



# Introduction to Version Control Systems

Facet Academy

<https://academy.facet.technology>

# Table of Contents

Introduction	3
Centralized Version Control System	3
Distributed Version Control System	4
Git – A popular DVCS	4
Nearly Every Operation Is Local in Git	6
Git Has Integrity	6
The Three States	6
Practical	8
First-Time Git Setup	8
Github Repository	9

# Introduction

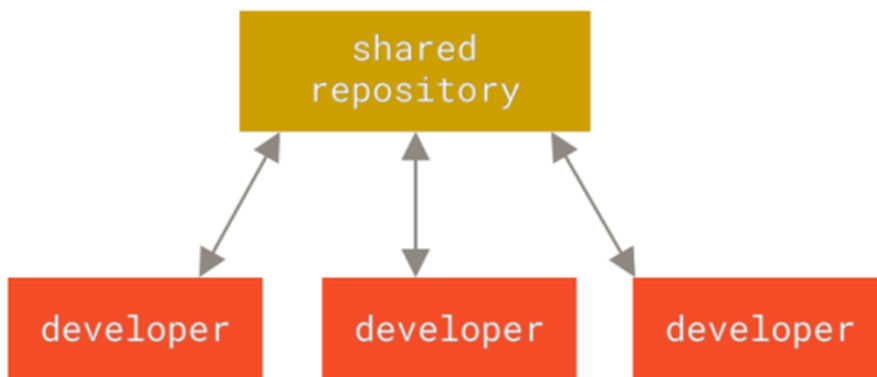
Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. You can do this with nearly any type of file on a computer.

VCS can be

- Centralized (CVCS)
- Distributed (DVCS)

## Centralized Version Control System

These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.



*Figure 1. Centralized Version Control System*

CVCS has some serious downsides. The most obvious is the single point of failure.

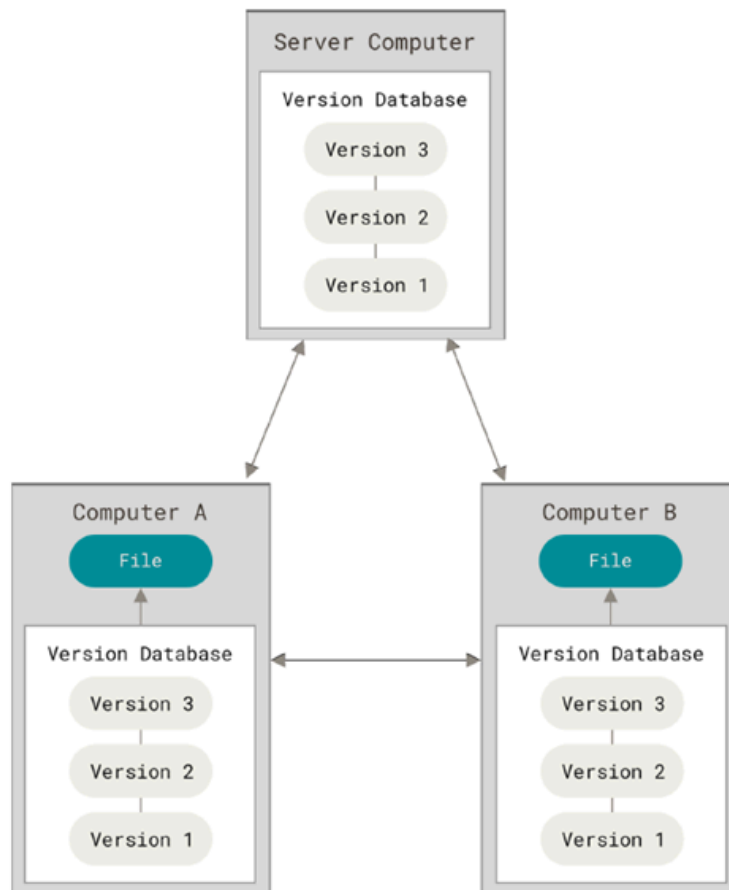
- If the server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they are working on.
- If the hard disk the central database is on becomes corrupted, and proper backups have not been kept, you lose absolutely everything — the entire history of the project.

# Distributed Version Control System

Distributed Version Control Systems (DVCSs) step in to address the issues with CVCS.

In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history.

Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

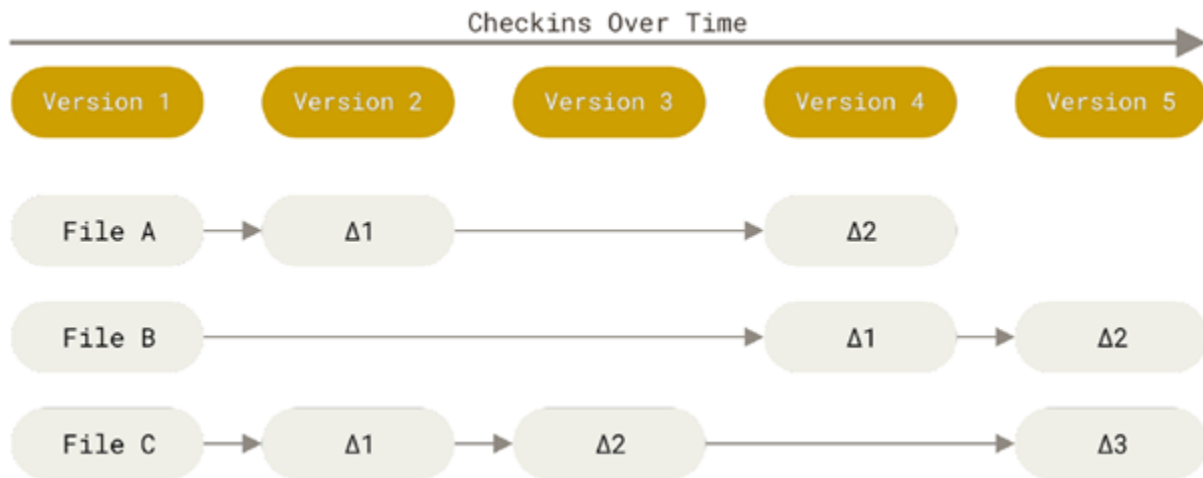


*Figure 2. Distributed VCS*

## Git – A popular DVCS

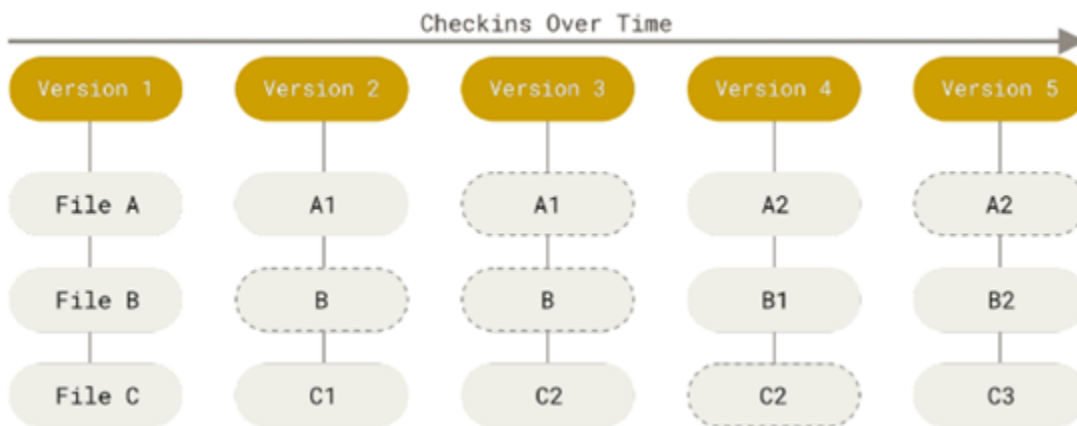
The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data.

Conceptually, most other systems store information as a list of file-based changes. These other systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they store as a set of files and the changes made to each file over time (this is commonly described as *delta-based version control*).



*Figure 3. Storing data as changes to a base version of each file*

Git does not store its data this way. Instead, Git thinks of its data more like a series of snapshots of a miniature filesystem. With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git 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.



*Figure 4. Storing data as snapshots of the project over time*

## Nearly Every Operation Is Local in Git

Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on your network. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

For example, to browse the history of the project, Git does not need to go out to the server to get the history and display it for you — it simply reads it directly from your local database. This means you see the project history almost instantly.

This also means that there is very little you can not do if you are offline. If you get on an airplane or a train and want to do a little work, you can commit happily (to your *local* copy) until you get to a network connection to upload.

## Git Has Integrity

Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it is impossible to change the contents of any file or directory without Git knowing about it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This 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 structure in Git.

A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Git stores everything in its database not by file name but by the hash value of its contents.

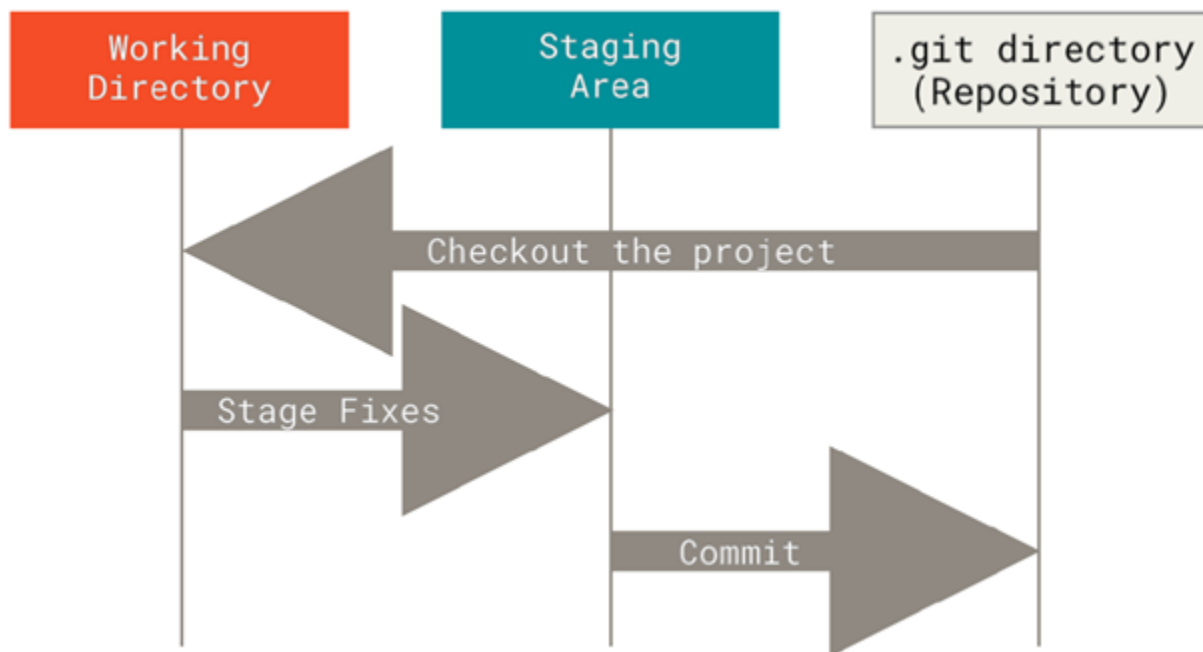
## The Three States

Git has three main states that your files can reside in:

- 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.

- Committed
  - means that the data is safely stored in your local database.

This leads us to the three main sections of a Git project: the working tree, the staging area, and the Git directory.



*Figure 5. Working tree, staging area, and Git directory*

The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit.

The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you *clone* a repository from another computer.

The basic Git workflow goes something like this:

1. You modify files in your working tree.
2. You selectively stage just those changes you want to be part of your next commit, which adds *only* those changes to the staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

## Practical

Install git by searching for instructions for your operating system in a search engine.

1. Confirm that Git is installed properly by running the following command. If you get a version number, you are good to go.

```
$ git --version  
git version 2.32.0
```

## First-Time Git Setup

2. The first thing you should do when you install Git is to set your user name and email address.

```
$ git config --global user.name "Himalaya Kakshapati"  
$ git config --global user.email "hk@gmail.com"
```

Replace the dummy name and email with your name and email address.

3. Set the default branch name as follows.

```
$ git config --global init.defaultBranch main
```

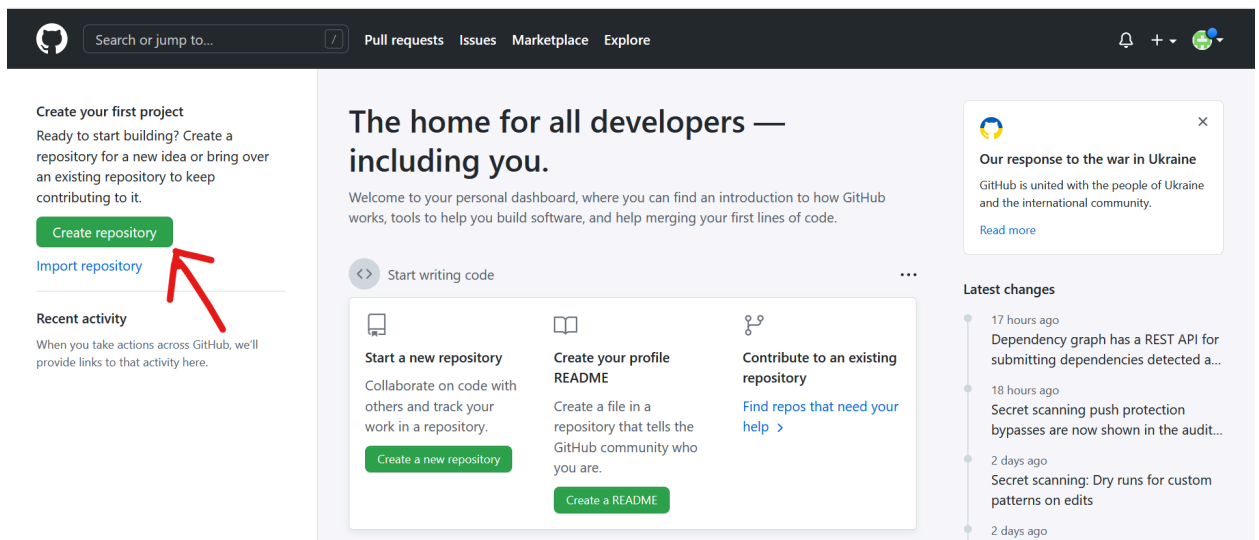
4. Check your settings as follows.



```
$ git config --list
user.email=hk@gmail.com
user.name=Himalaya Kakshapati
init.defaultbranch=main
```

## Github Repository

5. Sign up to Github website with your email at <https://github.com/>
6. After you sign up, sign in, and click on the “Create repository” button on the dashboard.




7. Enter the name of your new repository in the form and click on the “Create repository” button at the bottom of the page.

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*

 kakshapatih ▾

Repository name \*

/ TestRepo ✓



Great repository names are short and memorable. Need inspiration? How about [congenial-eureka?](#)

Description (optional)

☒  Public

Anyone on the internet can see this repository. You choose who can commit.

☐  Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

☐ Add a README file

This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore

Choose which files not to track from a list of templates. [Learn more.](#)

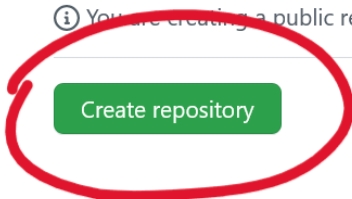
.gitignore template: None ▾

Choose a license

A license tells others what they can and can't do with your code. [Learn more.](#)


License: None ▾


 You are creating a public repository in your personal account.



Create repository

8. Follow the instructions on the resulting page.

 Search or jump to... Pull requests Issues Marketplace Explore

 kakshapatih / TestRepo Public

Pin Unwatch 1 Fork 0 Star 0

<> Code Issues Pull requests Actions Projects Wiki Security Insights Settings

### Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

### ...or create a new repository on the command line

```
echo "# TestRepo" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/kakshapatih/TestRepo.git
git push -u origin main
```


### ...or push an existing repository from the command line


```
git remote add origin https://github.com/kakshapatih/TestRepo.git
git branch -M main
git push -u origin main
```

### ...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

 **ProTip!** Use the URL for this page when adding GitHub as a remote.

 © 2022 GitHub, Inc. [Terms](#) [Privacy](#) [Security](#) [Status](#) [Docs](#) [Contact GitHub](#) [Pricing](#) [API](#) [Training](#) [Blog](#) [About](#)

<https://www.kdnuggets.com/2022/06/14-essential-git-commands-data-scientists.html>