Git Training - Part II

Eric Williams

Red Hat

Contributing Remotely

Overview

- For this session we'll be using GitHub to show a sample workflow
  - Not all projects in the office use GitHub but the concepts are similar
- This session will combine concepts from the last talk, as well as some new ones
- Please ensure you have a GitHub account set up and ready to go
  - No other GitHub experience needed

Remote Repositories

Cloning

- Copies an entire remote or local repository to a specified location

- *git clone <repo> <location>*
  - Repo can be a local repository, in which case just point *git clone* to the folder that contains the repo to be cloned
    Repo can also be a remote location, usually a URL pointing to a .git file
    - *git clone ssh://john@example.com/my-project.git*
  - Location can be blank, in which case the current directory is used

Branches

- In the previous session, we only dealt with local branches

- In the context of a remote repository, there are also remote branches
  - Remote branches are ones that exist in the origin
    - Origin is the name of the remote location after a clone operation
  - Remote branches are visible to all: when cloning a repo, remote branches will be cloned

Tracking Branches

- A tracking branch is a local branch that has a direct relationship to a remote branch

- When cloning a repo, git automatically creates a tracking branch (master) that tracks the remote master branch (origin/master)
  - Notation for branches: <location>/<branch_name>
  - For example: remote branches are origin/<branch_name>, the most common one is origin/master

Staying Up To Date

Overview

- Once a repo is cloned from a remote source, it won't update with remote changes automatically
  - This has to be done manually

- This can be accomplished with two commands
  - *git fetch*
  - *git pull*

Remotes

- Git has a concept of remotes, which allows a user to add multiple remote “destinations” for their repositories
- For example, consider someone has forked your repo and you’d like to fetch their changes
- Add their repo as a remote, and then fetch from their repo
  - We’ll go over this again during the GitHub PR workshop

Fetching

- *git fetch* is the most basic update command if you want to sync your local repo to the remote one
- Fetching only updates the remote branches in your repo, it does not touch any of the tracking branches
- This lets you take a look at the changes being made to the remote repo before merging them onto and of your branches

FETCH_HEAD

- Git provides a default head that matches *git fetch*
- FETCH_HEAD is a temporary reference created after a git fetch to show the tip of the remote branch after a *fetch* (before a *merge*)
  - FETCH_HEAD and HEAD will be the same after a merge operation with your tracking branch
- Usually only relevant if you have some conflicts to resolve manually before doing a *merge*
  - Also relevant if you want to do a code review/test

Pulling

- *git pull* simply combines *fetch* and *merge* into one operation
- Pulls all the latest updates and merges them with your local tracking branches
- Beware, if you have any conflicts this will put your repo into a merge conflict state
  - If you do most of your development on separate branches, this should not be an issue

Submitting Changes

Pushing

- *git push* is how you transfer commits from your local repository to a remote repo
- Opposite of a pull: updates the remote repo to the modifications you have made locally
- Think of this as running *git merge master* on the remote repo

GitHub's Workflow

- Every project will have its own workflow/review/submission process
- GitHub has its own as well, and today we'll go over how to submit to a GitHub project
- The crux of GitHub's workflow is the pull request

Pull Request Workflow

- Fork the repo into your GitHub account
- Clone your forked repo so you have a local copy
- Add the upstream project as a remote
- Create a branch, make your changes
- Push the branch to the origin project
- Create PR against upstream project using branch that was pushed to origin

Pull Request Workshop

Advanced Git

Bisecting

- *git bisect* enables you to hunt down issues in a repository
- For example: the tip of the master branch has a bug, but some commit in the past doesn't show the bug
- Which commit introduced the bug?
- If you have the commit that shows the bug, and some commit that doesn't, you can feed these two commit IDs to git and it will run a binary search via *git bisect*

Stashes

- *git stash* allows you to stash uncommitted changes for use at a later time
- Useful when you want to change branches in the middle of working on something, without having to commit
- The stashes can be applied at a later time

Clean

- *git clean* cleans up any unstaged changes in the repository
  - Can be used on files and directories

- Useful for cleaning up build files, temporary artifacts, etc.

- Configurable for different options depending on the circumstance

Blame

- *git blame* shows which user made the last edit to a line in a file
  - This is done by showing the commit that introduced the change

- Useful for debugging, but also winning petty arguments with co-workers

- Can be configured to show reverse history
  - For example: instead of showing the last edit to a line, show the earliest commit in which this line existed

Difftool

- Looking at a diff in the command line can be hard on the eyes
- Git is configurable to use other tools when doing a diff
- In this case we are going to use meld
- Instead of *git diff*, use *git difftool* <commits>
  - Will open meld for each applicable file, and show the differences

Questions?