# Week 5

# Git (GitBASH) - Version Control Programming

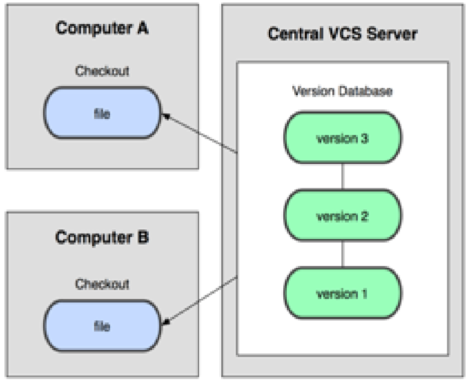
In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were as follows:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities. It's incredibly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development

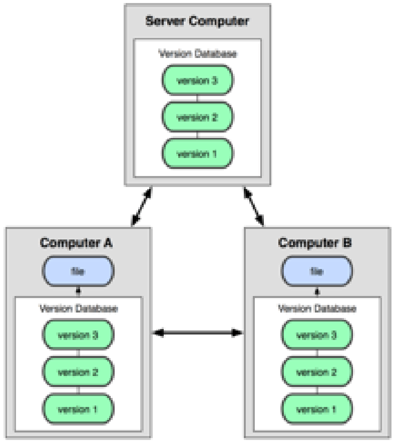
## Git uses a distributed model

Centralized Model



CVS, Subversion, Perforce

Distributed Model



(Git, Mercurial)

Result: Many operations are local

Git is a Version Control System (VCS), a tool which is an extremely smart choice to use even if it sounds too overwhelming. Simply put, a VCS is a group of files with monitored access. What does that mean? Let's consider a small example to help you understand.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a VCS is a very wise thing to use. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

## How does a VCS work?

So what exactly does a VCS do? Before we go any further, let's go ahead and understand some basic terminology.

You might come cross terms such as Source Control or Revision Control instead of Version Control, but these are interchangeable terms and keep in mind that they mean one and the same thing. However, we will be sticking to Version Control System for the remainder of this course.

A collection of all the files you and your team are working on is called a Repository. It also contains some special data such as the order in which the changes occurred, the description of the change and the person responsible for that particular change.

A change in your project might take hours, days or even weeks to finish. It would be unfair to treat unfinished work as a version. You are required to tell the VCS that you are done with the change and that you wish you save it as a version so that it can track it for you. The act of telling a VCS that a version is finished is called committing. Hence, versions are often referred to as commits.

VCS hides the complexity of multiple commits so that your project looks clean and tidy. It does this by storing all data in folders which are marked hidden so you don't realize that they are present on your computer until you need to interact with them.

But what if your repository is inside a hidden folder? This is the third major functionality of a VCS. It lets you view your project history. You can check how your project looked yesterday, a month before or even a year ago, often with a single command.

They also let you share your repository with your other team members so that collaboration is easy. Developers often work together on a single project and VCS just makes it easier for them.

## Why should I use Git?

Git was originally designed to help manage the Linux Kernel and make collaboration easy from the beginning. If Git can effectively manage a project as large as the Linux Kernel, it can manage your projects easily and effectively.

Furthermore, the architecture of Git is distributed Version Control as opposed to a centralized, network access VCS. A centralized VCS requires a network connection to work with and a central failure may result in all your work being destroyed. A distributed VCS such as Git, does not require a network connection to interact with the repository. Each developer has their own repository which makes it fast and easy to collaborate with.

Finally, on to GitHub. GitHub is easily the most popular website for sharing your projects with collaborators or with the whole world. It's like a social network for your projects. Repositories can be made public so that anyone can submit a commit and help make your project better.

Git uses checksums to secure your data. This makes it impossible to make changes to the data without Git getting a whiff of it. This functionality is built into Git at the lowest levels and is integral to its philosophy. The basic idea is that you can't lose information in transit or have files corrupted without Git being able to detect it.

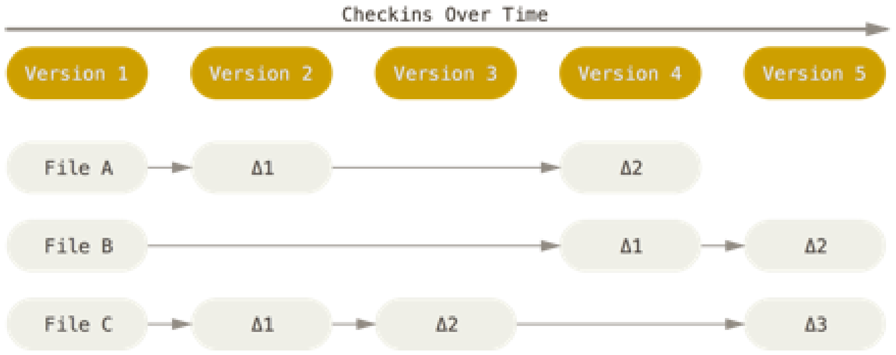
Git> uses the SHA-1 hash system which as you may probably know, is a 40- character string composed of hex characters and is calculated based on the content. Git extensively uses these values and stores files in database by this hash and not by the name.

GitHub has been gaining popularity in the software development world and it is likely that you would require to know Git if you are to be a collaborator on a project which is a reason on its own to learn more about Git.

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time.

## Git takes snapshots

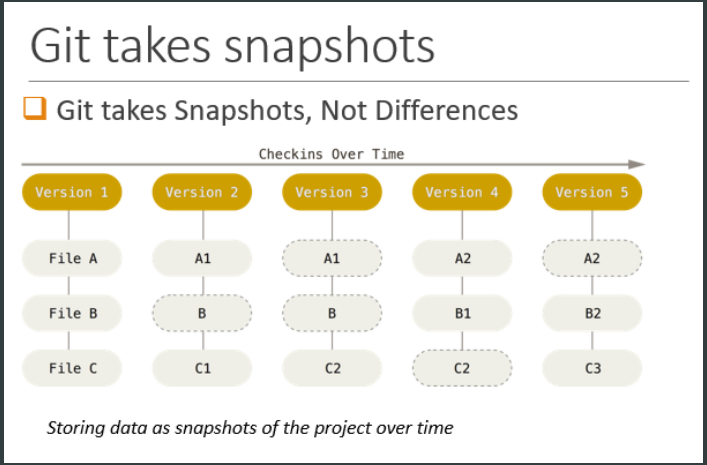
- Git takes Snapshots, Not Differences



*Storing data as changes to a base version of each file.*

Git does not think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in Git, it 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 does not 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.



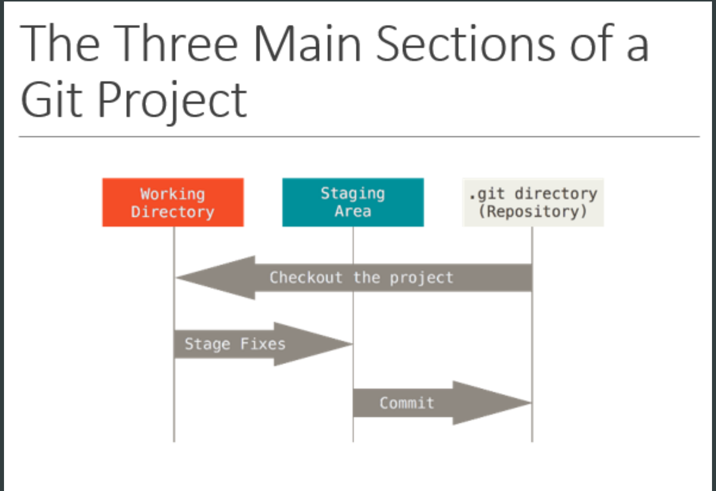
Git takes snapshots

❑ Git takes Snapshots, Not Differences

*Storing data as snapshots of the project over time*

**The Three States**

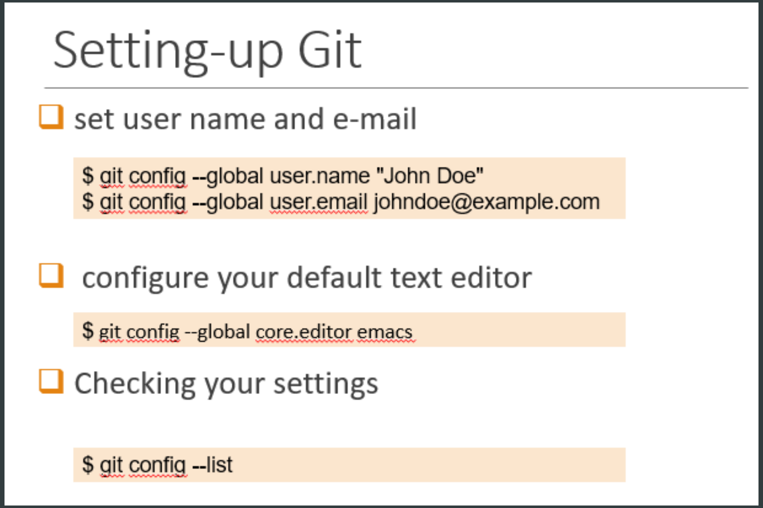
- Committed means that the data is safely stored in your local database.
- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.


The Three Main Sections of a Git Project
Working Directory
Staging Area
.git directory (Repository)
Checkout the project
Stage Fixes
Commit

*Working directory, staging area, and Git Directory*

**The Three Main Sections of a Git Project**

- Git directory is where Git stores the metadata and object database for your project.
- The working directory is a single checkout of one version of the project.
- The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit.



## Setting-up Git

- set user name and e-mail

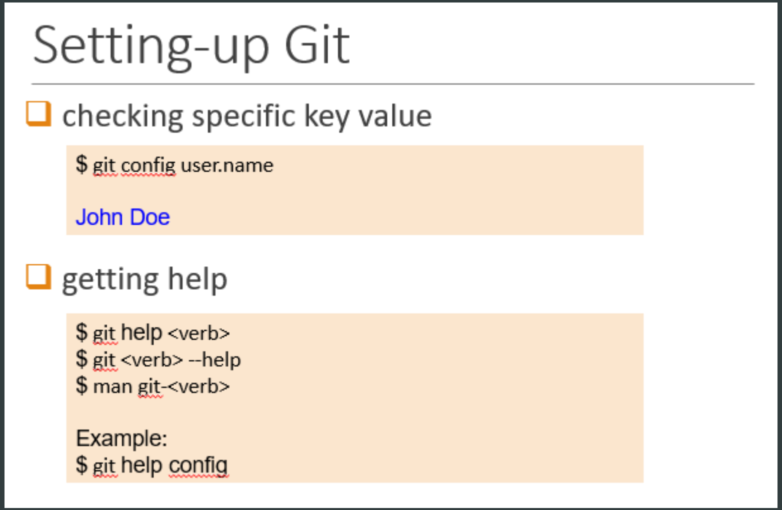
```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

- configure your default text editor

```
$ git config --global core.editor emacs
```

- Checking your settings

```
$ git config --list
```



## Setting-up Git

- checking specific key value

```
$ git config user.name

John Doe
```

- getting help

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>

Example:
$ git help config
```