GUIDE TO MAKE A REAL CONTRIBUTION TO AN OPEN SOURCE PROJECT
WHO AM I?

@tushar_rishav

- GSoC'16 student contributing to coala - a static code analysis tool, under Python Software Foundation.
- A senior year, pursuing B.Tech from the National Institute of Technology, Trichy, India.
- More at www.gtushar.co
LET'S GET STARTED

http://tinyurl.com/epworkshop2016
http://tinyurl.com/epwgist
VERSION CONTROL SYSTEMS
WHAT?

• A software tool that *manages changes* to source code over time.
HOW?

- Keeps track of every changes using simple database and hence the changes can be easily reverted if required. So don't panic if your colleague messed up!
WHY?

- Enables collaborative programming (we will do shortly) with minimum disruptions.
- Developers can work on unrelated features or changes and at the end get changes merged back together.
CENTRALISED VS DISTRIBUTED VCS
A single server that contains all the versioned files, and a number of clients that check out working copy from that central place. Eg: Subversion.
MAJOR DRAWBACK

- **Temporary failure**: Being centralised, if the server goes down for a certain period of time, then nobody can collaborate at all.
- **Permanent failure**: If proper backups haven’t been kept and the central database becomes corrupt, then we lose the entire history of the project.

Having the entire history of the project in a single place, we risk losing everything.
DISTRIBUTED VCS

- Developers fully mirror the repository.
Every clone is really a full backup of all the data.
If the server is down or corrupted, then any of the client repositories can be copied back up to the server to restore it.
Examples are Mercurial and Git.
git

Created by Linus Torvalds and the Linux development community.
FUNDAMENTALS

- Stores data as a set of snapshots of a miniature file systems.
- Nearly every operation is local. One can work offline and requires network connection while pushing/fetching the changes to/from the remote server only.
FUNDAMENTALS

WORKFLOW

1. Working Area
2. Staging Area
3. Committed Area
FUNDAMENTALS

WORKFLOW

- **Working Area:**
  Have changed the file but have not committed it to your database yet.

- **Staging Area:**
  Have marked a modified file in its current version to go into your next commit snapshot.

- **Committed Area:**
  The data is safely stored in your local database.
~/git> commands
Git Commands

**git configs**

$ git help  # Your git manual page.
$ git config --global user.name "Tushar Gautam"
$ git config --global user.email tushar.rishav@gmail.com

- Important as every git commit uses this information.
- Run without --global to override for a particular project.

Default text editor for git. In my case vim

$ git config --global core.editor vim

See complete list

$ git config --list
Git Commands

- **Inititialise a repository in a given directory.**
  
  ```
  git init <directory>
  # leaving `directory` will initialis a repo
  # in the current directory.
  ```

- **Cloning an existing repository.**

  ```
  git clone <repo-url> [<target-directory>]
  # creates a directory called "coala".
  git clone https://github.com/coala-analyzer/coala.git
  # creates a directory called "my-directory"
  git clone https://github.com/coala-analyzer/coala.git my-directory
  ```
FILE STATUS

- **Tracked**: files that were in the last snapshot; they can be unmodified, modified, or staged.
- **Untracked**: files in your working directory that were not in your last snapshot and are not in your staging area.
File status

$ git add <file-name>  # Add `file-name` to staging area.
$ git status  # check status for files.
$ git status -s  # same as above but less verbose.

• If we modify any file in staging area, we need to add it again otherwise Git stages a file exactly as it is when you run the git add command.
• .gitignore can be used to avoid Git to automatically add or even show you as being untracked.
File status

$ git diff  # To see what you’ve changed but not yet staged,
$ git diff --staged
$ git diff --cached    # same as above
# To see what have been staged that will go into next commit,
# compares the staged changes to last commit.

- If we modify any file in staging area, we need to add it again otherwise Git stages a file exactly as it is when you run the git add command.
- .gitignore can be used to avoid Git to automatically add or even show you as being untracked.
Git Commit

$ git commit # Opens default editor with output from `git status`
$ git commit -v # Similar but displays `diff` for the changes.
$ git commit -m "commit message" # Inline.
$ git commit -am "commit message"
$ git show <commit_hash>
# default is recent commit.
# Skip staging area and commit all changed files. Careful!
$ git rm <file>
# remove `file` from staging area and the filesystem.
$ git rm --cached <file>
# remove from staging area only.

• Commits the staged snapshot to the project history. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to (covered later).
• Every time a commit is performed, we’re recording a snapshot of the project that we can revert to or compare to later.
Git Log or History

$ git log
# Lists all commits with most recent commits show up first.
$ git log -p
# Shows diffs introduced with each commit.
$ git log -p -3 # Show first 3 entries only.
$ git log --oneline # Less verbose.
$ git log --graph --oneline.
# Shows graphical structure for branch and merge history.
$ git log --help # for more options.
Make changes

$ git commit --amend
# Change the last commit message.
$ git checkout <commit_hash>
# checkout to the repo state wrt to the commit.
$ git checkout -- <file>
# Drop the changes for `file`. Careful!
Remote

- Remote repositories are versions of your project that are hosted on the Internet or network somewhere.
- Saved in `.git/config` file.

```bash
$ git remote -v
# List all remotes with their urls.
$ git remote add <name | alias> <url> # add remote.
$ git remote rename <old-name> <new-name> # rename remote.
$ git remote set-url <name> <url>
# change the url for a remote.
$ git remote rm <name> # remove remote.
$ git fetch <remote-name> # fetch from remote.
# ^ Only downloads the data and not merge or modifies.
$ git merge # merge the changes.
$ git pull # fetch + merge
$ git push <remote-name> <branch-name> # use `-f` to force push.
```
Workflow

1. Add remote.
2. Create branch.
3. Add/Make changes and commit.
4. Fetch from remote and rebase.
5. Push or force push.
6. Create a pull request.

We have talked about all except 6 and 7.

$ git fetch <remote-name>
$ git rebase -i <remote/branch-name>
$ git request-pull [-p] <start> <url> [<end>]
# Use git request-pull and a mailing list. eg. Linux kernel.
Branching

$ git branch  # display branch
$ git branch -a # all branch
$ git branch <new-branch>  # create a new branch.
$ git checkout <new-branch>  # Checkout to new branch.
$ git checkout -b <other-new-branch>  # Combines above two.
$ git checkout -D <brnach-name>
# Force delete branch with unmerged changes.
• **Merge:** Adds a merge commit.
• **Rebase:** Re-writes the project history by creating new commits. It adds the local commits on top of the

```bash
$ git merge
# merge changes from feature to master
$ git rebase
$ git rebase -i  # interactive rebase.
```
Cherry-pick

$ git cherry-pick <commit_hash>
$ git revert
$ git reset HEAD
# HEAD means current commit
$ git reset --hard HEAD
# Move branch pointer to certain commit state
REFERENCE

- Git Books - Scott Chacon
Let's start the contribution process :)