory, something like this 32c9sa3843212834ab493b52f8696cd6d3b00373 . This means that it's impossible to change the contents of any file or directory without Git knowing about it, because the hash will be different if there was any change.

When you do actions in Git, nearly all of them only add data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. As with any VCS, you can lose or mess up changes you haven't committed yet, but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

Pay attention now – here is the main thing to remember about Git if you want the rest of your learning process to go smoothly.

**Git File Status**

Git has 4 states that your files can reside in: **untracked**, **unmodified**, **modified**, **staged**. However, while the first status is **untracked,** all others are **tracked**.

1. **untracked** any files in your directory that git doesn't know about. They were not in your last snapshot and are not in your staging area

2. **unmodified** any files, who were in your last snapshot, but you haven't edited them. Files in this status are safely stored in your local Git database.

3. **modified** means that you have changed the file, but you have not committed them to your Git database yet

4. **staged** means that you have marked a modified file in its current version to go into your next commit snapshot

Depending on what state a particular file is, it will usually reside in 1 of the 3 main areas of a Git managed directory.

**The 3 Main Areas of a Git Project**

**Working Directory**

**Staging Area (Index)**

**.git Directory**

add

commit

checkout

The 3 main areas of a Git managed directory: **Working Directory**, **Staging Area** and **.git Directory (also called Repository)**

1. **Working Directory:** All files in a folder, in our filesystem, which folder is managed by Git. Git also refer to this directory as a working tree of files. The working tree is a single checkout of one version of the project, so any files in this directory are pulled out of the compressed database in the **.git Directory** and placed on disk for you to use and modify.

2. **Staging Area:** A file, generally contained in your **.git Directory**, that stores information about what will go into your next commit. Its technical name is "index", but the phrase "staging area" works just as well.

3. **.git Directory:** is where Git stores the metadata and object database for all your files. This is the most important part of Git, and it is what is copied when you clone a repository from another computer, or from a repository hosting platform like GitHub.



index.html



styles.css



assets



project

**Working Directory**



project



assets



styles.css

index.html



.git

**git init**



**For example:** if we create a new folder named "project", which will be for our web application, it will contain our files with code, it will be a simple folder in our filesystem and it will also not be under any version control currently. In order to start controlling it with Git, we need to turn this folder into a **Working Directory** managed by Git. This is done by running the command **git init** at the root level of that folder. This will create the hidden **.git Directory**, which is technically called a **Repository** and is where Git stores all information about your files. At this point your normal directory is now a **Working Directory** Git knows about and all files in it, if there were any to begin with, will initially enter the **untracked** status.

**Staging Area**



styles.css

index.html



assets

**git add**



**Working Directory**



styles.css

index.html



assets



.git

Then if we want to save this first version of our project, meaning all files inside the **Working Directory** to be stored as our first copy inside the Git database, which is inside the **.git Directory (Repository)**, in Git we can do this by committing a snapshot. You can think of this snapshot as a copy of our files in the end, so the first commit we create right here could be "snapshot1". This first snapshot will contain all files and folders we commit, but in order to commit, we must also change all status of the files we want to commit from **untracked** to **staged,** since Git will only commit tracked files who are also in the **Staging Area**. By running the command git add we move the files into the **Staging Area** and change their status to **staged**. Next by running the command git commit we save the files into the database inside the **.git Directory (Repository)** and change their status to **unmodified.**

Now it is important to realize that our commits, or saved data is a series of snapshots, because Git is only tracking changes, NOT storing files again and again! Because imagine if it was storing a full copy of the files every time, how quickly would the directory size increase. That is NOT to say that it doesn't store any files at all, but when it does, it stores them only if they change and then passes a reference to that file if it is in that snapshot. That's an oversimplified explanation, because the storing logic is of course quite complex, but we can think of Git more like a mini filesystem with some incredibly powerful tools build on top of it.

**Lab with Trainer**

git init

git status

git add

git commit

git log

git checkout

gitk

**The basic Git workflow goes like this:**

1. You add or modify files in your **Working Directory**.

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 save your work by doing a commit, which takes the files as they are in the **Staging Area** and stores that snapshot permanently to your database inside the **.git Directory**

**Step. 1 Initialize a Git Repository**

We typically obtain a Git repository in one of two ways:

1. we can take a local directory that is currently not under version control, and turn it into Git repository, or

2. we can clone an existing Git repository from elsewhere

Let's begin with a local directory example. Navigate to a place in your filesystem of your choosing and create a folder named "**project**", then open the folder with **GitBash** terminal and run the command git init in order to start controlling it with Git. We can confirm the status of our **Working Directory**, the **Staging Area** and also the state of our files by running the command git status. At this point, nothing in your project is tracked yet.

Run the command git status and you should see something like this:

|  |
| --- |
| $ git status  On branch main  No commits yet  nothing to commit (create/copy files and use "git add" to track) |

This means we have a clean working directory; in other words, none of our tracked files are modified. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tells us which branch we're on and informs us that it has not diverged from the same branch on the server. For now, that branch is always **"main"**, which is the default; You won't worry about it here. We will go over branches later.

Now let's add some files (We can re-create the files and folders from the resources folder)

Run the command git status again. The files we just added didn't exist before, so we see that our new files are **"untracked"**, which basically means that Git sees a file we didn't have in the previous snapshot (commit), and which hasn't yet been staged:

|  |
| --- |
| $ git status  On branch main  No commits yet  Untracked files:  (use "git add <file>..." to include in what will be committed)  assets/  index.html  style.css  nothing added to commit but untracked files present (use "git add" to track) |

**Step 2. Selectively stage** **just those changes you want to be part of your next commit.**

Run the command git add index.html style.css assets/ or

As a shorthand run git add .

We can run the command git status again, to confirm that our files are staged.

|  |
| --- |
| $ git status  On branch main  No commits yet  Changes to be committed:  (use "git rm --cached <file>..." to unstage)  new file: assets/logo.png  new file: index.html  new file: style.css |

Let's run the command git log to see the history. At that point we should see an empty log

|  |
| --- |
| $ git log  fatal: your current branch "main" does not have any commits yet |

**Step 3. Save your work by doing a commit.**

Now, if we commit at this point, the version of the file at the time we ran git add is what will be in the subsequent historical snapshot.

Run the command git commit -m "initial commit" and we should see a success message. We have now successfully saved our work inside the **Objects Database** in Git and our files status have been changed from **staging** to **unmodified**.

Now, if we run the command git status , we would see "nothing to commit, working tree clean". So that's nice, but where is our commit now? Well, we have a command in Git which gives us access to all our commits in our current branch, which is currently **main**. The command I'm referring to is git log , so let's do that.

$ git log

By default, with no arguments, the command lists all commits made in that branch, in reverse chronological order, that is, the most recent commits show up first. We get some information about each commit with it's SHA-1 checksum, the author's name and email, the date written and the commit message. Each created commit has this unique id, by which we can always refer to this specific snapshot at that moment of history and for example load all files as they were back then. This process is technically called "**checkout**", but the word "loading" works just as well. Pay attention here, this "**checkout**" action is very important and we will revisit it several times until it clicks, so now for example, we have just saved our work from our **Working Directory** to the **.git Repository** and we said that the **Working Directory** is a single checkout of one version of the project. Well now we have one version, it's only one commit, but the files of this commit are safely stored in the **.git Repository** inside something called an **Objects Database** and what we have is simply just a reference.

Let's prove this, delete all our files in the **Working Directory** and then run the command git checkout . The files are restored! They were simply just a reference.

Alright, we have created a Git Repository, wrote some code and made our first commit. Let's update our project with more code and then we will have a look at this once again.

**Step 4. Record new changes to the Repository**

Let's now create another file, **"script.js"** and run the command git status again. We should see something like this:

|  |
| --- |
| $ git status  On branch main  Untracked files:  (use "git add <file>..." to include in what will be commited)  scripts.js  nothing added to commit but untracked files present (use "git add" to track) |

Let's also modify and existing file, add the string "Hello World" inside the body tag of **"index.html"** and run the command git status again. We should see something like this:

|  |
| --- |
| $ git status  On branch main  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: index.html  Untracked files:  (use "git add <file>..." to include in what will be commited)  scripts.js  no changes added to commit (use "git add" and/or "git commit -a") |

Run the command git add . and run git status again. We have both added the **"index.html"** and **"scripts.js"** files to the **Staged Area,** so the **"index.html"** file status changed from **modified** to **staged** and the **"scripts.js"** file status changed from **untracked** to **staged**. We should see something like this:

|  |
| --- |
| $ git status  On branch main  Changes to be committed:  (use "git restore –staged <file>..." to unstage  modified: index.html  new file: scripts.js |

So once again we see files in which we have new changes and we can now commit them. Again, to commit changes we run thecommand git commit -m "adds x functionality" and we save our work.

Let's now dive again into the commits history, so run the command git log again