Git

מדריך למידה

Table of Content

1. GIT

Git is a version control system (VCS) created for a single task: **managing changes to your files**.

It lets you track every change a software project goes through, as well as where those changes came from.

Benefits of version control:

- Enables easy Work with versions
- Code Together - Version control synchronizes versions and makes sure that your changes don't conflict with other changes from your team.
- Keeps a history - Version control keeps a history of changes as your team saves new versions of your code. This history can be reviewed to find out who, why, and when changes were made. History gives you the confidence to experiment since you can roll back to a previous good version at any time.

1.1   Evolution of version control system

Step 1 - Files and Folders

Before the advent of revision control software, there were only files and folders. The only way to track revisions of a project was to copy the entire project and give it a new name.

But, it's easy to see how copying files from folder to folder could prove disastrous for software developers. What happens if you mislabel a folder? Or if you overwrite the wrong file? How would you even know that you lost an important piece of code? It didn't take long for software developers to realize they needed something more reliable.
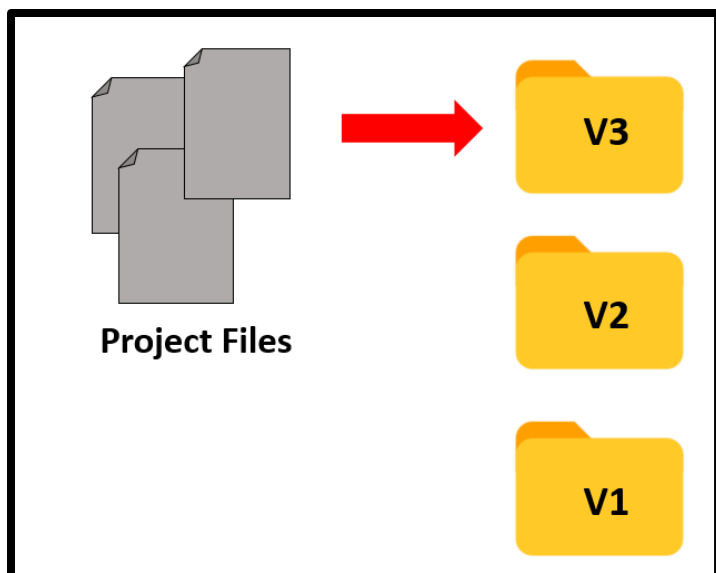
Step 2 - Local VCS

So, developers began writing utility programs dedicated to managing file revisions. Instead of keeping old versions as independent files, these new VCSs stored them in a database.

When you needed to look at an old version, you used the VCS instead of accessing the file directly. That way, you would only have a single "checked out" copy of the project at any given time, eliminating the possibility of mixing up or losing revisions.
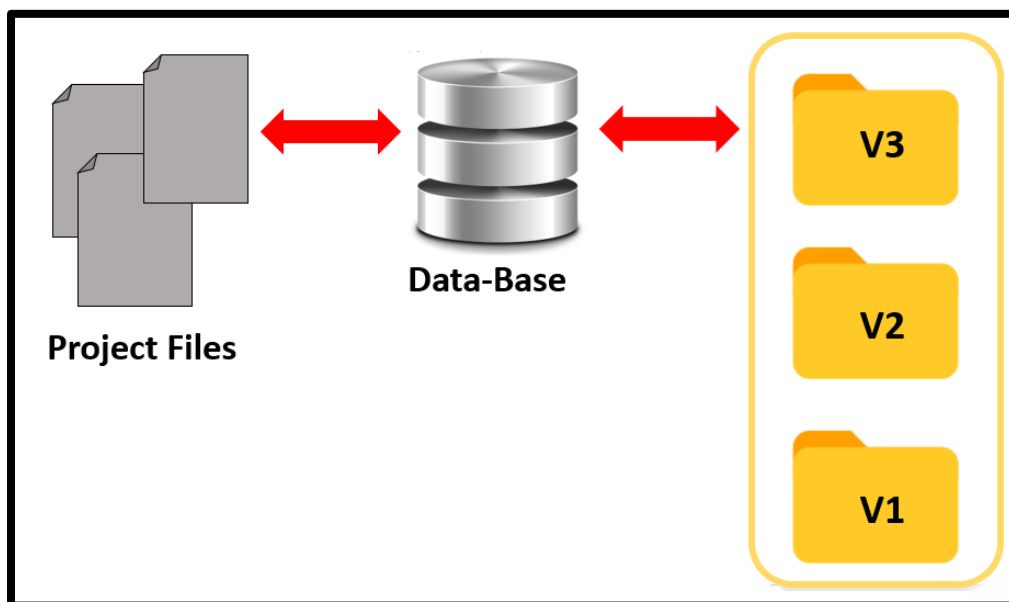
At this point, versioning only took place on the developer's local computer—there was no way to efficiently share code amongst several programmers.

Step 3 - Centralized VCS

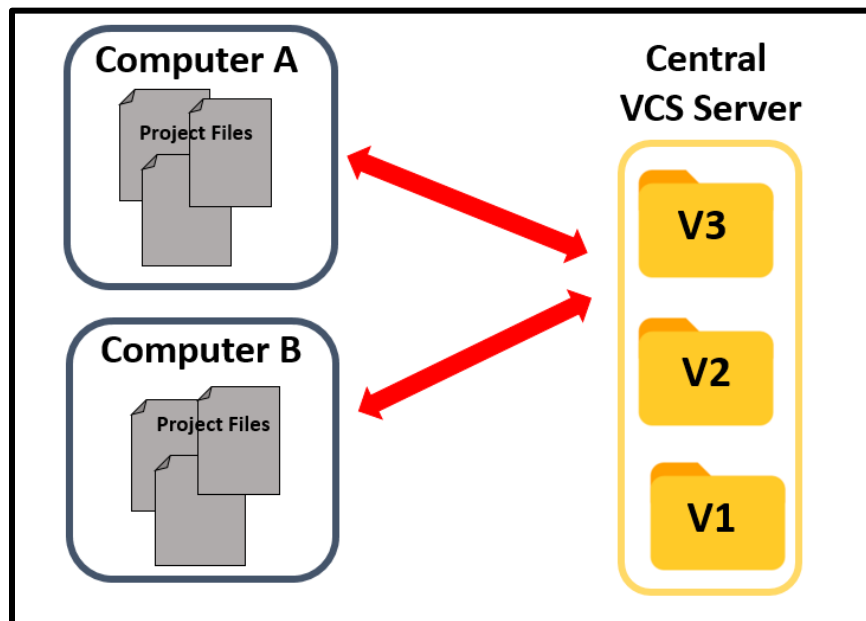Enter the centralized version control system (CVCS). Instead of storing project history on the developer's hard disk, these new CVCS programs stored everything on a server.

Developers checked out files and saved them back into the project over a network. This setup let several programmers collaborate on a project by giving them a single point of entry.

While a big improvement on local VCS, centralized systems presented a new set of problems: how do multiple users work on the same files at the same time? Just imagine a scenario where two people fix the same bug and try to commit their updates to the central server. Whose changes should be accepted?

CVCSs addressed this issue by preventing users from overriding other's work. If two changes conflicted, someone had to manually go in and merge the differences.

This solution worked for projects with relatively few updates (which meant relatively few conflicts), but proved cumbersome for projects with dozens of active contributors submitting several updates everyday: development couldn't continue until all merge conflicts were resolved and made available to the entire development team.

Step 4 - Distributed VCS

The next generation of revision control programs shifted away from the idea of a single centralized repository, opting instead to give every developer their own local copy of the entire project. The resulting distributed network of repositories let each developer work in isolation, much like a local VCS—but now the conflict resolution problem of CVCS had a much more elegant solution.
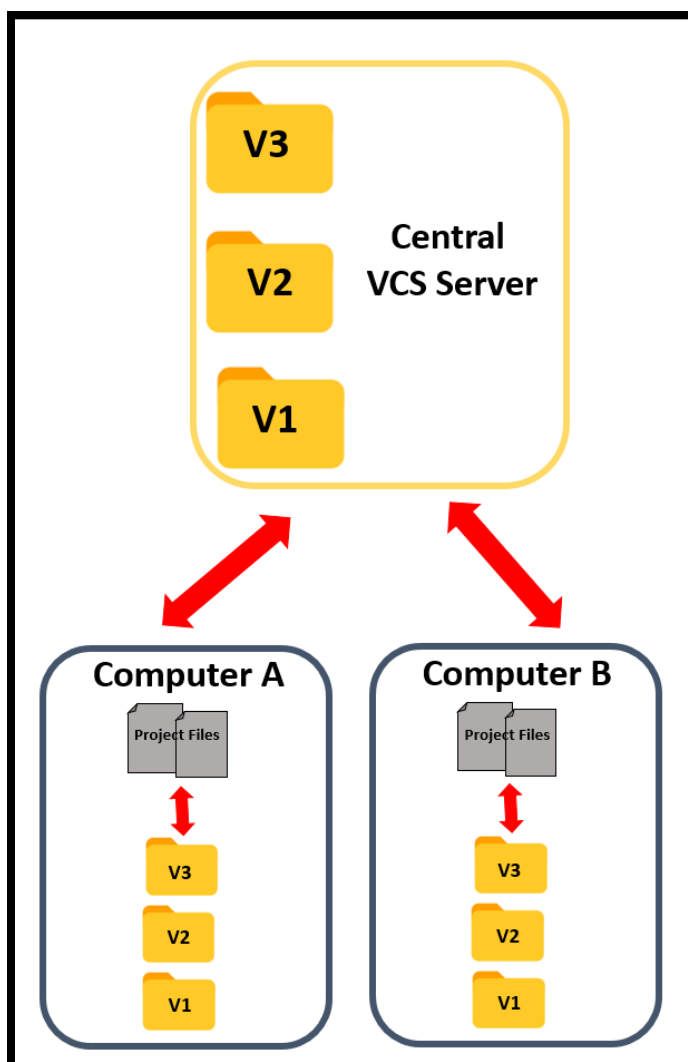
Since there was no longer a central repository, everyone could develop at their own pace, store the updates locally, and put off merging conflicts until their convenience. In addition, distributed version control systems (DVCS) focused on efficient management for separate branches of development, which made it much easier to share code, merge conflicts, and experiment with new ideas.

The local nature of DVCSs also made development much faster, since you no longer had to perform actions over a network. And, since each user had a complete copy of the project, the risk of a server crash, a corrupted repository, or any other type of data loss was much lower than that of their CVCS predecessors.
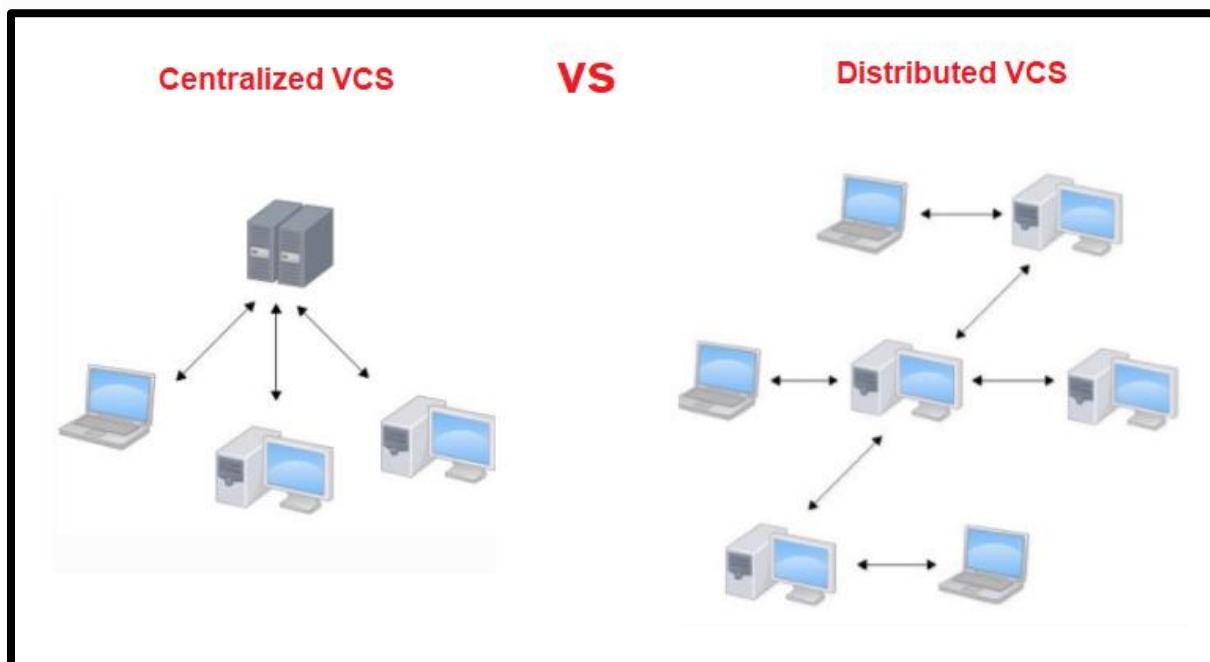
Step 5 - Git

And so, we arrive at Git, a distributed version control system created to manage the Linux kernel. In 2005, the Linux community lost their free license to the BitKeeper software, a commercial DVCS that they had been using since 2002. In response, Linus Torvalds advocated the development of a new open-source DVCS as a replacement. This was the birth of Git.

As a source code manager for the entire Linux kernel, Git had several unique constraints, including:

- Reliability
- Efficient management of large projects
- Support for distributed development
- Support for non-linear development

While other DVCSs did exist at the time, none of them could satisfy this combination of features. Driven by these goals, Git has been under active development for several years and now enjoys a great deal of stability, popularity, and community involvement.

Git originated as a command-line program, but a variety of visual interfaces have been released over the years. Graphical tools mask some of the complexity behind Git and often make it easier to visualize the state of a repository, but they still require a solid foundation in distributed version control.

We'll be sticking to the command-line interface, which is still the most common way to interact with Git.

1.2 Remote and local repositories

There are two types of Git repositories:

- **Remote repository** - Repository that resides on a remote server that is shared among multiple team members.
- **Local repository** - Repository that resides on a local machine of an individual user.

You can use all of Git's version control features on your local repository such as reverting changes, tracking changes, etc. However, when it comes to sharing your changes or pulling changes from your team members, that is where a remote repository is needed.
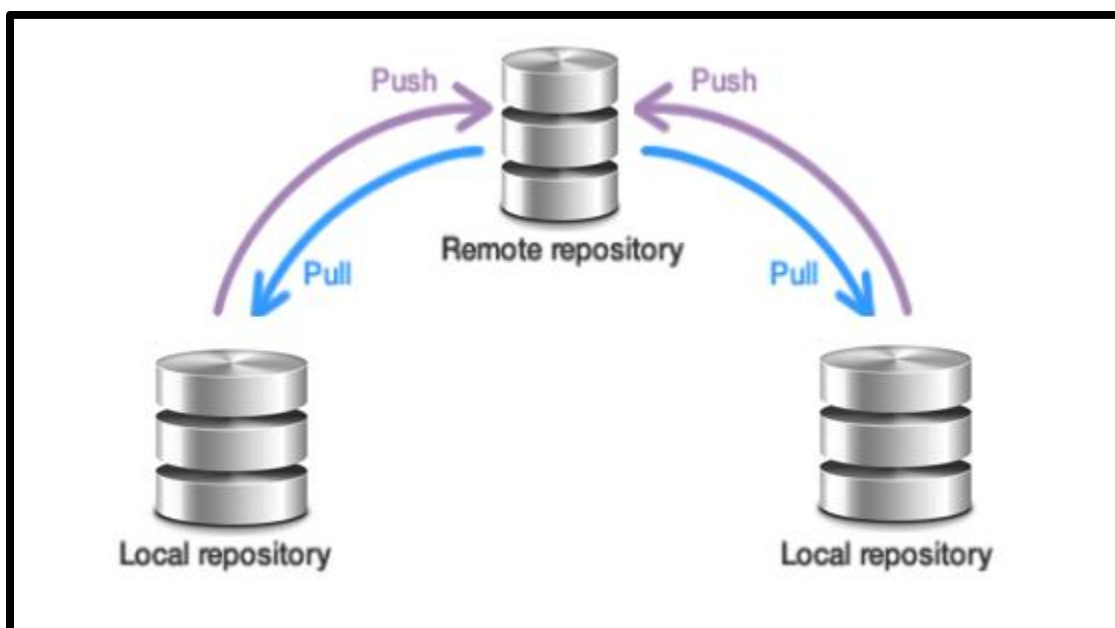
2. GIT Basic setup

2.1 Git Installation

Downloads for all supported platforms are available via the official Git website.

For Windows users, this will install a special command shell called Git Bash. You should be using this shell instead of the native command prompt to run Git commands.

To test your installation, run the git **--version** command:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /
$ git --version
git version 2.14.1.windows.1
```

To clear all the displayed text, type the following command:

```
$ clear
```

2.2 First time Git setup

the first setup will stay between upgrades. (Changing this setup can be made at any time by running through the commands again)

Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

1. /etc/gitconfig file: Contains values for every user on the system and all their repositories. If you pass the option **--system** to git config, it reads and writes from this file specifically.
2. ~/.gitconfig or ~/.config/git/config file: Specific to your user. You can make Git read and write to this file specifically by passing the **--global** option.
3. config file in the Git directory (that is git/config) of whatever repository you're currently using: Specific to that single repository.

Each level overrides values in the previous level, so values in. git/config trump those in /etc/gitconfig.

2.3 Git user setup

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

This is important because every Git commit uses this information, and it's immutably baked into the commits.

The name and email are modified with the following commands:

$ git config --global user.name <"NAME">
$ git config --global user.email <"EMAIL">

```
Anna@DESKTOP-RLP7NAJ MINGW64 /
$ git config --global user.name "Ana Karp"

Anna@DESKTOP-RLP7NAJ MINGW64 /
$ git config --global user.email "anakarpf6@gmail.com"
```

If you want to check your settings, you can use the **git config --list** command to list all the settings:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /
$ git config --list
core.symlinks=false
core.autocrlf=true
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
diff.astextplain.textconv=astextplain
rebase.autosquash=true
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
http.sslbackend=openssl
diff.astextplain.textconv=astextplain
credential.helper=manager
user.name=Ana Karp
user.email=anakarpf6@gmail.com
```

You can also check Git specific key's value by typing **git config <key>**:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /
$ git config user.name
Ana Karp

Anna@DESKTOP-RLP7NAJ MINGW64 /
$ git config user.email
anakarpf6@gmail.com
```

2.4   Getting help

There are a few ways to get the manual page help for any of the Git commands:

$ git help <verb>
$ git <verb> --help

For example, get help for the **config** command by running:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /
$ git help config
```

or:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /
$ git config --help
```

These commands can be accessed even offline, and will show the following result:

2.5   Basic commands

- **pwd** - shows the current location

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c
$ pwd
/c
```

- **cd -** (change directory) changes the current location to the given path
- **cd .. -** goes back one directory
- **ls** - lists all the folder in the current directory

```
Anna@DESKTOP-RLP7NAJ MINGW64 /
$ cd /c/Git_Rep

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ cd ..

Anna@DESKTOP-RLP7NAJ MINGW64 /c
$ ls
'$GetCurrent'/     devlist.txt                Git_Rep/
'$Recycle.Bin'/   'Documents and Settings'@   hiberfil.sys
 Boot/             emu8086/                   Intel/
```

In this tutorial we will create our repository in the following path: C:/ Git_Rep.

In order to initialize and start this folder to an empty Git repository, we can change our current directory with **cd** command to "/C/ Git_Rep" , and then type **git init:**

```
Anna@DESKTOP-RLP7NAJ MINGW64 /
$ cd /c/Git_Rep

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep
$ git init
Initialized empty Git repository in C:/Git_Rep/.git/
```

Now let's check this folder in file explorer:
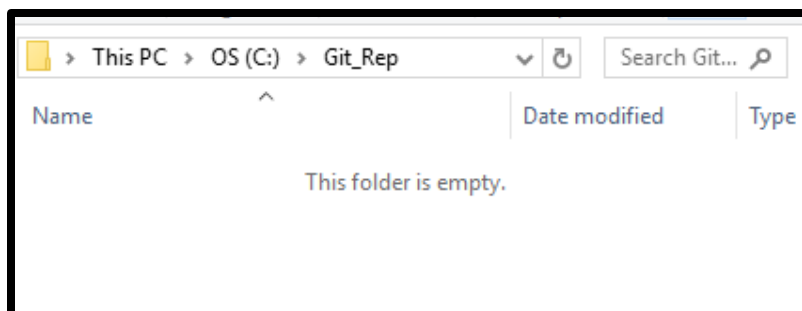
As we can see the folder is empty.

But there are some hidden files that have been added to this folder when we executed the init command.
We can view this hidden file, by adding the **-la** option with the ls command:

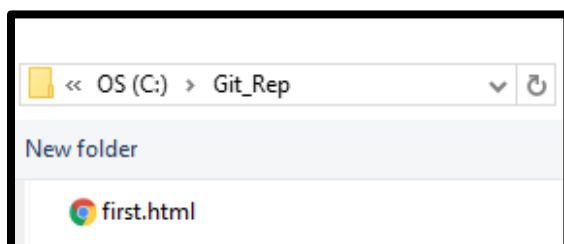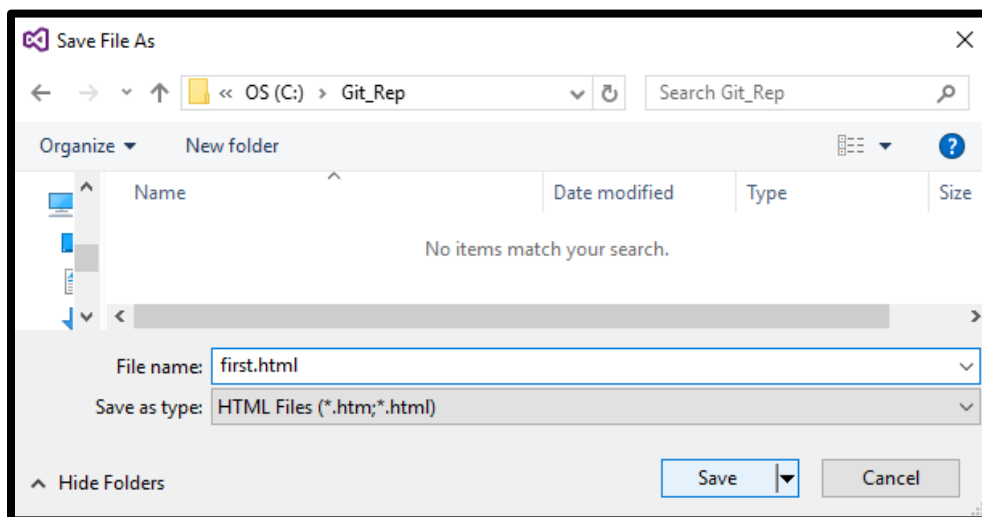# 3. GIT Repository and commit

Our first step is creating a simple html file:

```html
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <h1>first line</h1>
</body>
</html>
```

Now - Let's Save this file in the Git_Rep folder:

3.1 Stage a Snapshot

A snapshot represents the state of your project at a given point in time.

Git's term for creating a snapshot is called staging because we can add or remove multiple files before actually committing it to the project history. Staging gives us the opportunity to group related changes into distinct snapshots, makes it possible to track the meaningful progression of a software project (instead of just arbitrary lines of code).

To inform Git about the changes that we made (we added a new file), Use the **add** command:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git add .
```

The git **add** command takes one of the following options:

- a path name for either a file or a directory - if it's a directory, the command adds all the files in that directory recursively.
- **.** - adds all the directory's content

3.2 Commit a snapshot

Saving a version of your project is a two-step process:

1. Staging - Telling Git what files to include in the next commit.
2. Committing - Recording the staged snapshot with a descriptive message.
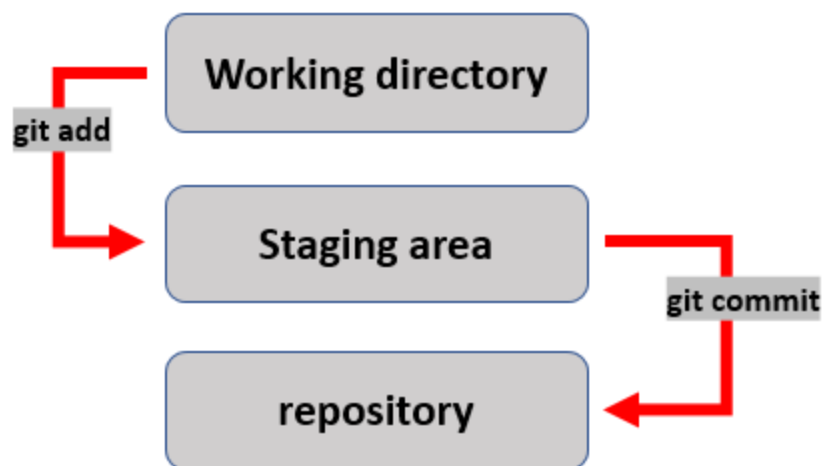
Staging files with the git add command doesn't affect the repository in any significant way—it just lets us get our files for the next commit. Only after executing **git commit** will our snapshot be recorded in the repository. Committed snapshots can be seen as "safe" versions of the project. Git will never change them, which means you can do almost anything you want to your project without losing those "safe" revisions. This is the principal goal of any version control system.
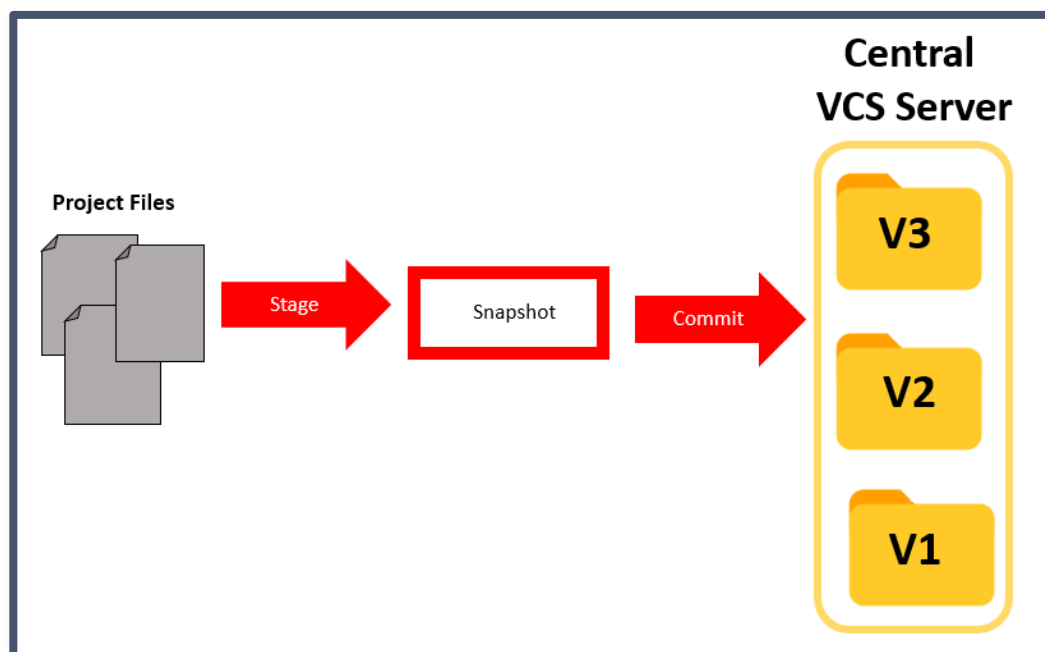
We already staged our changes, so let's go ahead and commit them:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git commit -m "this is the first step"
[master (root-commit) b63c6ad] this is the first step
 1 file changed, 17 insertions(+)
 create mode 100644 first.html
```

3.3 View a Repository History

to view our commit history:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git log
commit b63c6adca17decf30ae437d12b2a9d5929d572d0 (HEAD -> master)
Author: Ana Karp <anakarpf6@gmail.com>
Date:   Thu Sep 18 17:18:26 2017 +0300

    this is the first step
```

By default, with no arguments, git log lists the commits made in that repository in reverse chronological order – that is, the most recent commits show up first.

Every commit shows the following details:
- First, the commit is identified with an SHA-1 checksum of the commit's contents, which ensures that the commit will never be corrupted without Git knowing about it.
- Next, Git displays the author of the commit. and outputs the date, time, and time zone (+0300) of when the commit took place.
- Finally, we see the commit message that was entered in commit command.

3.4 Summarizing options with Git log

Option	Description	Usage Example:
-p	Show the patch introduced with each commit.	-p, which shows the difference introduced in each commit. You can also use -2, which limits the output to only the last two entries: $ git log -p -2
--stat	Show statistics for files modified in each commit.	$ git log --stat
--shortstat	Display only the changed/insertions/deletions line from the --stat command.	$ git log --shortstat
--name-only	Show the list of files modified after the commit information.	$ git log --name-only
--name-status	Show the list of files affected with added/modified/deleted information as well.	$ git log --name-status
--abbrev-commit	Show only the first few characters of the SHA-1 checksum instead of all 40.	$ git log --abbrev-commit
--relative-date	Display the date in a relative format (for example, "2 weeks ago") instead of using the full date format.	$ git log --relative-date

--graph	Display an ASCII graph of the branch and merge history beside the log output.	$ git log --graph

--pretty	Show commits in an alternate format. Options include oneline, short, full, fuller, and format (where you specify your own format).	oneline option prints each commit on a single line, which is useful if you're looking at a lot of commits:
		
		$ git log --pretty=oneline
		
		The format, allows to specify the log output format.
		
		$ git log --pretty=format:"%h - %an, %ar : %s"
		The list of the formatting options:

The list of the formatting options:

Option	Description of Output
%H	Commit hash
%h	Abbreviated commit hash
%T	Tree hash
%t	Abbreviated tree hash
%P	Parent hashes
%p	Abbreviated parent hashes
%an	Author name
%ae	Author email
%ad	Author date (format respects the --date=option)
%ar	Author date, relative
%cn	Committer name
%ce	Committer email
%cd	Committer date

3.5   Filtering options with Git

Option	Description
-(n)	Show only the last n commits
--since, --after	Limit the commits to those made after the specified date. For example, command gets the list of commits made in the last two weeks: $ git log --since=2.weeks
--until, --before	Limit the commits to those made before the specified date.
--author	Only show commits in which the author entry matches the specified string.
--committer	Only show commits in which the committer entry matches the specified string.
--grep	Only show commits with a commit message containing the string
-S	Only show commits adding or removing code matching the string

For example:

on a huge project with a bunch of people like a team of programmers then it's helpful to view commits from only a specific (you don't need to type in the full name ,you can only type part of the name)

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git log --author="Ana K"
```

3.6 Status output VS Log output

status command compares our repository which is that main project against our working directory which is just our local copies that we are working on. and it tells us if there are any changes

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Remember, we can see staged changes with **git status**, but not with **git log**. The latter is used only for committed changes.
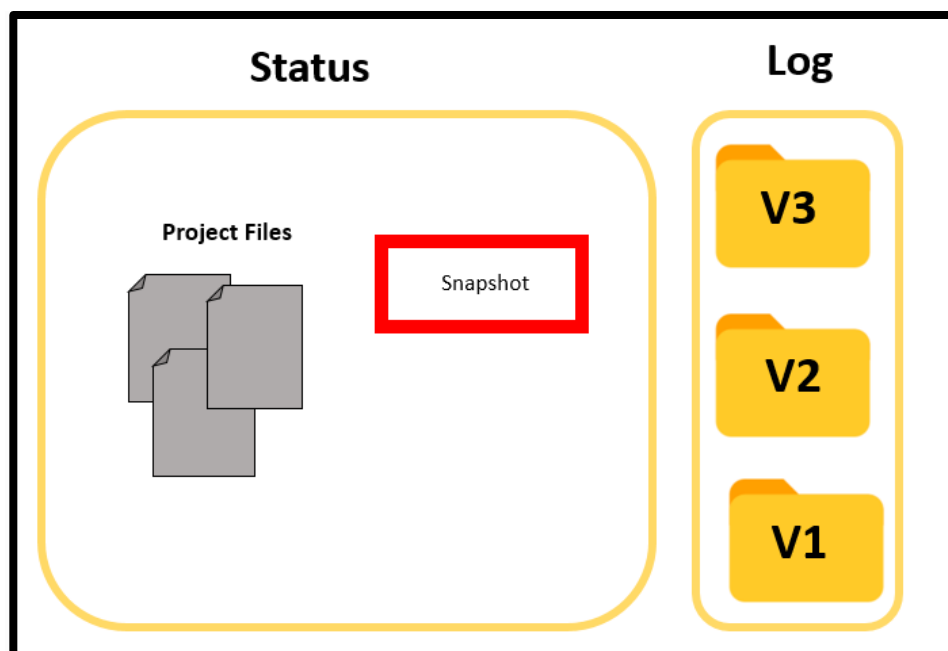
Let's create another simple html file:

```html
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <h1>second line</h1>
</body>
</html>
```

Save this file in the Git_Rep folder:

```
 >  This PC  >  OS (C:)  >  Git_Rep

Name

  first.html
  second.html
```
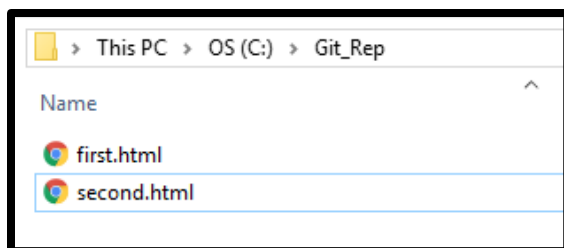
Now, since we did not add it to our staging area, if we will check our status we will get the following output:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        second.html

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

We can see that we have untracked files - untracked means that we didn't commit them and if we don't commit them git isn't keeping track of them, even if we will write now a commit command.

An untracked file is one that is not under version control. Git doesn't automatically track files because there are often project files that we don't want to keep under revision control. These include binaries created by a C program, compiled Python modules (.pyc files), and any other content that would unnecessarily bloat the repository. To keep a project small and efficient, you should only track source files and omit anything that can be generated from those files. This latter content is part of the build process—not revision control.

To save the second.html file in the repository, we must follow two steps:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git add second.html

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git commit -m "this commit added the second page"
[master de0f535] this commit added the second page
 1 file changed, 17 insertions(+)
 create mode 100644 second.html
```

Now, if we will check our history log, we will get the following output:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git log --pretty=format:"%an. Msg: %s"
Ana Karp. Msg: this commit added the second page
Ana Karp. Msg: this is the first step
```

Current project history:

(Each circle represents a commit, the red circle designates the commit we're currently viewing, and the arrow points to the preceding commit).

4. Modify files in the repository

Let's make a change in the first.html file, and then save it:

```
first.html
    <!DOCTYPE html>

    <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta charset="utf-8" />
        <title></title>
    </head>
    <body>
        <h1>first line</h1>
        <p>change in the first page</p>
    </body>
    </html>
```

Now this file on our computer is different than the file in our repository because in the repository we still have the previous version:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
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:   first.html

no changes added to commit (use "git add" and/or "git commit -a")
```

git realized that we modified the first.html file. But these changes must to be added to the stage in order to include them in the commit:
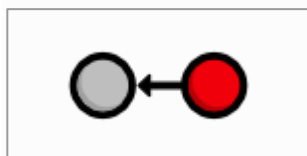
```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git add first.html

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git commit -m "saving changes in the first page"
[master f65fe37] saving changes in the first page
 1 file changed, 1 insertion(+)
```

Current project history:
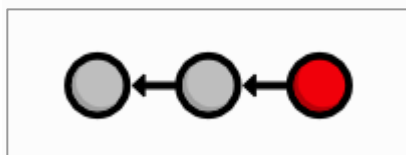
As we can see through the log:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git log --pretty=format:"%an. Msg: %s"
Ana Karp. Msg: saving changes in the first page
Ana Karp. Msg: this commit added the second page
Ana Karp. Msg: this is the first step
```

4.1 View diff in files

Let's make again a change in the first.html file, and then save it:
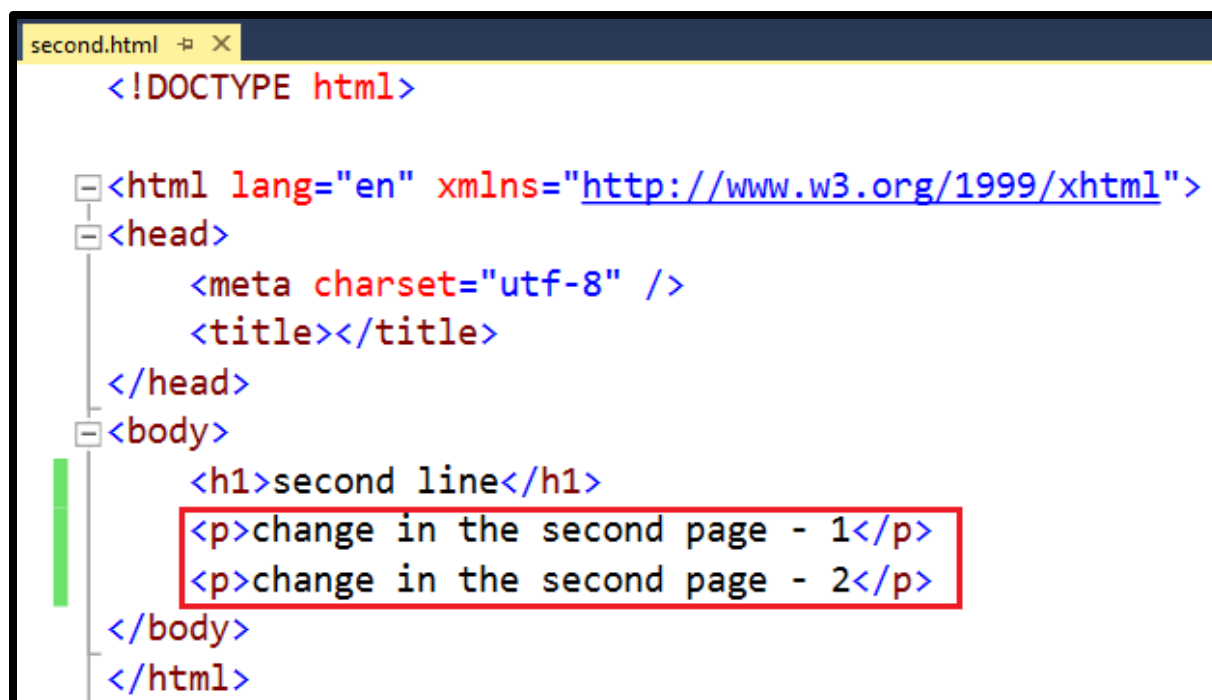
```
first.html    ↗ X                                      29
    <!DOCTYPE html>

  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
        <meta charset="utf-8" />
        <title></title>
  </head>
  <body>
        <h1>first line</h1>
        <p>change in the first page - again</p>
  </body>
  </html>
```

Now this file on our computer is different than the file in our repository because in the
repository we still have the previous version, and we can view all the changes with the
diff command:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git diff
diff --git a/first.html b/first.html
index 89cb812..3f5bdcf 100644
--- a/first.html
+++ b/first.html
@@ -7,7 +7,7 @@
 </head>
 <body>
        <h1>first line</h1>
-       <p>change in the first page</p>
+       <p>change in the first page - again</p>
 </body>
 </html>
```

Let's also make some changes in the second.html file, and then save them:

```
second.html  ⊣ ✕                        30
    <!DOCTYPE html>

  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
      <meta charset="utf-8" />
      <title></title>
  </head>
  <body>
      <h1>second line</h1>
      <p>change in the second page - 1</p>
      <p>change in the second page - 2</p>
  </body>
  </html>
```

Now the first.html and the second.html files on our computer are different than the files

in our repository, and we can view all the changes with the **diff** command:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git diff
diff --git a/first.html b/first.html
index 89cb812..3f5bdcf 100644
--- a/first.html
+++ b/first.html
@@ -7,7 +7,7 @@
 </head>
 <body>
     <h1>first line</h1>
-    <p>change in the first page</p>
+    <p>change in the first page - again</p>
 </body>
 </html>

diff --git a/second.html b/second.html
index 5cca862..6f94829 100644
--- a/second.html
+++ b/second.html
@@ -7,6 +7,8 @@
 </head>
 <body>
     <h1>second line</h1>
+    <p>change in the second page - 1</p>
+    <p>change in the second page - 2</p>
 </body>
 </html>
```

We can see that all the changes, in both files, are recognized.

Now, let's add the first.html file to the stage:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git add first.html
```

And if we type again the **diff** command, we will get only the changes in the second.html file:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git diff
diff --git a/second.html b/second.html
index 5cca862..6f94829 100644
--- a/second.html
+++ b/second.html
@@ -7,6 +7,8 @@
 </head>
 <body>
     <h1>second line</h1>
+    <p>change in the second page - 1</p>
+    <p>change in the second page - 2</p>
 </body>
 </html>
```

The reason that the changes in first.html are not displayed although they ar not commuted yet, is that the **diff** command recognizes only the changes between the local folder and the stage.

If we want to get all the changes between the stage and the commit, we can add to the **diff** command the **--staged** option:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git diff --staged
diff --git a/first.html b/first.html
index 89cb812..3f5bdcf 100644
--- a/first.html
+++ b/first.html
@@ -7,7 +7,7 @@
 </head>
 <body>
     <h1>first line</h1>
-    <p>change in the first page</p>
+    <p>change in the first page - again</p>
 </body>
 </html>
```

Let's save all the changes in both files, and commit:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git add second.html

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git commit -m "saving changes in both files"
[master 0e787a0] saving changes in both files
 2 files changed, 3 insertions(+), 1 deletion(-)
```

Now, we have no changes between the local files and the stage or between the stage
and the commit:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git diff --staged

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git diff
```

4.2 Delete files from the repository

Our local folder contains the following files:

> This PC > OS (C:) > Git_Rep

Name

- first.html
- second.html
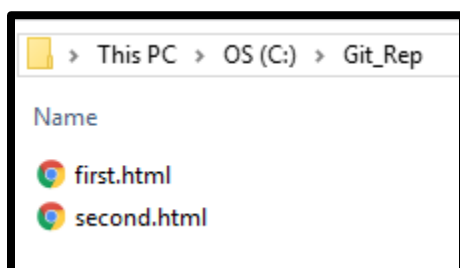
If we want to remove second.html, we can use the **rm** command:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git rm second.html
rm 'second.html'
```

Now, if we open again the local folder, we will see the following result:

:

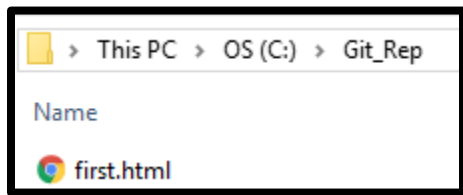But if we check the git status, we will see that the changes are not committed (it is only staged):

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    second.html
```
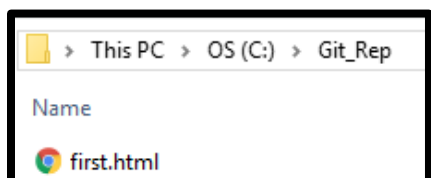
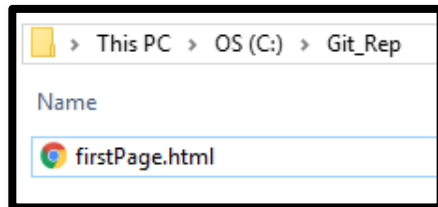So, the last step in the file removing, is to commit:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git commit -m "removed second.html"
[master 4ec1797] removed second.html
 1 file changed, 19 deletions(-)
 delete mode 100644 second.html
```

4.3   Rename files in the repository

The state of our local folder is:

:

Let's rename first.html to firstPage.html:

Now, if we check the git status, we will get the following output:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    first.html

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

        firstPage.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Git realized the file renaming, as two different actions:

1. removing first.html

2. adding a new firstPage.html file.

so , let's stage these two changes:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git rm first.html
rm 'first.html'

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git add firstPage.html

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    first.html -> firstPage.html
```

And now, if we commit, the rename will be snapshotted:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git commit -m "rename first.html to firstPage.html"
[master 0c50f5e] rename first.html to firstPage.html
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename first.html => firstPage.html (100%)
```

An easier way to rename the file, is to do it with the **mv** command:

We have the following folder -

```
> This PC > OS (C:) > Git_Rep

Name

🔴 firstPage.html
```

In the following command we are renaming firstPage.html to Page1.html:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git mv firstPage.html Page1.html
```

And we can see that this command affects our local file:

```
> This PC > OS (C:) > Git_Rep

Name

🔴 Page1.html
```

Let's check if this command staged the renaming action:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    firstPage.html -> Page1.html
```
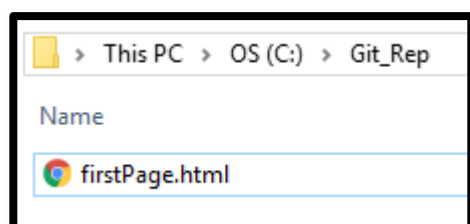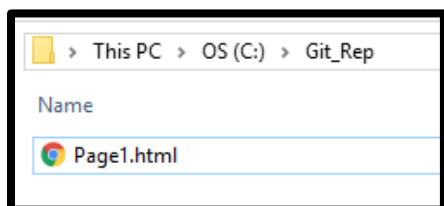
The renaming action is staged, so our last step is to commit:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git commit -m "rename firstPage.html to Page1.html"
[master 8ebf650] rename firstPage.html to Page1.html
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename firstPage.html => Page1.html (100%)
```

4.4 Return to current version

Sometimes, we might do some changes in our local files, and regret these changes.
In the following scenario we will see how to undo the changes in our local files, by
updating our local files to the last commit in git.

Let's change our Page1.html file, and save it:

```
Page1.html ⊓ ✕
   <!DOCTYPE html>

 ⊟<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
 ⊟<head>
       <meta charset="utf-8" />
       <title></title>
   </head>
 ⊟<body>
       <h1>first line</h1>
       <p>change in the first page - again</p>
       <p>added by mistake</p>
   </body>
   </html>
```
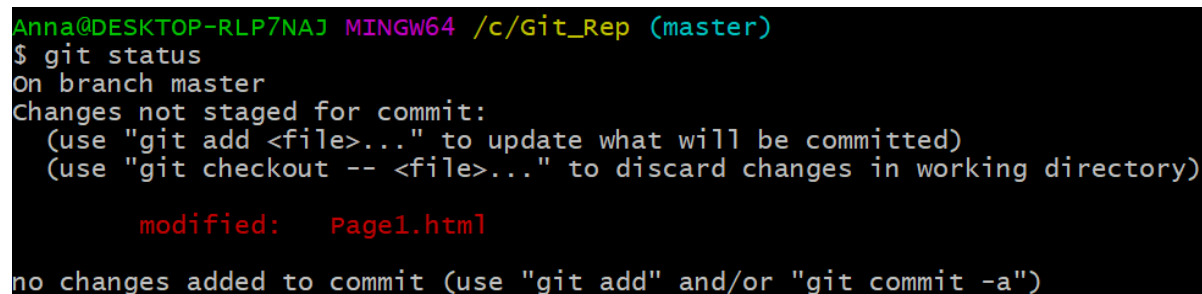
Now, Let's check the git status:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
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:   Page1.html

no changes added to commit (use "git add" and/or "git commit -a")
```

We can see that git recognized the change, and this change is not staged yet. So, we can get all the differences between the local files and the commit with the **diff** command:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git diff
diff --git a/Page1.html b/Page1.html
index 3f5bdcf..8372720 100644
--- a/Page1.html
+++ b/Page1.html
@@ -1,4 +1,4 @@
-<!DOCTYPE html>
+<U+FEFF><!DOCTYPE html>

 <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
 <head>
@@ -8,6 +8,7 @@
 <body>
     <h1>first line</h1>
     <p>change in the first page - again</p>
+    <p>added by mistake</p>
 </body>
 </html>
```

In order to undo this change, we can use the **git checkout** command to return to the **master** branch.

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git checkout -- Page1.html
```

This makes Git update our working directory to reflect the state of the **master** branch's snapshot. It re-creates the **Page1.html** file for us, and the content of **Page1.html** is updated as well. We're now back to the current state of the project:
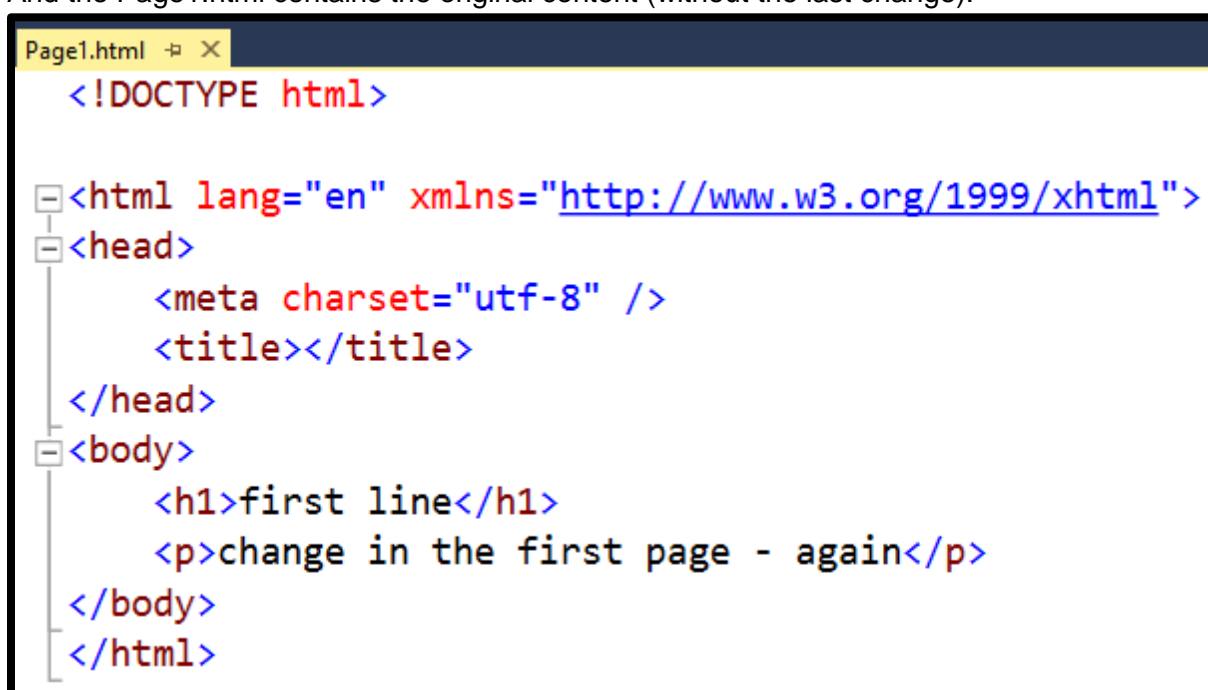
```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
nothing to commit, working tree clean

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git diff

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ |
```

And the Page1.html contains the original content (without the last change):
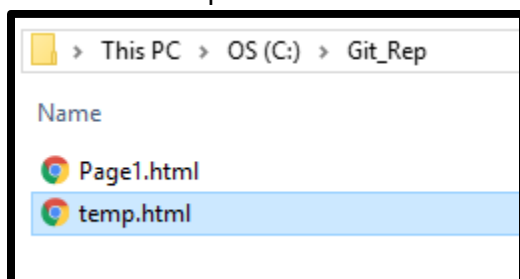


```
Page1.html  ⊟  ✕
  <!DOCTYPE html>

  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
      <meta charset="utf-8" />
      <title></title>
  </head>
  <body>
      <h1>first line</h1>
      <p>change in the first page - again</p>
  </body>
  </html>
```

4.5   Remove untracked files

Let's add a temp.html file to our local folder:


```
> This PC > OS (C:) > Git_Rep
Name
  Page1.html
  temp.html
```

The status will show that temp.html is untracked:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        temp.html

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

Next, let's remove the **temp.html** file. Of course, we could manually delete it, but using Git to reset changes eliminates human errors when working with several files in large teams. Run the following command:

- **git clean -f** - This will remove all untracked files

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git clean -f
Removing temp.html
```

Now, when we check our local folder we will see the following result:

> This PC > OS (C:) > Git_Rep

Name

🌐 Page1.html
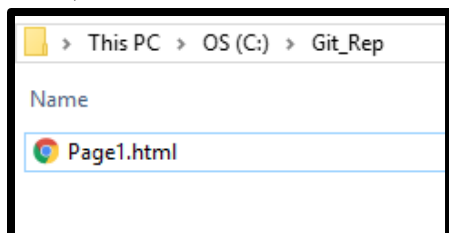
With **temp.html** gone, **git status** should now tell us that we have a "clean" working directory, meaning our project matches the most recent commit:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
nothing to commit, working tree clean
```

4.6   Undo Uncommitted changes

Before we start undoing things, let's do a change in our Page1.html file:

```
Page1.html  ⊉ ✕
  <!DOCTYPE html>

  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
      <meta charset="utf-8" />
      <title></title>
  </head>
  <body>
      <h1>first line</h1>
      <p>change in the first page - again</p>
      <p>this line will be stages and unstaged</p>
  </body>
  </html>
```

In the next step, we will stage this change:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git add Page1.html
```

So, we can see this staged change, at the status of our repository:

```
AnAnna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ $ git status
OnOn branch master
ChChanges to be committed:
    (use "git reset HEAD <file>..." to unstage)

        modified:   Page1.html
```

Now in order to unstage this stage we will use the git reset command with the following syntax:
   ● **git reset HEAD <FILE_NAME>** - Unstaging tracked files.

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git reset HEAD Page1.html
Unstaged changes after reset:
M        Page1.html
```

Now, we can see that the change is not staged:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
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:   Page1.html

no changes added to commit (use "git add" and/or "git commit -a")
```

And Page1.html still has the change:

```
Page1.html  ⌐ ×
    <!DOCTYPE html>

  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
        <meta charset="utf-8" />
        <title></title>
  </head>
  <body>
        <h1>first line</h1>
        <p>change in the first page - again</p>
        <p>this line will be stages and unstaged</p>
  </body>
  </html>
```

So, if we will type the **add** command:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git add Page1.html
```

The change that we unstaged' is staged again.

Now, let's use another option of the reset command:

- **git reset --hard** -Reset tracked files to match the most recent commit.

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git reset --hard
```

Now, we can see that the change does not exist in our Page1.html file:

```
Page1.html  + ✕
    <!DOCTYPE html>

⊟<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
⊟<head>
      <meta charset="utf-8" />
      <title></title>
  </head>
⊟<body>
      <h1>first line</h1>
      <p>change in the first page - again</p>
  </body>
  </html>
```

And the status will show that the tree is clean:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git status
On branch master
nothing to commit, working tree clean
```

In either case, **git reset** only operates on the working directory and the staging area, so our **git log** history remains unchanged.

Be careful with **git reset** and **git clean**. Both operate on the working directory, not on the committed snapshots. They permanently undo changes, so make sure you really want to trash what you're working on before you use them.

5. Versions management

5.1 Tag a Release

Let's call this a stable version of the example website. We can make it official by tagging the most recent commit with a version number.

git tag -a <TAG_NAME> -m "<DESCRIPTION>"

Create an annotated tag pointing to the most recent commit:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (v2)
$ git tag -a t1 -m "my msg"
```

Tags are convenient references to official releases and other significant milestones in a software project. It lets developers easily browse and check out important revisions.

To view a list of existing tags, execute **git tag** without any arguments.

In the above snippet, the **-a** flag tells Git to create an annotated tag, which lets us record our name, the date, and a descriptive message (specified via the **-m** flag).

Let's look at our stable revision. (Remember that the t1 tag now serves as a user-friendly shortcut to the last commit's ID):

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (v2)
$ git checkout t1
Note: checking out 't1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 8ebf650... rename firstPage.html to Page1.html

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep ((t1))
$
```

5.2 View an old revision

Let's look on our repository history:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git log --pretty=format:"%an. Hash: %h. Msg: %s"
Ana Karp. Hash: 8ebf650. Msg: rename firstPage.html to Page1.html
Ana Karp. Hash: 0c50f5e. Msg: rename first.html to firstPage.html
Ana Karp. Hash: 4ec1797. Msg: removed second.html
Ana Karp. Hash: 0e787a0. Msg: saving changes in both files
Ana Karp. Hash: f65fe37. Msg: saving changes in the first page
Ana Karp. Hash: de0f535. Msg: this commit added the second page
Ana Karp. Hash: b63c6ad. Msg: this is the first step
```

we are now in commit 8ebf650 (the latest snapshot), and our local folder contains the following content:

> This PC > OS (C:) > Git_Rep

Name

Page1.html
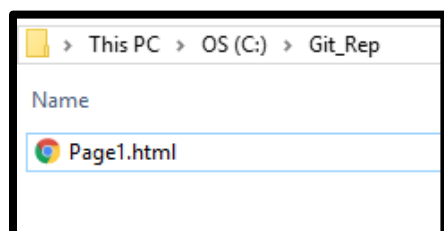
We can see that we have yet only one file. But we want to roll back to this commit:

```
Ana Karp. Hash: 0e787a0. Msg: saving changes in both files
```

To go back to a specific commit, we will use the command
git checkout <COMMIT_ID>

```
$ git checkout 2440cc8
Note: checking out '2440cc8'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 2440cc8... changes 0e787a0 commit

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep ((2440cc8...))
$
```
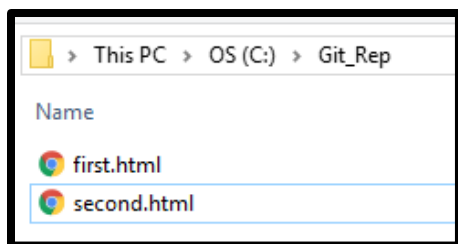
This will output a lot of information about a detached HEAD state, and all this information means that the head moved from one commit to another, as we can see in the following diagram:
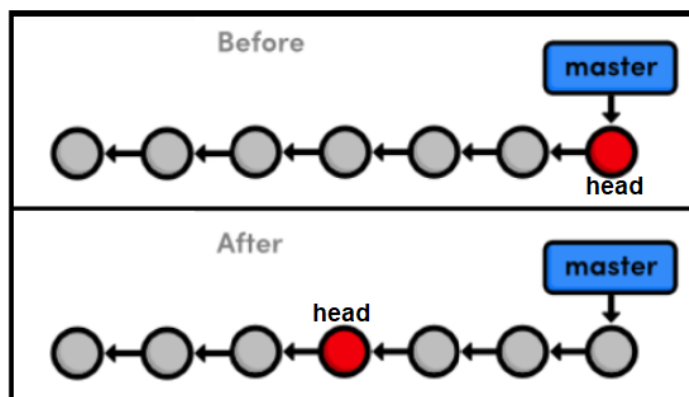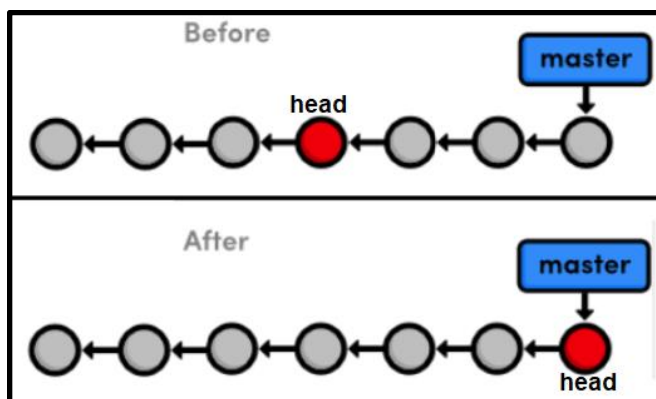

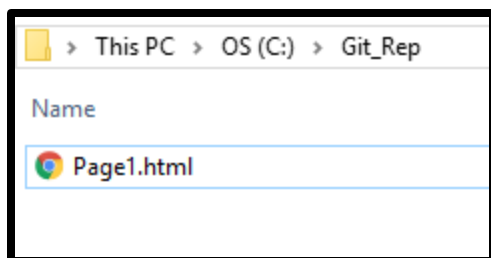And as we expect, the local folder will now contain the following content:

We can easily return our **HEAD** to the last commit, with the following command:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep ((2440cc8...))
$ git checkout master
```

Now, Our HEAD moved again:

And the local folder will contain the following content:

5.3   View an old revision file

We can stay in the current commit, and add to our local folder a specific file from an old commit. this action is done with the following syntax:
git checkout <COMMIT_ID> -- <FILE_NAME>

in this case, we didn't move our **HEAD**, and our diagram is without any changes between the before section and the after section:
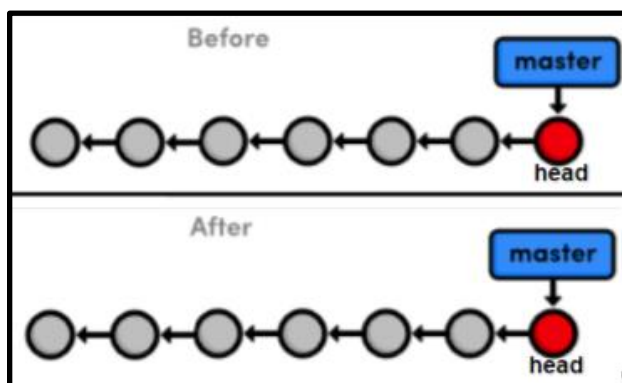
The only effect of this command, reflects in our local folder:
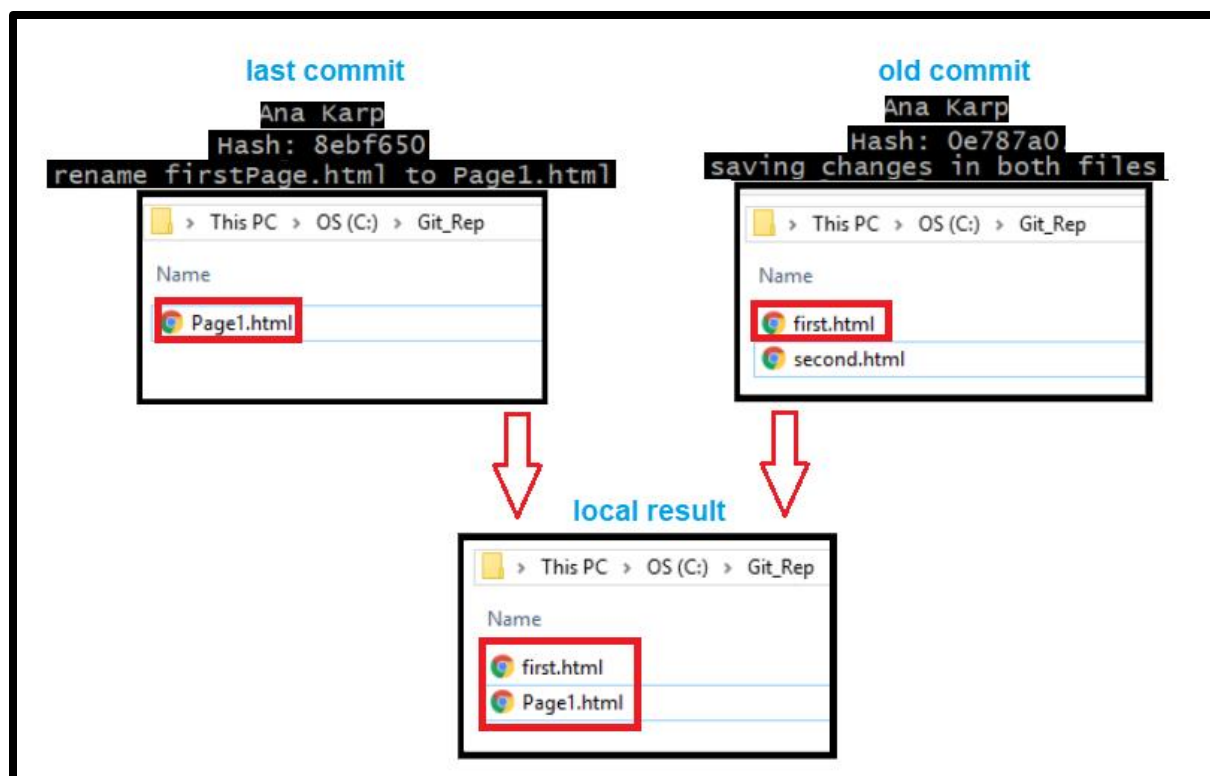
6. Branches

Branches gives us four core elements to work with:

- The Working Directory
- The Staged Snapshot
- Committed Snapshots
- Development Branches

In Git, a branch is an independent line of development.

For example, if you wanted to experiment with a new idea without using Git, you might copy all your project files into another directory and start making your changes. If you liked the result, you could copy the affected files back into the original project. Otherwise, you would simply delete the entire experiment and forget about it.

This is the exact functionality offered by git:

- Branch is essentially an independent line of development. You can take advantage of branch when working on new features or bug fixes as it helps to isolate your work from that of other team members.
- Different branches can be merged into any one branch if they belong to the same repository.
- Branching enables you to isolate your work from others. Changes in the primary branch or other branches will not affect your branch, unless you decide to pull the latest changes from those branches.
- Upon making the first commit in a repository, Git will automatically create a master branch by default. Subsequent commits will go under the master branch until you decide to create and switch over to another branch.

Git branches key improvements:

- branches present an error-proof method for incorporating changes from an experiment
- branches let you store all your experiments in a single directory, which makes it much easier to keep track of them and to share them with others.
- branches lend themselves to several standardized workflows for both individual and collaborative development.

There are two types of branches:

1. **Integration branch** - should be always kept stable. This is important because new branches will be created off from this branch, or perhaps this is the branch that will go out live on production.
   When some changes need to be merged into the Integration branch, it is generally a good idea to create a Topic branch to work on independently.

   ➢ "master" is usually used as integration branch.

2. **A Topic branch** - represents a specific task (ie. new feature, bug fixing). Topic branch is typically created off from an Integration branch. A completed Topic branch will eventually be merged back into the Integration branch.

6.1 Head

HEAD is used by Git to represent the current snapshot/position of a branch. For a new repository, Git will by default point HEAD to the master branch. Changing where HEAD is pointing to will updating your current active branch.
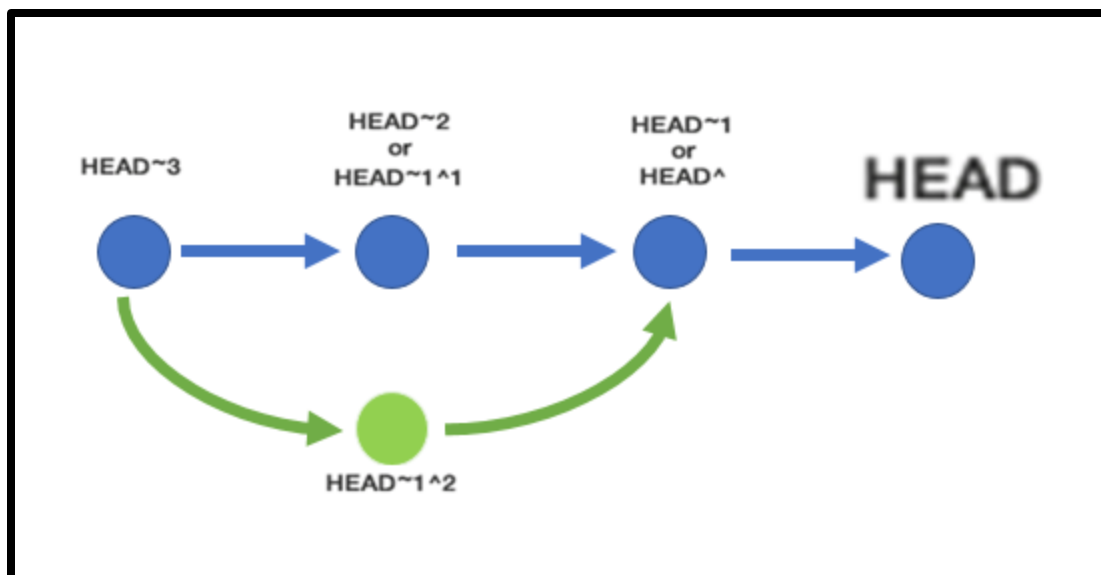
The ~(tilde) and ^(caret) symbols are used to point to a position relative to a specific commit. The symbols are used together with a commit reference, typically HEAD or a commit hash.

For instance, ~<n> refers to the <n>th grandparent. HEAD~1 refers to the commit's first parent. HEAD~2 refers to the first parent of the commit's first parent.

^<n> refers to the <n>th parent. HEAD^1 refers to the commit's first parent. HEAD^2 refers to the commit's second parent. A commit can have two parents in a merge commit.

6.2   View Existing Branches

Let's start our exploration by listing the existing branches for our project with the **git branch** command:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git branch
* master
```

This will display our one and only branch: * master.

The **master** branch is Git's default branch, and the asterisk next to it tells us that it's currently checked out. This means that the most recent snapshot in the **master** branch resides in the working directory:
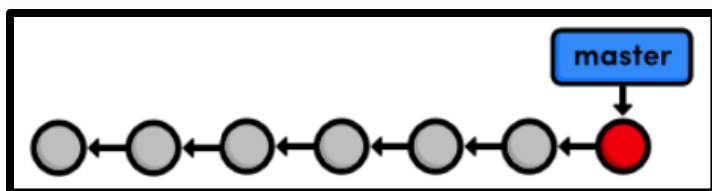
Notice that since there's only one local working directory for each project, only one branch can be checked out at a time.

let's view our history:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git log --pretty=format:"%an. Hash: %h. Msg: %s"
Ana Karp. Hash: 8ebf650. Msg: rename firstPage.html to Page1.html
Ana Karp. Hash: 0c50f5e. Msg: rename first.html to firstPage.html
Ana Karp. Hash: 4ec1797. Msg: removed second.html
Ana Karp. Hash: 0e787a0. Msg: saving changes in both files
Ana Karp. Hash: f65fe37. Msg: saving changes in the first page
Ana Karp. Hash: de0f535. Msg: this commit added the second page
Ana Karp. Hash: b63c6ad. Msg: this is the first step
```

and move to commit 0c50f5e:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git checkout 0c50f5e
Note: checking out '0c50f5e'.
```

The output of checking the branch:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep ((0c50f5e...))
$ git branch
* (HEAD detached at 0c50f5e)
  master
```

The red circle in each of our history diagrams represents Git's HEAD. The following

figure shows the state of our repository before and after we checked out an old commit:
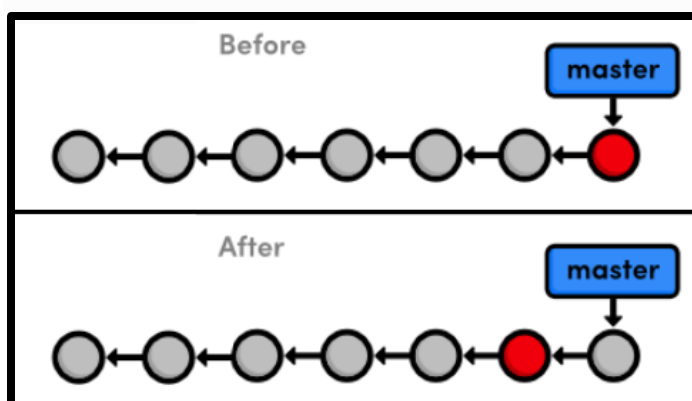
As shown in the "before" diagram, the HEAD normally resides on the tip of a development branch. But when we checked out the previous commit, the HEAD moved to the middle of the branch. We can no longer say we're on the master branch since it contains more recent snapshots than the HEAD.

6.3   Create a New Branch

We can't add new commits when we're not on a branch, so let's create one now. This

will take our current working directory and fork it into a new branch.

New branch is created with the **git branch <BRANCH_NAME>** command:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep ((0c50f5e...))
$ git branch branch1
```

Note that **git branch** is a versatile command that can be used to either list branches or create them. However, the above command only creates the branch—it doesn't check it out. So, we need to checkout:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep ((0c50f5e...))
$ git checkout branch1
Switched to branch 'branch1'

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (branch1)
$
```

We're now free to experiment in the working directory without disturbing anything in the **master** branch. The **branch1** branch is a completely isolated development environment that can be visualized as the following.
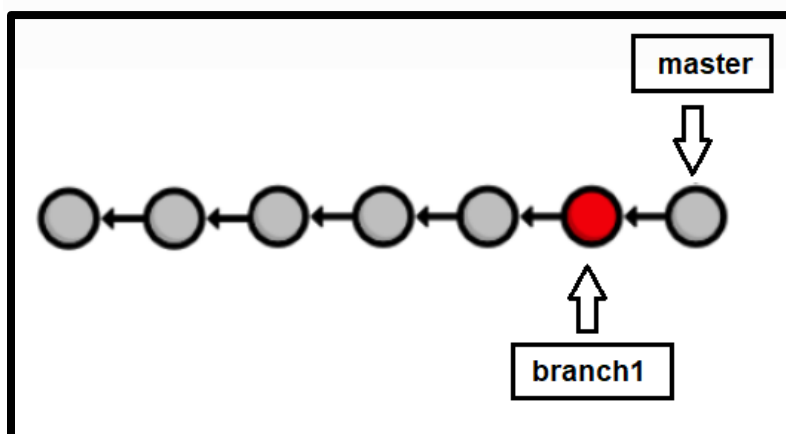
Right now, the **branch1** branch, **HEAD**, and **working-directory** are the exact same as the sixth commit. But as soon as we add another snapshot, we'll see a fork in our project history.

6.4 Stage and Commit Branch

We'll continue developing our **Page1.html** from the current:

```html
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <h1>first line</h1>
    <p>change in the first page - again</p>
</body>
</html>
```

To the following:

```html
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <h1>first line</h1>
    <p>change in the first page - branch1</p>
</body>
</html>
```

Now, we will stage and commit the changes:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (branch1)
$ git add .

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (branch1)
$ git status
On branch branch1
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   firstPage.html
```

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (branch1)
$ git commit -m "changed firstPage content"
[branch1 f5d5444] changed firstPage content
 1 file changed, 6 insertions(+), 2 deletions(-)
```

After committing on the **branch1** branch, we can see two independent lines of development in our project:
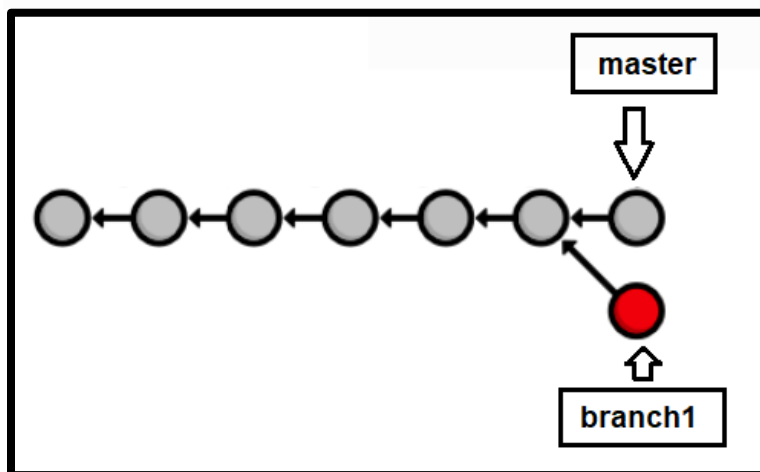
Also notice that the **HEAD** (designated by the red circle) automatically moved forward to the new commit, which is intuitively what we would expect when developing a project.

The above diagram represents the complete state of our repository, but **git log** only displays the history of the current branch:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (branch1)
$ git log --pretty=format:"%an. Hash: %h. Msg: %s"
Ana Karp. Hash: f5d5444. Msg: changed firstPage content
Ana Karp. Hash: 0c50f5e. Msg: rename first.html to firstPage.html
Ana Karp. Hash: 4ec1797. Msg: removed second.html
Ana Karp. Hash: 0e787a0. Msg: saving changes in both files
Ana Karp. Hash: f65fe37. Msg: saving changes in the first page
Ana Karp. Hash: de0f535. Msg: this commit added the second page
Ana Karp. Hash: b63c6ad. Msg: this is the first step
```

Note that the history before the fork is considered part of the new branch. That is to say, the **branch1** history spans all the way back to the first commit:
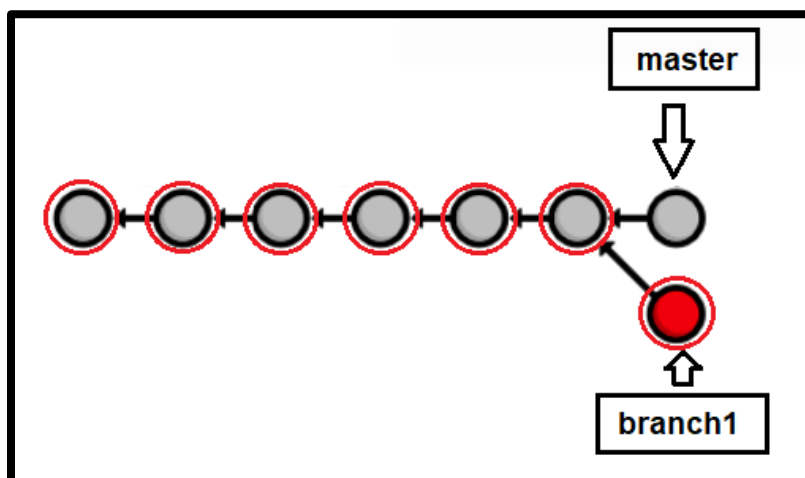
The project now has a complex history. however, each individual branch still has a linear history (snapshots occur one after another).

6.5   Create branch and checkout with one command

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (master)
$ git checkout -b branch2
Switched to a new branch 'branch2'

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep (branch2)
$ git branch
  branch1
* branch2
  master
```

7. Git repository to GitHub

Step 1 - create a new repository in your GitHub account:

Step 2 - copy your repository url:
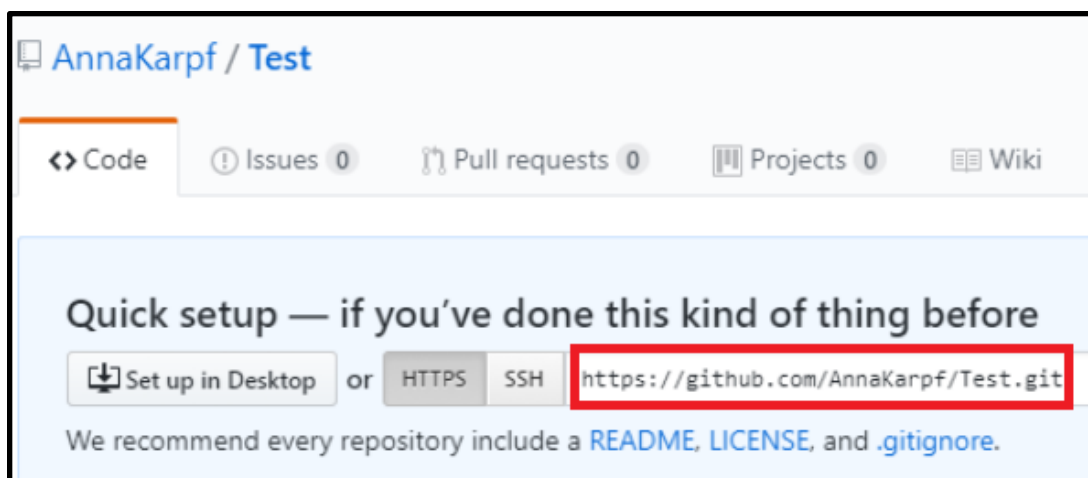
Step 3 - remote your repository with the git bash:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /
$ cd /c/Git_Rep

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep ((0e787a0...))
$ git remote add origin https://github.com/AnnaKarpf/Test.git

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep ((0e787a0...))
$ git remote
origin

Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep ((0e787a0...))
$ |
```

Step 4 - push local repository to the GitHub repository:

```
Anna@DESKTOP-RLP7NAJ MINGW64 /c/Git_Rep ((0e787a0...))
$ git push -u origin master
Counting objects: 19, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (19/19), 1.92 KiB | 392.00 KiB/s, done.
Total 19 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), done.
To https://github.com/AnnaKarpf/Test.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Step 5 - check if the change is visible in the GitHub repository:
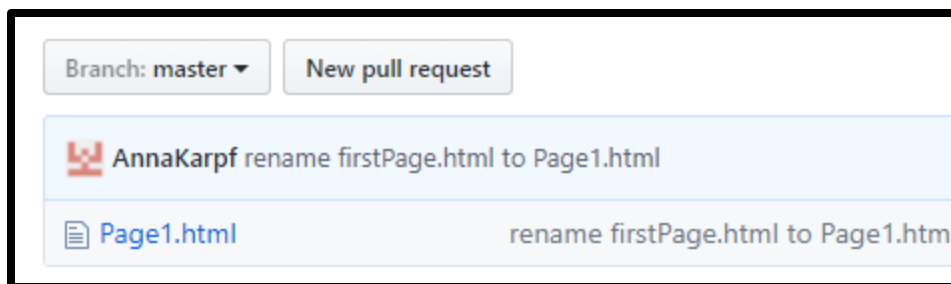
8. Git repository from GitHub

Step 1 - move to the folder that you want to add the git repository to:

Step 2 - clone the git repository with the url:

```
Anna@DESKTOP-RLP7NAJ MINGW64 ~/Desktop/Test
$ git clone https://github.com/AnnaKarpf/Test.git
Cloning into 'Test'...
remote: Counting objects: 19, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 19 (delta 4), reused 19 (delta 4), pack-reused 0
Unpacking objects: 100% (19/19), done.
```

Step 3 - you are done!!

> Test > Test

Name

⬭ Page1.html
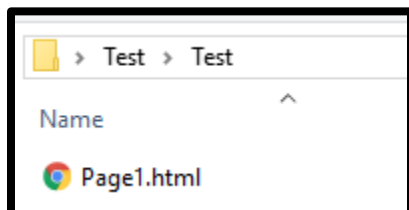
Appendix 1 – Git storage mechanism

In simplified form, git object storage is "just" a DAG of objects, with a handful of different types of objects.

They are all stored compressed and identified by an SHA-1 hash (that, incidentally, *isn't* the SHA-1 of the contents of the file they represent, but of their representation in git).

blob - the simplest object, just a bunch of bytes. This is often a file, but can be a symlink or pretty much anything else. The object that points to the `blob` determines the semantics.
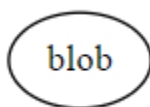
tree - Directories are represented by `tree` object. They refer to `blob`s that have the contents of files (filename, access mode, etc is all stored in the `tree`), and to other `tree`s for subdirectories.

When a node points to another node in the DAG, it *depends* on the other node: it cannot exist without it. Nodes that nothing points to can be garbage collected with `git gc`, or rescued much like filesystem inodes with no filenames pointing to them with `git fsck --lost-found`.
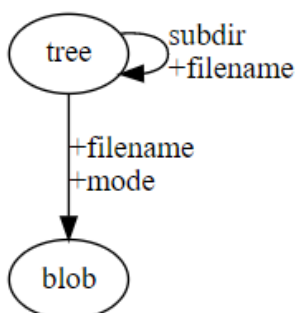
commit - A `commit` refers to a `tree` that represents the state of the files at the time of the commit. It also refers to 0..n other `commit`s that are its parents. More than one parent means the commit is a merge, no parents mean it is an initial commit, and interestingly there can be more than one initial commit; this usually means two separate projects merged. The body of the `commit` object is the commit message.
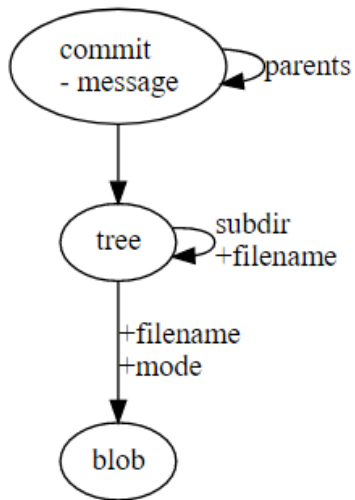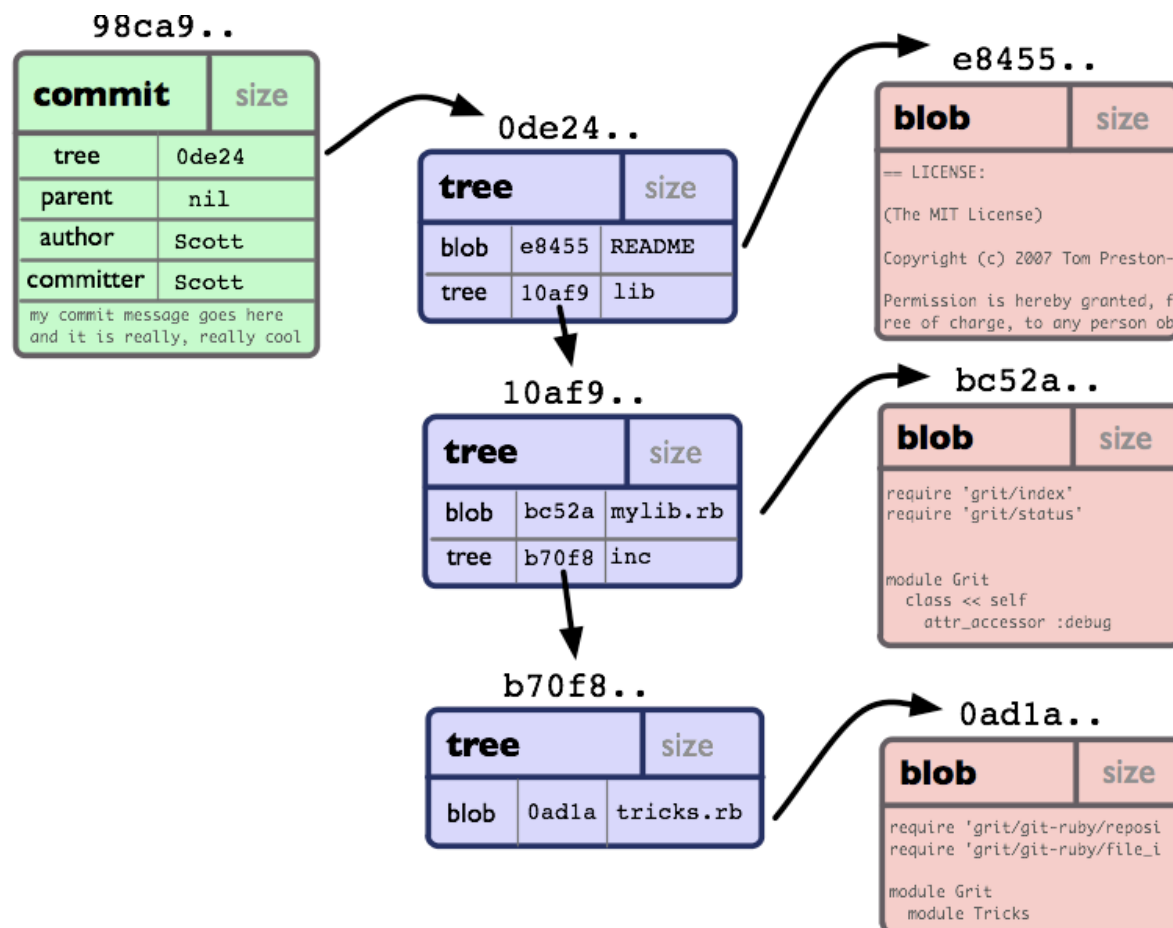
Diagram of commits + trees + blobs:

refs - References, or heads or branches, are like post-it notes slapped on a node in the DAG. Whereas the DAG only gets added to and existing nodes cannot be mutated, the post-its can be moved around freely. They don't get stored in the history, and they aren't directly transferred between repositories. They act as sort of bookmarks, "I'm working here".

`git commit` adds a node to the DAG and moves the post-it note for current branch to this new node.

The `HEAD` ref is special in that it points to another ref. It is a pointer to the currently active branch. Normal refs are in a namespace `heads/XXX`, but you can often skip the `heads/` part.
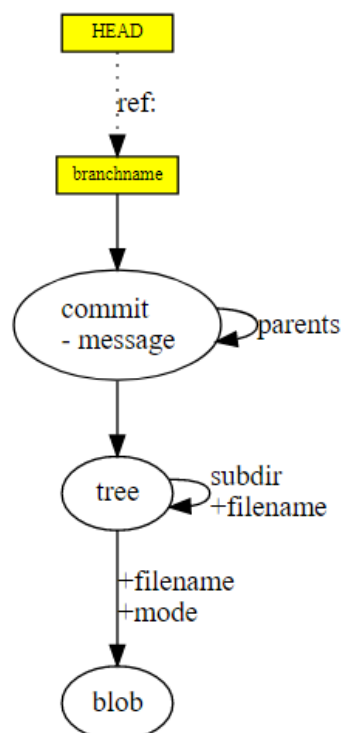
remote refs - Remote references are in different namespace from the normal `refs`, and the fact that remote refs are essentially controlled by the remote server. `git fetch` updates them.
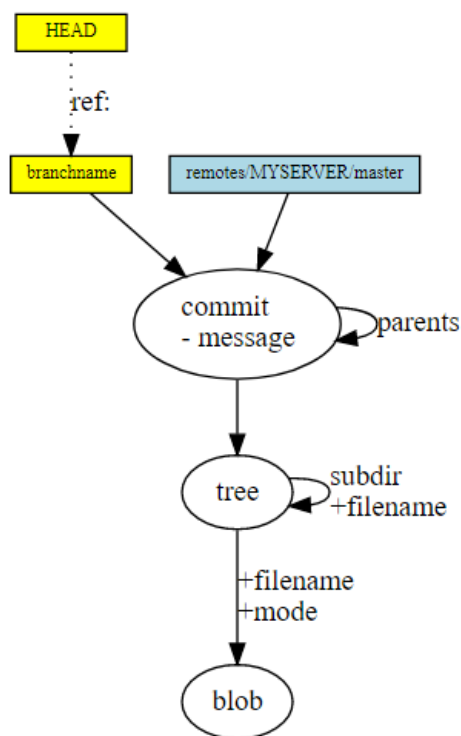
tag - A `tag` is both a node in the DAG and a post-it note. A `tag` points to a `commit`, and includes an optional message and a GPG signature.

The post-it is just a fast way to access the tag, and if lost can be recovered from just the DAG with `git fsck --lost-found`.

The nodes in the DAG can be moved from repository to repository, can be stored in more effective form (packs), and unused nodes can be garbage collected. But in the end, a `git` repository is always just a DAG and post-its.
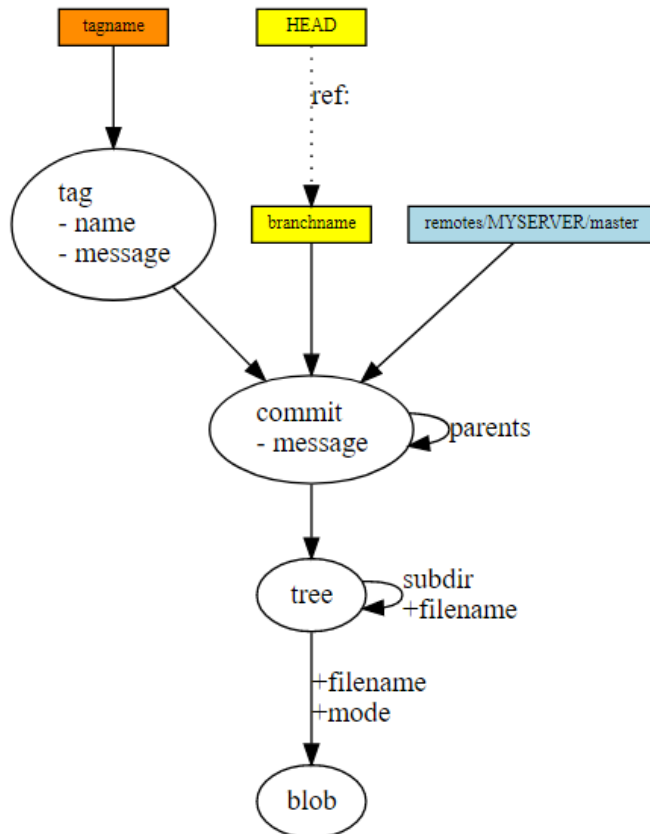
Diagram of tags+ heads + commits + trees + blobs:
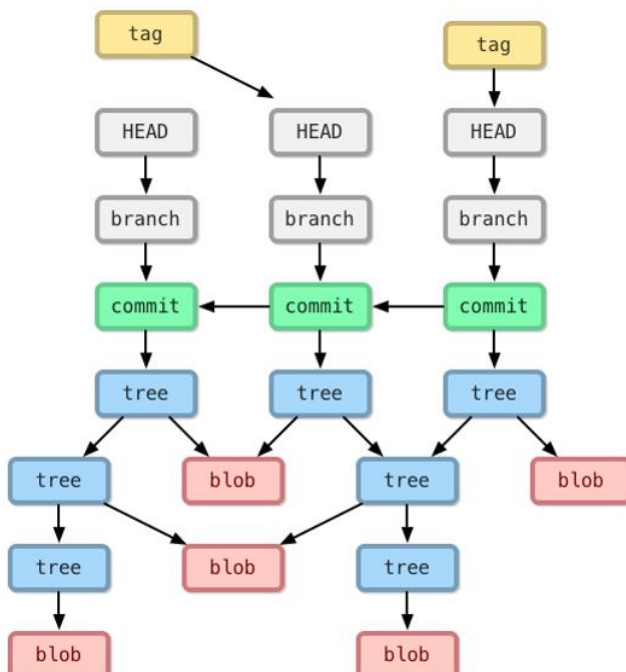
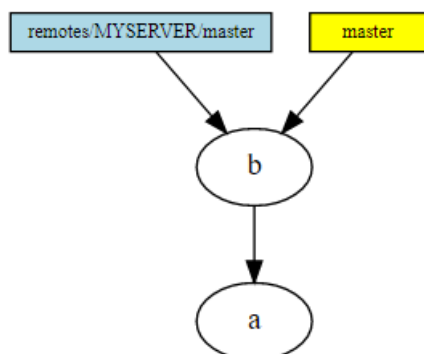# Appendix 2 – git history tracking

how do we visualize things like merges, and how does `git` differ from tools that try to manage history as linear changes per branch?
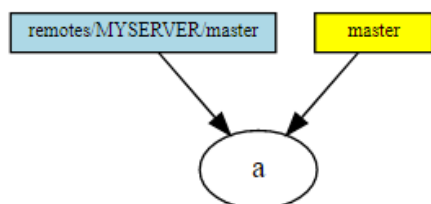

This is the simplest repository. We have `cloned` a remote repository with one commit in it.
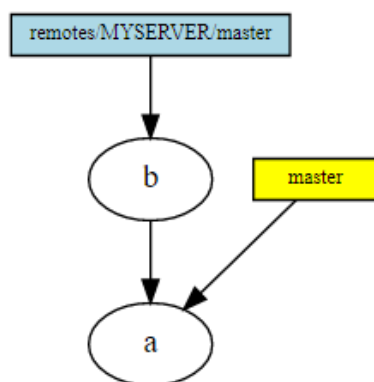

Here we have `fetched` the remote and received one new commit from the remote, but have not merged it yet.

The situation after `git merge remotes/MYSERVER/master`. As the merge was a `fast forward` (that is, we had no new commits in our local branch), the only thing that happened was moving our post-it note and changing the files in our working directory respectively.
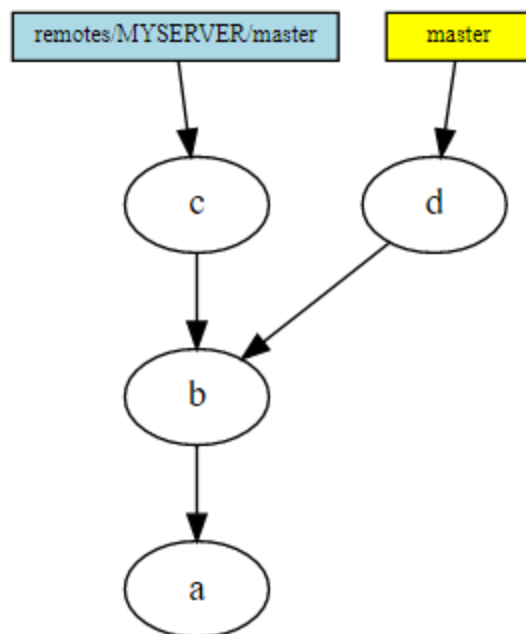
One local `git commit` and a `git fetch` later. We have both a new local commit and a new remote commit. Clearly, a merge is needed.
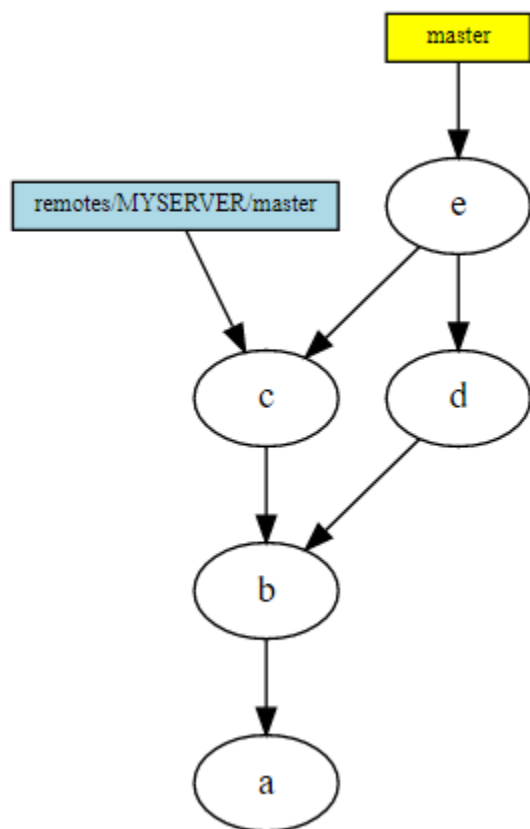
Results of `git merge remotes/MYSERVER/master`. Because we had new local commits, this wasn't a `fast forward`, but an actual new `commit` node was created in the DAG. Note how it has two parent `commit`s.
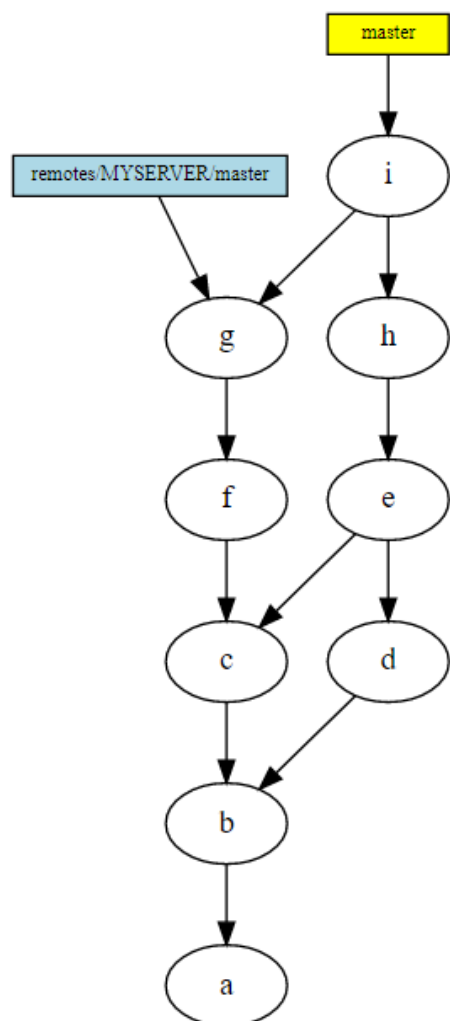

Here's what the tree will look after a few commits on both branches and another merge. See the "stitching" pattern emerge? The `git` DAG records exactly what the history of actions taken was.
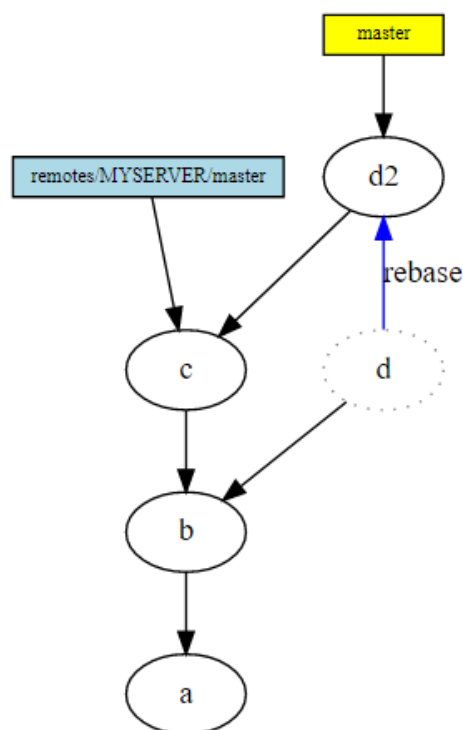
The "stitching" pattern is somewhat tedious to read. If you have not yet published your branch, or have clearly communicated that other should not base their work on it, you have an alternative. You can `rebase` your branch, where instead of merging, your commit is replaced by another commit with a different parent, and your branch is moved there.

Your old commit(s) will remain in the DAG until garbage collected. Ignore them for now, but just know there's a way out if you screwed up totally. If you have extra post-its pointing to your old commit, they will remain pointing to it, and keep your old commit alive indefinitely. That can be confusing, though.

Don't rebase branches that others have created new commits on top of. It is possible to recover from that, it's not hard, but the extra work needed can be frustrating.
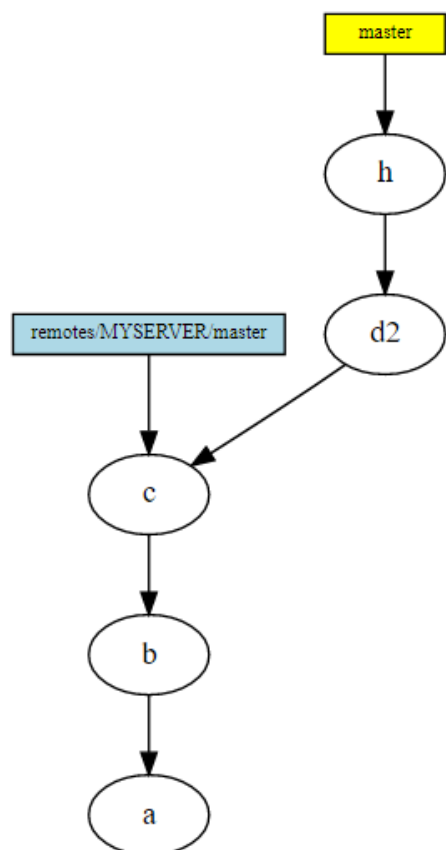
The situation after garbage collecting (or just ignoring the unreachable commit), and creating a new commit on top of your `rebased` branch.
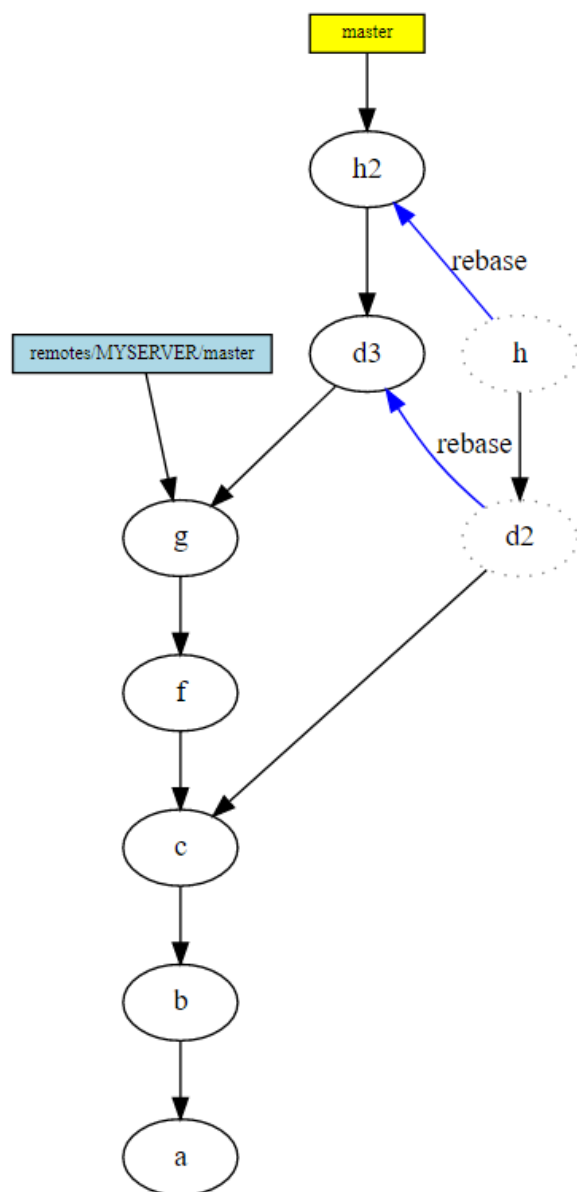
`rebase` also knows how to rebase multiple commits with one command.