

# Git and GitHub

👋 Welcome to Version Control!

*(A bonus resource is given at the end of the article)*

Here are the basics:

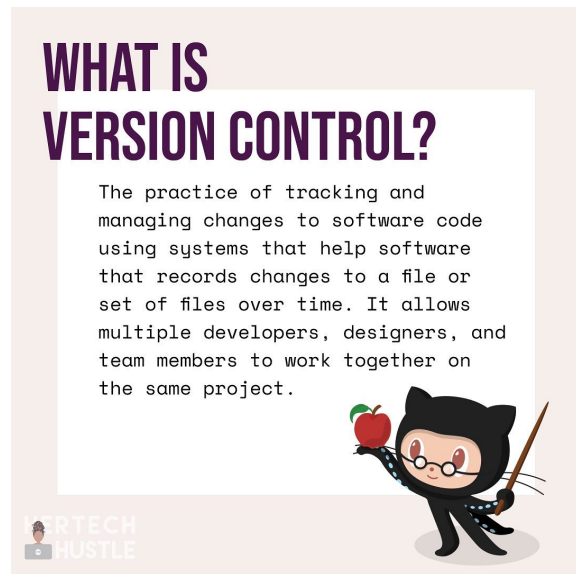


Image by [hertechhustle](https://hertechhustle.com)

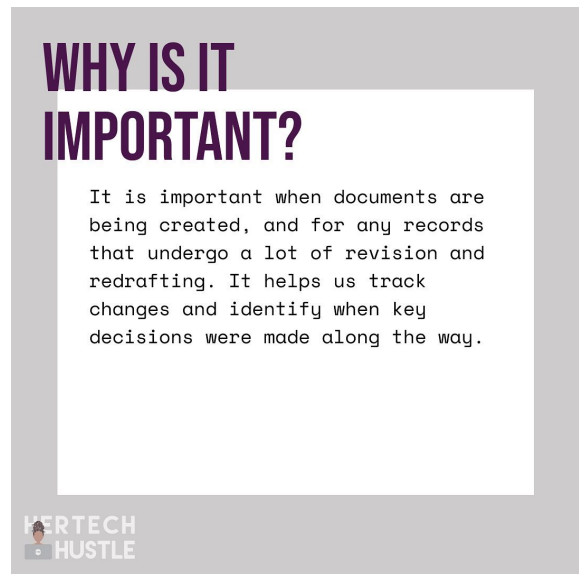
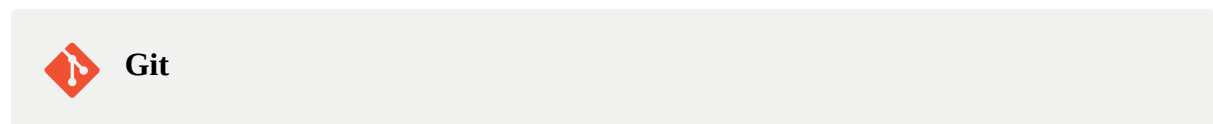


Image by [hertechhustle](https://hertechhustle.com)



## Just before we start..

How is Git different from other VCS (Version Control Systems)? Probably the most obvious difference is that Git is distributed (unlike SVN or TFS for instance). This means, you'll have a local repository which lives inside a special folder named `.git` and you'll normally (but not necessarily) have a remote, central repository where different collaborators may contribute their code. Note that each of those contributors has an **exact clone** of the repository on their local workstation.

Git itself can be imagined as something that sits on top of your file system and manipulates files. Even better, you can imagine Git as a **tree** structure where **each commit creates a new node** in that tree. Nearly all Git commands actually serve to navigate on this tree and to

manipulate it accordingly. As such in this tutorial I'd like to take a look at how Git works by viewing a Git repository from the point of view of the tree it constructs. To do so I walk through some common use cases like

- adding/modifying a new file
- creating and merging a branch with and without merge conflicts
- Viewing the history/changelog
- Performing a rollback to a certain commit
- Sharing/synching your code to a remote/central repository

Before starting here, I highly recommend to first go through the initial pages of the [Git Reference Manual](#), especially the ["Getting Started - Git Basics"](#) part.

## Terminology

Here's the git terminology:

- **master** - the repository's main branch. Depending on the work flow it is the one people work on or the one where the integration happens
- **clone** - copies an existing git repository, normally from some remote location to your local environment.
- **commit** - submitting files to the repository (the local one); in other VCS it is often referred to as "checkin"
- **fetch or pull** - is like "update" or "get latest" in other VCS. The difference between fetch and pull is that pull combines both, fetching the latest code from a remote repo as well as performs the merging.
- **push** - is used to submit the code to a remote repository
- **remote** - these are "remote" locations of your repository, normally on some central server.
- **SHA** - every commit or node in the Git tree is identified by a unique SHA key. You can use them in various commands in order to manipulate a specific node.
- **head** - is a reference to the node to which our working space of the repository currently points.
- **branch** - is just like in other VCS with the difference that a branch in Git is actually nothing more special than a particular label on a given node. It is not a physical copy of the files as in other popular VCS.

# Workstation Setup

To set up command line Git access simply go to [git-scm.com/downloads](https://git-scm.com/downloads) where you'll find the required downloads for your OS. More detailed information can be found [here as well](#).

After everything is set up and you have "git" in your PATH environment variable, then the first thing you have to do is to config git with your name and email:

```
$ git config --global user.name "Juri Strumpflohner"
$ git config --global user.email "myemail@gmail.com"
```

## Let's get started: Create a new Git Repository

Before starting, lets create a new directory where the git repository will live and `cd` into it:

```
$ mkdir mygitrepo
$ cd mygitrepo
```

Now we're ready to initialize a brand new git repository.

```
$ git init
Initialized empty Git repository in c:/projects/mystuff/temprepos/mygitrepo/.git/
```

We can check for the current status of the git repository by using

```
$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

## Create and commit a new file

The next step is to create a new file and add some content to it.

```
$ touch hallo.txt
$ echo Hello, world! > hallo.txt
```

Again, checking for the status now reveals the following

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       hallo.txt
nothing added to commit but untracked files present (use "git add" to track)
```

To “**register**” the file for committing we need to **add** it to git using

```
$ git add hallo.txt
```

Checking for the status now indicates that the file is ready to be committed:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   hallo.txt
#
```

We can now **commit** it to the repository

```
$ git commit -m "Add my first file"
1 file changed, 1 insertion(+)
create mode 100644 hallo.txt
```

*It is common practice to use the “presence” in commit messages. So rather than writing “added my first file” we write “add my first file”.*

## Add another file

Lets add another file:

```
$ echo "Hi, I'm another file" > anotherfile.txt
$ git add .
$ git commit -m "add another file with some other content"
1 file changed, 1 insertion(+)
create mode 100644 anotherfile.txt
```

Btw, note that this time I used `git add .` which adds all files in the current directory (`.`).

From the point of view of the tree we now have another node and master has moved on to that one.

## Create a (feature)branch

Branching and merging is what makes Git so powerful and for what it has been optimized, being a distributed version control system (VCS). Indeed, **feature branches** are quite popular to be used with Git. Feature branches are created for every new kind of functionality you're going to add to your system and they are normally deleted afterwards once the feature is merged back into the main integration branch (normally the master branch). The advantage is that you can experiment with new functionality in a separated, isolated “playground” and quickly switch back and forth to the original “master” branch when needed. Moreover, it can be easily discarded again (in case it is not needed) by simply dropping the feature branch. There's a nice article on [understanding branches in Git](#) which you should definitely read.

But lets get started. First of all I create the new feature branch:

```
$ git branch my-feature-branch
```

Executing

```
$ git branch
* master
  my-feature-branch
```

we get a list of branches. The `*` in front of `master` indicates that we're currently on that branch. Lets switch to `my-feature-branch` instead:

```
$ git checkout my-feature-branch
Switched to branch 'my-feature-branch'
```

Again

```
$ git branch
master
* my-feature-branch
```

**Note** you can directly use the command `git checkout -b my-feature-branch` to create and checkout a new branch in one step.

What's different to other VCS is that there is only *one working directory*. All of your branches live in the same one and there is not a separate folder for each branch you create. Instead, when you switch between branches, Git will replace the content of your working directory to reflect the one in the branch you're switching to.

Lets modify one of our existing files

```
$ echo "Hi" >> hallo.txt
$ cat hallo.txt
Hello, world!
Hi
```

...and then commit it to our new branch

```
$ git commit -a -m "modify file adding hi"
2fa266a] modify file adding hi
1 file changed, 1 insertion(+)
```

**Note**, this time I used the `git commit -a -m` to add and commit a modification in one step. This works only on files that have already been added to the git repo before. New files won't be added this way and need an explicit `git add` as seen before

So far everything seems pretty normal and we still have a straight line in the tree, but note that now `master` remained where it was and we moved forward with `my-feature-branch`.

Lets switch back to master and modify the same file there as well.

```
$ git checkout master
Switched to branch 'master'
```

As expected, `hallo.txt` is unmodified:

```
$ cat hallo.txt
Hello, world!
```

Lets change and commit it on master as well (this will generate a nice *conflict* later).

```
$ echo "Hi I was changed in master" >> hallo.txt
$ git commit -a -m "add line on hallo.txt"
c8616db] add line on hallo.txt
1 file changed, 1 insertion(+)
```

## Polishing your feature branch commits

When you create your own, personal feature branch you're allowed to do as much commits as you want, even with kinda dirty commit messages. This is a really powerful approach as you can jump back to any point in your dev cycle. However, **once you're ready to merge back to master** you should polish your commit history. This is done with the `rebase` command like this:

```
git rebase -i HEAD~<num-commits>
```

## Merge and resolve conflicts

The next step would be to merge our feature branch back into `master`. This is done by using the merge command

```
$ git merge my-feature-branch
Auto-merging hallo.txt
CONFLICT (content): Merge conflict in hallo.txt
Automatic merge failed; fix conflicts and then commit the result.
```

As expected, we have a merge conflict in `hallo.txt`.

```
Hello, world!  
<<<<<<< HEAD  
Hi I was changed in master  
=====  
Hi  
>>>>>> my-feature-branch
```

Lets resolve it:

```
Hello, world!  
Hi I was changed in master  
Hi
```

..and then commit it

```
$ git commit -a -m "resolve merge conflicts"  
[master 6834fb2] resolve merge conflicts
```



GitHub

GitHub is a for-profit company that offers a cloud-based Git repository hosting service. Essentially, it makes it a lot easier for individuals and teams to use Git for version control and collaboration.

It's interface is user-friendly enough so even novice coders can take advantage of Git. Without GitHub, using Git generally requires a bit more technical savvy and use of the command line.

GitHub is so user-friendly, though, that some people even use GitHub to manage other types of projects – like writing books.

Additionally, anyone can sign up and host a public code repository for free, which makes GitHub especially popular with open-source projects.



## Sharing/Synching your Repository

Ultimately we want to share our code, normally by synching it to a central repository. For doing so, we have to add a **remote**.

```
$ git remote add origin https://github.com/sarthakg043/demo.git
```

To see whether I succeeded, simply type:

```
$ git remote -v
```

which lists all of the added remotes. Now we need to **publish our local branch master** to the remote repository. This is done like

```
$ git push -u origin master
```

And we're done.

The real powerful thing is that you can add multiple different remotes. This is often used in combination with cloud hosting solutions for deploying your code on your server. For instance, you could add a remote named “deploy” which points to some cloud hosting server repository, like

```
$ git remote add deploy origin https://github.com/sarthakg043/demo.git
```

and then whenever you want to publish your branch you execute a

```
$ git push deploy
```

## Cloning

Similarly it works if you'd like to start from an existing remote repository. The first step that needs to be done is to “checkout” the source code which is called **cloning** in Git terminology. So we would do something like

```
$ git clone https://github.com/sarthakg043/demo.git
Cloning into 'intro.js'...
remote: Counting objects: 430, done.
```

```
remote: Compressing objects: 100% (293/293), done.  
remote: Total 430 (delta 184), reused 363 (delta 128)  
Receiving objects: 100% (430/430), 419.70 KiB | 102 KiB/s, done.  
Resolving deltas: 100% (184/184), done.
```

This will create a folder (in this case) named “intro.js” and if we enter it

```
$ cd intro.js/
```

and check for the remotes we see that the according tracking information of the remote repository is already set up

```
$ git remote -v  
origin https://github.com/sarthakg043/demo.git (fetch)  
origin https://github.com/sarthakg043/demo.git (push)
```



We can now start the commit/branch/push cycle just normally.



**Bonus Resource:** Here is a link to [Git Cheat sheet](#) made by [mujs.dev](#)

## Git vs GitHub

Git vs Github03



Git is a software.



Linux maintains Git.

Github is a service.

Microsoft maintains GitHub.

@tech.programmer.70

Git vs Github04



It is a command-line tool.



You can install it locally on the system.

It is a graphical user interface.

It is hosted on the web. It is exclusively cloud-based.

@tech.programmer.70

Git vs Github 05



It is a VCS to manage source code history.



It is a hosting service for Git repositories.

It focuses on code sharing and version control.

It focuses on centralized source code hosting.

@tech.programmer.70

Git vs Github 06



It lacks a user management feature.



It has a built-in user management feature.

Git was launched in 2005.

GitHub was released in 2008.

@tech.programmer.70

Git vs Github 07



Git has minimum external tool configuration.



It has an active marketplace for tool integration.

It is open-source licensed.

It has a free-tier and pay-for-use tier.

@tech.programmer.70

Git vs Github 08



Its desktop interface is named Git Gui.

Its desktop interface is named GitHub Desktop.

@tech.programmer.70

Images by [tech.programmer.70](https://www.instagram.com/tech.programmer.70)

# Thank You