Version Control with Git

Jay Modi

github.com/mjrulesamrat
linkedin.com/in/mjrulesamrat

Objectives

- Introduction to GIT
- Hands on basic commands
- Know your resources - Official Documentations and take benefit of stackoverflow
- Best practices
- Git for Agile teams
- Resolve Merge/Rebase conflict notes

Which changes were made? Who made the changes? When were the changes made? Why were changes needed?

- Git (Distributed VCS)
- A version control system.
- Created by Linus Torvalds,
- creator of Linux, in 2005
- Goal was to Support for non-linear development(thousands of parallel branches)
- https://guides.github.com/introduction/git-handbook/

Git

- By far, the most widely used modern version control system.
- Git is a distributed version-control system for tracking changes in source code during software development.
- It is designed for coordinating work among programmers, but it can be used to track changes in any set of files.
- Available as service with GitHub, Bitbucket, GitLab etc.
- Wide range of GUI clients. Ex. SmartGit, Built-in with many editors like vscode, pycharm, sublime etc.
- Best known used among open-source software developers with Github.

Centralized VCS

- Subversion, CVS, Perforce, etc.
- A central server repository (repo) holds the "official copy" of the code - the server maintains the sole version history of the repo.
- You make "checkouts" of it to your local copy - You make local changes to the file and they are not versioned.
- Later you “Check-in” these changes of files back to the server and then they are versioned.

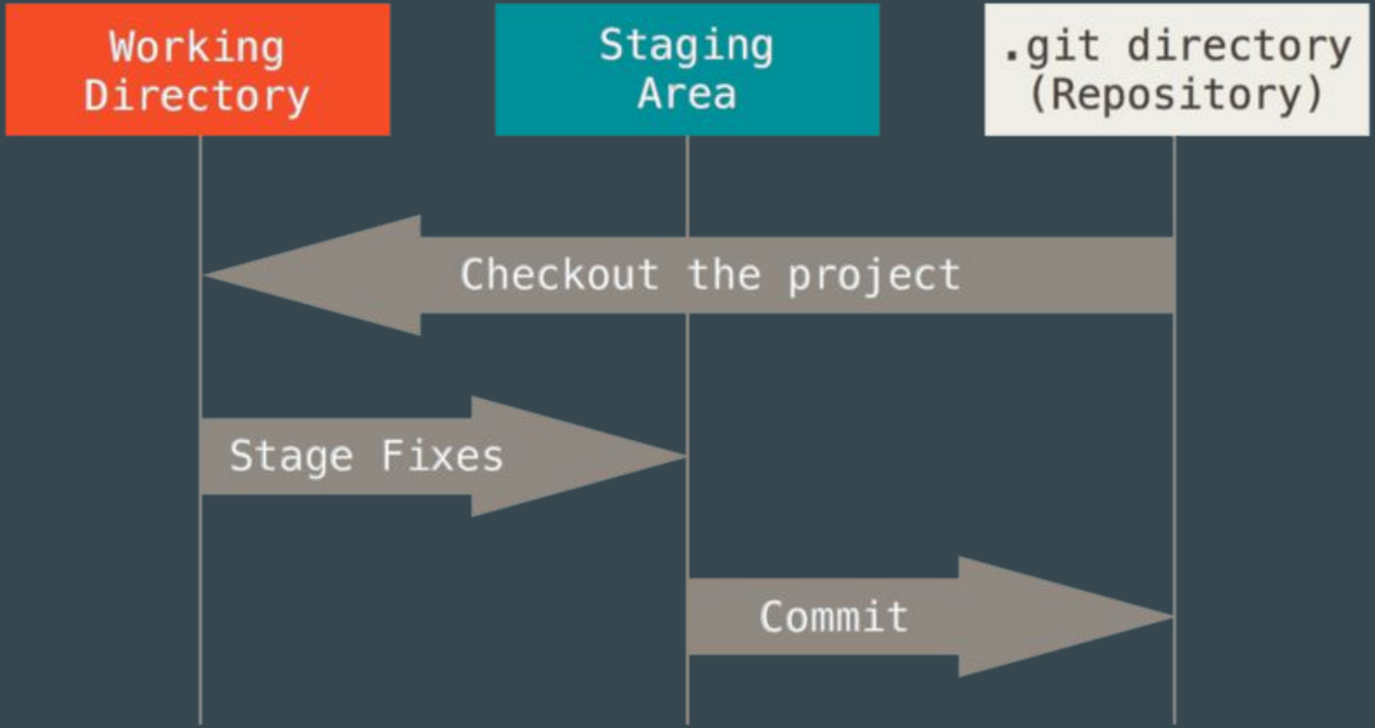
Distributed VCS(Git)

- In git, mercurial etc., you don't "checkout" from central repo, you "clone" it and "pull" changes from it.
- Your local repo is a complete copy of everything on the remote server.
- "yours" is just as good as "theirs"
- Many operations are local:
    - checkin/out from local repo
    - commit changes to local repo
    - Local repo keeps version history
- And when you're ready, you can "push" changes back to server.

Git Snapshots

- Central VCS tracks changes on the each individual file.
- Git keep “snapshots” of entire state of the project.
- Each check-in version of the overall code has a copy of a each file.
- More redundancy but faster.

Local git areas(wd, index, HEAD)



Git commit checksums

- In subversion each modification to the central repo increments version # of the overall repo.
- In Git, each user has their own copy of the repo, and commits changes to their local copy of the repo before pushing to the central server. Work anywhere!
- So git generates a unique SHA-1 hash (40 character string of hex digits) for every commit.
- Refers to commits by this ID rather than a version number.
- Often we only see the first 7 characters:
    - 1677b2d Edited first line of readme
    - 258efa7 Added line to readme

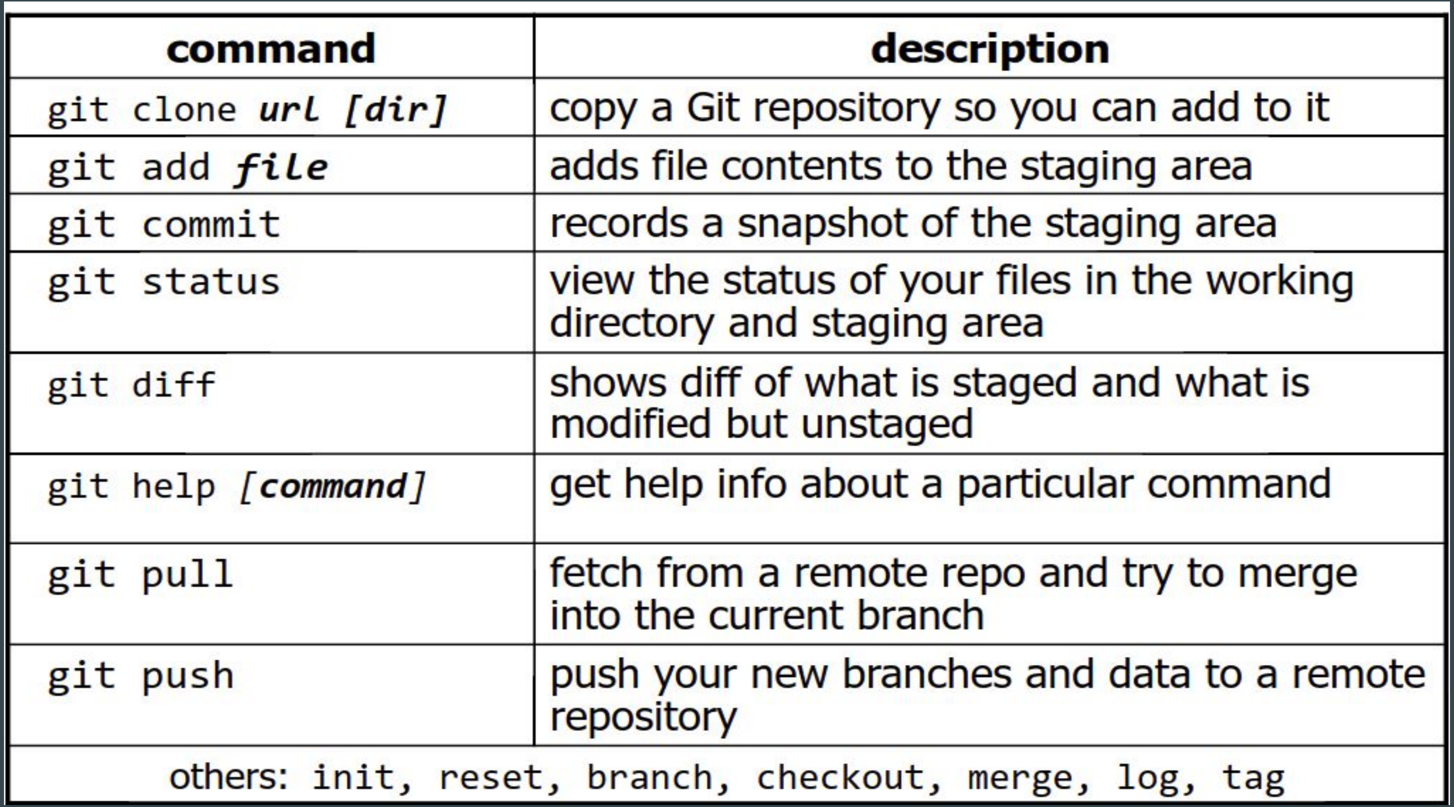
Initial Git configuration

- Set the name and email for Git to use when you commit:
  - git config --global user.name “Jay Modi”
  - git config --global user.email mjrulesamrat@gmail.com
- Set the editor that is used for writing commit messages:
  - git config --global core.editor.vim
  - (default is already set to vim, you can change to sublime, gedit, nano etc.)
- Use multiple SSH keys
  - https://gist.github.com/mjrulesamrat/d054630303563a3a286c3f996b9f132f

Creating a Git repo (Two common scenarios)

- To create a new local Git repository in your current directory
  - git init
    - This will create a .git directory in your current dir
    - Then you can commit files in the directory into the repo.
  - git add filename
  - git commit -m “commit message here”
- To clone a remote repo to your current directory
  - git clone url localDirectoryName(optional)
    - This will create given local directory, containing working copy of the files from the repo, and a .git directory (used to hold the staging area and your actual local repo)
- Don’t forget to add readme.md file

Git Basics(https://rogerdudler.github.io/git-guide/)



command	description
`git clone` *`url [dir]`*	copy a Git repository so you can add to it
`git add` *`file`*	adds file contents to the staging area
`git commit`	records a snapshot of the staging area
`git status`	view the status of your files in the working directory and staging area
`git diff`	shows diff of what is staged and what is modified but unstaged
`git help` *`[command]`*	get help info about a particular command
`git pull`	fetch from a remote repo and try to merge into the current branch
`git push`	push your new branches and data to a remote repository
others: `init`, `reset`, `branch`, `checkout`, `merge`, `log`, `tag`	

Some fine grained resources

1. https://git-scm.com/book/en/v2
2. https://schacon.github.io/git/gittutorial.html
3. https://shinglyu.com/web/2018/03/25/merge-pull-requests-without-merge-commits.html
4. https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf (GIT cheat sheet from github)
5. https://guides.github.com/introduction/git-handbook/

Git Best Practices

- https://github.com/trein/dev-best-practices/wiki/Git-Commit-Best-Practices (Our baby steps kind of best practices)
- https://chris.beams.io/posts/git-commit/ (Commit message Best practices, You’ve got to follow this. No Exception!)
- https://sethrobertson.github.io/GitBestPractices/ (Detailed best practices, I do this all the time)

1. Commit Related Changes
2. Commit Often
3. Don't Commit Half-Done Work
4. Test Your Code Before You Commit
5. Write Good Commit Messages
6. Use Branches
7. Agree on A Workflow

Git for Agile Team (probably best thing you'll learn today)

1. Pull to update your local master
2. Check out a feature branch
3. Do work in your feature branch, committing early and often
4. Rebase frequently to incorporate upstream changes
5. Interactive rebase (squash) your commits
6. Merge your changes with master
7. Push your changes to the upstream

Also, questions you need to ask yourself :D: http://justinhileman.info/article/git-pretty/git-pretty.png

http://reinh.com/blog/2009/03/02/a-git-workflow-for-agile-teams.html

Single most useful workflow, that I can find over internet and also the one I've incorporated in my daily programming routine. Just perfect!

Comes with the silent message "Danger". It will rewrite your git history. So, never do rebase in `master` branch. Always perform these operations in your feature branch.

But spiderman takes the risk, why not us!

Tips to fix conflict

While fixing ***merge*** conflicts:

1. Do `git status` to find those files which has conflicts
2. Head is theirs and other part is yours or commit will be specified
3. Fix conflicts
4. `git add` those files
5. Hit `git commit`. That will open up editor with merge commit. Save the file and exit.
6. Repeat 1 to 5 until merge completes.

You'll only need to push your changes to your remote branch after merge[conflicts].

While fixing ***rebase*** conflicts:

1. Do `git status` to find those files which has conflicts
2. Head is theirs and other part if yours or commit will be specified in the file
3. Fix conflicts
4. `git add` those files
5. Hit `git rebase --continue`.
6. Repeat 1 to 5 until all conflicts/rebase completes.

You'll need to *force* push changes to your remote branch after rebase. Because rebase rewrites your commit history.

You are awesome!
Thank You