**Source code management notes**

(Study from book)

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

committed, modified, and staged.

**Committed:**

The data is safely stored in your local database.

**Modified** :

We have changed the file but have not committed it to your database yet. **Staged:**

We have marked a modified file in its current version to go into

our next commit snapshot.

Three main sections of a Git project:

Git directory,working directory, staging area

If a particular version of a file is in the Git directory, it’s considered committed.

If it has been modified and was added to the staging area, it is staged.

If it was changed since it was checked out but has not been staged, it is modified.

To get a copy of an existing Git repository : **git clone**

Each file in our 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 our working directory that were not in our last snapshot and are not in your staging area.

When we first clone a repository, all of our files will be tracked and unmodified because Git just checked them out

and we haven’t edited anything.

As we edit files, Git sees them as modified, because we’ve changed them

since our last commit. We stage these modified files and then commit all our staged changes.

staged :“Changes to be committed” heading

git init then ran git add (files) – >begin tracking files in your directory.

The git add command takes a path name for

either a file or a directory

“Changes not staged for commit” – a file that is tracked has been modified

in the working directory but not yet staged.

To stage it, we run the git add command.

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

**Git basics:**

We can get a Git project using **two main approaches**.

1. Take an existing project or directory and import it into Git.
2. Clone an existing Git repository from another server.

**Commands**

1)**echo** 'My Project' > README

The above line will create a ReadME file with My Project written in it

2)After new file created

**git status**:will show untracked file

3)**git add filename** :tracking started but not allowed on master branch in company

4)first create a branch and only then make changes

5)go to my folder then create branch :

**git checkout -b mybranch**

6)If on new branch,new file create then add content then git add filename.Run git status->now our file is now tracked and staged to be committed:

We can say that it’s staged because it’s under the “Changes to be committed” heading.

7)**Short Status**

git status -s or git status --short

* new files that have been added to the staging area have an A,
* modified files have an M

For files we don’t want Git to automatically add or even show us as being untracked. These are generally automatically generated files such as log files or files produced by your build system.

In such cases,we can create a file listing patterns to match them named .gitignore.

**example**

**In** .gitignore file:

$ cat .gitignore

\*.[oa]: ignore any files ending in “.o”(object file) or “.a”(archive file

\*~:ignore all files whose names end with a tilde (~)(temporary files)

**To know exactly**

**what you changed, not just which files were changed :**

**git diff :** shows you the exact lines added and removed

answer these two questions:

1. What have you changed but not yet staged?
2. what have you staged that you are about to commit?

**git diff --cached** :to see what we’ve staged so far (--staged and

--cached are synonyms):

**Committing Your Changes**

Anything that is still unstaged – any files we have created or modified that we haven’t run git add on since we edited them –

won’t go into this commit. They will stay as modified files on our disk.

run git status, check that everything is staged .Now commit:

**git commit**

Removing Files

**Tracked files:in staging area**

1)To remove a file from Git, remove it from tracked files

remove it from staging area) and then commit.

2)git rm command :removes the file from our working directory so

we don’t see it as an untracked file the next time around.

If we simply remove the file from our working directory, it shows up under

the “Changed but not updated” (that is, unstaged) area of our git status

Then, if you run git rm, it stages the file’s removal:

$ git rm PROJECTS.md

rm 'PROJECTS.md'

$ 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)

deleted: PROJECTS.md

If we want to rename a file in Git, we can run something like:

$ git mv file\_from file\_to

**Viewing commit history**

1)git log

2)for last two entries :git log -p -2

3)to see some abbreviated stats for each commit

$ git log --stat

the --stat option prints below each commit entry a list of

modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

4) changes the log output to formats other than the default.

git log --pretty=oneline

The author is the person who originally wrote the work, whereas the committer is the person who last applied the work

**Undoing Things**

We can’t always undo some of these undos.

$ git commit --amend

It overwrites our previous commit.

If we commit and then realize we forgot to stage the changes in a file we wanted to add to this commit:

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit --amend

**Unstaging a Staged File**

If we’ve changed two files and want to commit them as

two separate changes, but we accidentally type git add \* and stage them

both.

Now unstaging one of the two:use git reset filename

$ git add \*

$ git status

On branch master

Changes to be committed:

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

renamed: README.md -> README

modified: CONTRIBUTING.md

**Do**

$ git reset HEAD CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

renamed: README.md -> 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

The CONTRIBUTING.md file is modified but once again unstaged

**While git reset can be a dangerous command if you call it with –hard**

**Unmodifying a Modified File**

unmodify it – revert it back to what it looked

like when we last committed

git checkout -- CONTRIBUTING.md

$ git status

We can see that the changes have been reverted.

Remote repositories are versions of your project that

are hosted on the Internet or network somewhere.

**origin** – that is the default name Git gives to the server we cloned from

We can also specify -v, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

git remote -v origin url

**The clone command implicitly adds the origin remote for you.**

To add a new remote explicitly.

To add a new remote Git repository as a shortname we can reference

easily,

run git remote add <shortname> <url>

$ git remote add pb https://github.com/paulboone/ticgit

$ git remote -v

When you have your project at a point that you want to share, you have to push it upstream.

git push [remote-name] [branch-name].

If you want to push your master branch to your origin server

(cloning generally sets up both of those names for you automatically),

To push any commits you’ve done back up to the server:

$ git push origin master

This command works only if you cloned from a server to which you have

write access and if nobody has pushed in the meantime.

If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected.

You’ll have to fetch their work first and incorporate it into yours before you’ll be allowed to push.

**Inspecting a Remote**

To see more information about a particular remote:

git remote show [remote-name]

ex:git remote show origin

**Removing and Renaming Remotes**

To rename pb to paul:

$ git remote rename pb paul

To remove a remote for some reason :

$ git remote rm paul

To set up an alias for each command using **git config.**

$ git config --global alias.ci commit

Instead of typing git commit: git ci

A branch : movable pointer to one of the commits.

The default branch name in Git is master.

Master branch is given that points to the last commit you made.

Every time you commit, it moves forward automatically.

Doing so creates a new pointer for you to move around.

Let’s say we create a new branch called testing.

$ git branch testing

This creates a new pointer to the same commit you’re currently on

HEAD:pointer to the local branch you’re currently on. In this case, you’re still on master.

The git branch command only created a new branch – it didn’t switch to that branch

git log:shows where the branch pointers are pointing.

Switch to the new testing branch:

$ git checkout testing

This moves HEAD to point to the testing branch.

Because a branch in Git is in actuality a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy

**$ git checkout -b iss53**

**Switched to a new branch "iss53"**

**This is shorthand for:**

**$ git branch iss53**

**$ git checkout iss53**

Switch back to your master branch.

However, before you do that, note that if your working directory or staging

area has uncommitted changes that conflict with the branch you’re checking out, Git won’t let you switch branches.

when you try to merge one commit with a commit that can be reached by

following the first commit’s history, Git simplifies things by moving the pointer

forward because there is no divergent work to merge together – this is called a

“fast-forward.”

check out the branch you wish to merge into and then run the git

merge command:

$ git checkout master

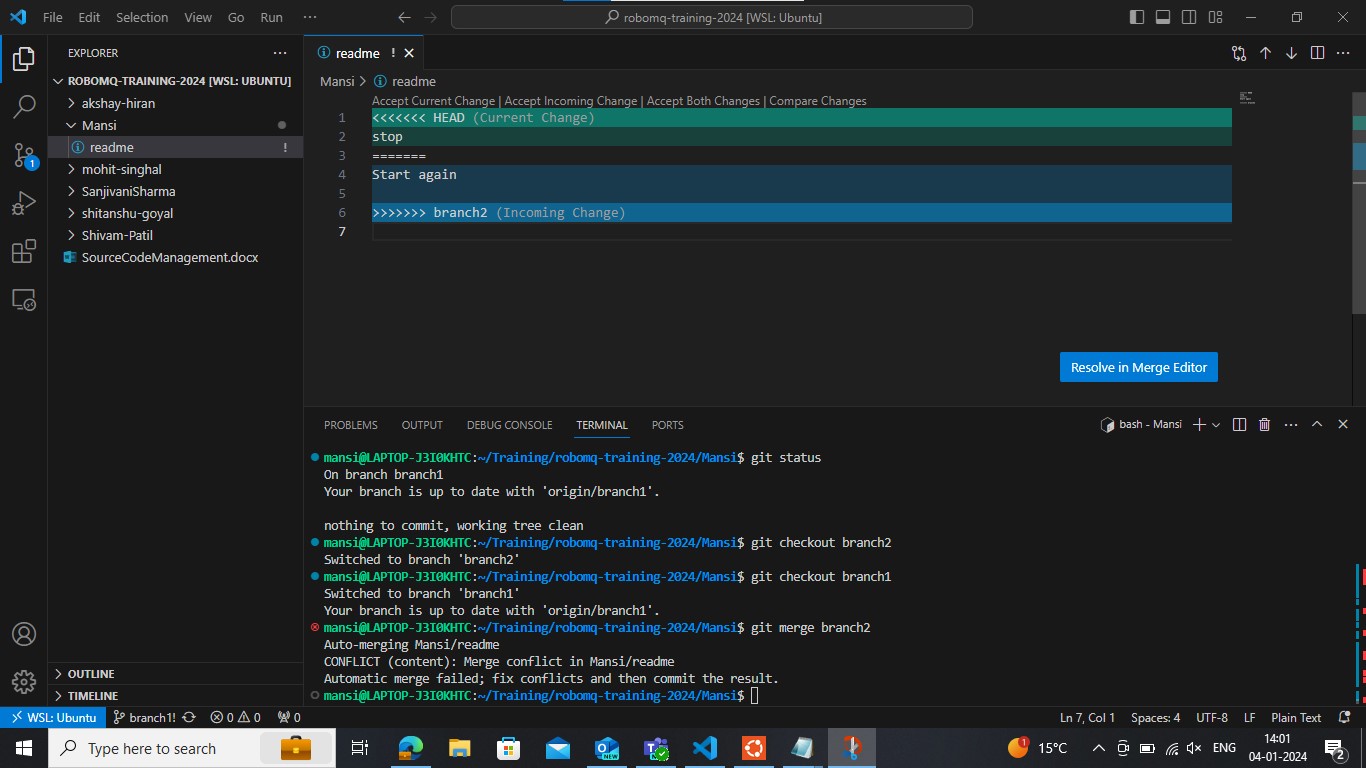
Switched to branch 'master'

$ git merge iss53

**Basic Merge Conflicts**

If you changed the same part of the same file differently in the two branches you’re merging together, Git won’t be able to merge them cleanly

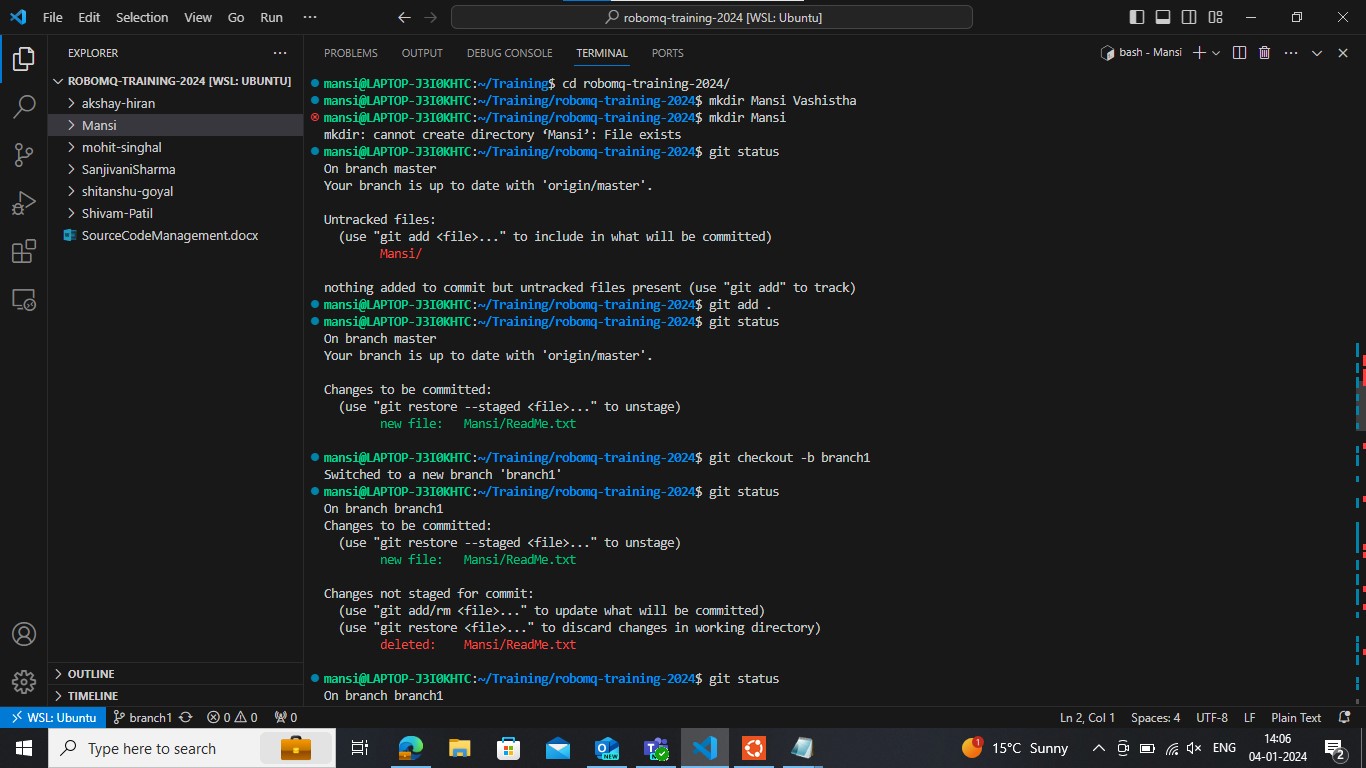
Anything that has merge conflicts and hasn’t been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts.



Resolving merge conflict

A screenshot of a computer program

Description automatically generated

Screenshots of git commands:  


A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated