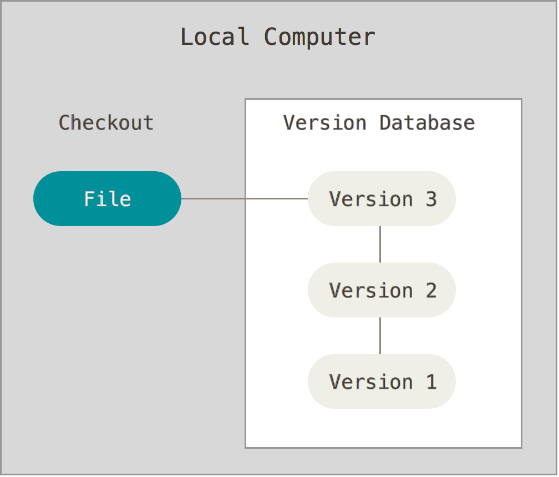
# Github :

**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.

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 Version Control System (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 all this for very little overhead.

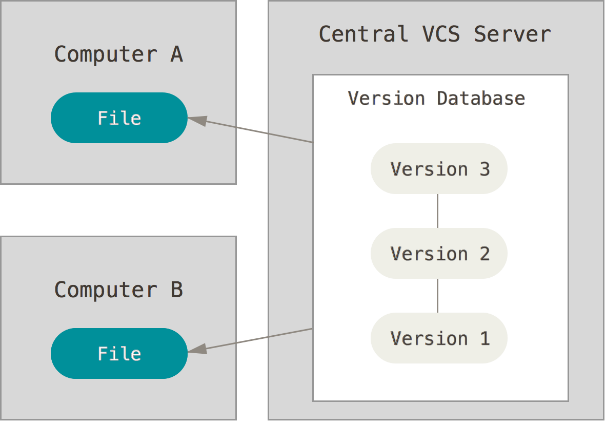
## Local Version Control Systems

One of the more popular VCS tools was a system called **RCS,** which is still distributed with many computers today. Even the popular Mac OS X operating system includes the rcs command when you install the Developer Tools. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.



## Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed 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.



**Advantages:**

**Difference bet TFS and Git**

* TFS is an Application Life-cycle Management solution, SVN and Git are source control only.
* TFS does source control as well as issue tracking, document management, reporting, continuous integration, virtual labs for testing etc.
* TFS's Source Control & SVN are centralized source control, Git is distributed.
* TFS is the most tightly integrated into Visual Studio.
* SVN has a few third party options for integrating into Visual Studio and they are quite nice, but not as tightly integrated as TFS.
* Git has GitExtensions which allows for a low level of integration within Visual Studio.

## Create an account from the below link :

<https://github.com/>

**Installing and aligning Git through Cmd :**

1. Download Git exe from the given link <http://git-scm.com/download/win> and Install it on your PC.
2. Now go to the folder in C drive where git is installed.
3. Copy the path **C:\Program Files\Git\bin.**
4. Right click on My Computer and click on Environment variables.
5. Edit Path and Add the above copied git path in environment variables.
6. Now open command prompt and run git command.

This will install and align command prompt to git and you can run git commands through command prompt.

1. **Your Identity:** After installation, you have to set username and password:

**$ git config --global user.name "John Doe"**

**$ git config --global user.email johndoe@example.com**

[user]name = Ankitasanadhya

[user]email = [reachankita.s1989@gmail.com](mailto:reachankita.s1989@gmail.com)

1. **Your Editor:** set youreditor also

Git uses your system’s default editor. If you want to use a different text editor, such as Emacs,

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

While on a Windows system, if you want to use a different text editor, such as Notepad++, you can do the following:

On a x86 system

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

On a x64 system

**$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -nosession"**

**To check settings:**

1. **git config --list** command to list all the settings Git can find at that point**.**
2. **git config <key>** you can check what Git thinks a specific key’s value

**Eg: $ git config user.name**

John Doe

**To get any help:**

1. **$ git help &lt;verb&gt;** or
2. **$ git &lt;verb&gt; --help** or
3. **$ man git-&lt;verb&gt;**

**Eg:** $ git help config

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

echo "# Test\_1-" >> README.md

git init

git add README.md

git commit -m "first commit"

git remote add origin https://github.com/Ankitasanadhya/Test\_1-.git

git push -u origin master

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

git remote add origin https://github.com/Ankitasanadhya/Test\_1-.git

git push -u origin master

## Recording Changes to the Repository

Each file in your working directory can be in one of two states: tracked or untracked.

**Tracked files** are files that were in the last snapshot; they can be unmodified, modified, or staged.

**Untracked files** are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.

## [Checking the Status of Your Files](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Checking-the-Status-of-Your-Files)

**$** echo 'My Project' > README

**$** git status

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

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

README

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

### [Short Status](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Short-Status)

If you run git status -s or *git status --short* you get a far more simplified output from the command:

**$** git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

?? New files that aren’t tracked

A New files that have been added to the staging area

M Modified files

There are two columns to the output - the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree

### [Tracking New Files](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Tracking-New-Files)

To begin tracking a new file, you use the command git add.

**$** git add README

If you modify a file after you run git add, you have to run git add again to stage the latest version of the file.

### [Ignoring Files](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Ignoring-Files)

This command is use to ignore any file from tracking.

**For linux,** Cat command is used.

**$** cat .gitignore

\*.[oa]

\*~

**For windows,**

You can simply create a text file named **.gitignore** and then add the file names (For eg. to ignore apk file u have to write \*.apk in your .gitignore file and save the gitignore file ) or you can use wildcard char also to specify file names that u want to ignore while pushing your code to server. Now push the gitignore to the server.

Next time whenever u push any changes to the server it will ignore to push the .apk files that are added in gitignore

Eg:

1. Create any file named **.gitignore** in local repository.
2. Add files that u want to ignore like \*.apk in .gitignore file and save it.
3. Now push the changes to server:

Git pull

Git add .

Git commit -m “updated gitignore”

Git push

This will push the gitignore file to server

1. Now add any .apk file in local repo and then push it to the server
2. You will see that all the changes done in local folder gets reflected to the server but it will ignore pushing the .apk file to the server as you have added it to gitignore.

\*.[oa]

\*~

The first line tells Git to ignore any files ending in “.o” or “.a” – object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a .gitignore file before you get going is generally a good idea so you don’t accidentally commit files that you really don’t want in your Git repository.

**The rules for the patterns you can put in the .gitignore file are as follows:**

* Blank lines or lines starting with # are ignored.
* Standard glob patterns work.
* You can start patterns with a forward slash (/) to avoid recursivity.
* You can end patterns with a forward slash (/) to specify a directory.
* You can negate a pattern by starting it with an exclamation point (!).

Glob patterns are like simplified regular expressions that shells use. An asterisk **(\*)** matches zero or more characters; **[abc]** matches any character inside the brackets (in this case a, b, or c); a question mark **(?)** matches a single character; and brackets enclosing characters separated by a hyphen **([0-9])** matches any character between them (in this case 0 through 9). You can also use two asterisks to match nested directories; **a/\*\*/z** would match **a/z**,**a/b/z**, **a/b/c/z**, and so on.

Here is another example .gitignore file:

# no .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in the build/ directory

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .pdf files in the doc/ directory

doc/\*\*/\*.pdf

**TIP**

GitHub maintains a fairly comprehensive list of good **.gitignore** file examples for dozens of projects and languages at [*https://github.com/github/gitignore*](https://github.com/github/gitignore) if you want a starting point for your project.

### [Viewing Your Staged and Unstaged Changes](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Viewing-Your-Staged-and-Unstaged-Changes)

**git status** command answers about which files were changed by listing the file names

**git diff** shows you the exact lines added and removed – the patch, as it were

Let’s say you edit and stage the README file again and then edit the CONTRIBUTING.md file without staging it. If you run your **git status** command, you once again see something like this:

**$** git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

To see what you’ve changed but not yet staged, type **git diff** with no other arguments:

**$** git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

If you want to see what you’ve staged that will go into your next commit, you can use **git diff --staged**. This command compares your staged changes to your last commit:

**$** git diff --staged

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

+My Project

**git diff --cached** to see what you’ve staged so far (--staged and --cached are synonyms):

**$** git diff --cached

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

**NOTE**

**Git Diff** in an External Tool

We will continue to use the **git diff** command in various ways throughout the rest of the book. There is another way to look at these diffs if you prefer a graphical or external diff viewing program instead. If you run **git difftool** instead of **git diff**, you can view any of these diffs in software like emerge, vimdiff and many more (including commercial products). Run **git difftool --tool-help** to see what is available on your system.

### [Committing Your Changes](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Committing-Your-Changes)

Use **commit** changes to the server

**$** git commit

### **-m** is used to write a comment regarding the commit.

### Eg.- git commit -m “Updated readme”

### [Skipping the Staging Area](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Skipping-the-Staging-Area)

### [Remove any file or directory](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Viewing-Your-Staged-and-Unstaged-Changes)

Use **rm** command to **delete any file**

Git rm filename

Git rm -r directoryname, (in linux, this command is used for recursive delete- delete all files inside the folder and then delete the folder)

To **delete any directory**, just delete it from local repository and then push the changes to server, this will delete it from server also.