Git 101 for Think Global Hack Local

Jason Tseng and Matt Gingerich, 2013

What is Git?

Simple definition - Git is a Distributed Revision Control System

Slightly detailed definition - Git is a platform that allow collaborators to track project change histories and share/sync code as each contributor works separately

Basic Terminology

- Repository (repo) - Representation of your project - all enclosing files, delta, change histories etc.
- Branch - a virtual copy of your project, allowing you to make changes to this copy and not introduce impact on the main copy of the project. Once satisfied with their changes, user can merge the changes back to the main copy and resolve any change conflicts.
- Commit - creating a snapshot of changes to the files in the branch that the user is currently using
- Remote Branch - link/path to the remote repo-branch where user can sync with others
- Staging - readying files that user wants to be recorded in the next snapshot (commit). Users must place files that contain changes into “staging” mode before committing
- Checkout - Retrieval of the branch that the user wants to work on.

Useful Commands

- Create Repo: git init
- Add to Stage: git add [<path> | .]
- Commit: git commit [-m <msg>]
- Push to remote: git push <remoteName> <remoteBranch> (eg: git push origin master)
- Check revision log: git log
- Check differences: git diff <commit1_hash> <commit2_hash> [<specific_file_path>]
- Check differences between staged and HEAD: git diff --cached
- Check differences between staged and non-staged modifications: git diff
- Check local branches: git branch
- Check remote branches: git branch -r
- Add remote repo: git remote add <remoteName> <remotePath> (eg: git remote add origin git://xyz.com/myrepo.git)
- Switch to another local branch: git checkout <branch_name>
- Make a copy of the local branch to a new branch and checkout the new branch: git checkout -b <new_branch_name> <existing_branch_name> (eg: git checkout -b dev master)
- Delete a local branch: git branch -D <branch_name>
- Set Conflict display style to include ancestor branch code: git config merge.conflict style diff3

Overall Strategy for hackathon

1. Create a repository on github, share the repo to all members on your team with commit access
2. Download git to your local machine
3. On each of your local machine, use "git clone <path>" command to clone the repo to your machine. This will auto set up the remote branch path for you so you don't have to worry about it later
4. CD (change directory) to the folder git created from cloning (your project folder), use the following command to set conflict display style “git config merge.conflict style diff3”
5. Assuming you start on the local master branch, use the following command to create a dev branch on your machine. "git checkout -b dev master"
6. Start crunching your code on the branch created after the clone command
7. At each stage which you are satisfied with your code, stage the code and commit to your local dev branch. "staging -> git add <path | . >" "committing -> git commit [-m "/msg/"]"
8. Prepare to merge the code with the main branch first by checking out the master branch, pulling latest commit from the github, then merge the changes from your dev, resolve conflicts, and finally push the latest code back to github.
    a. "git checkout master"
    b. "git pull" (this will perform git fetch and git update together, which will make your master branch up to date with the latest code from github)
    c. "git merge dev" - merges dev branch back to master branch, git will try its best to resolve conflicts, and will notify you to perform manual conflict resolve if it failed doing so. *See conflict resolution*
    d. Test your code, make sure it works as expected. Even after conflicts are resolved, there may be bugs introduced by merging the codes.
    e. "git push origin master" - pushes the latest code to github
9. Go back to your dev branch and update it with the latest code. "git checkout dev" "git merge master"
10. back to work (Step 5)

Merging / Conflict Resolution

If a conflict arose during merging, you will need to manually check the conflict files to resolve the conflicting lines, and commit the fixed files.

1. With the previous config set for conflicts, you should be able to find the following in the conflict files

   <<<<<<< HEAD

   code from master branch

   | | | | | | | |

   code from the closest common ancestor

   =========

   code from your dev branch

   >>>>>>>> dev
2. What you need to do here is simply decide which code snippet you want to keep, remove all unwanted snippets, and save the file
3. Test your program/code to make sure it behaves as you expected
4. git add <fixed_file_path>
5. git commit [-m "msg"]

Branch Recommendation

Have a master branch on github as the central repo for the latest working code. Each team member have a master and dev branch on their own machine. Local master branch is used for syncing with latest code from github, and merging with local changes. Local dev branch will be where individuals perform actual development.

If some initial framework is being used, (eg. setting up a project for objC app), have one of the member set up the framework, commit to the github, then everyone else clone from it and start from there.

Ignore Files

If your project/program creates cached files, or if you have config files that are unique between members, you can prevent those files from being committed by creating a ".gitignore" file in the root folder, and list out a path per line for folders/files you want to ignore from staging/commits. *Note: if you have already committed a file you want to ignore, check here*

Additional Resources

Good resource for git commands:

https://www.kernel.org/pub/software/scm/git/docs/git.html

Git Branching (Basics):

http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging

Git Branching (Advanced):

http://nvie.com/posts/a-successful-git-branching-model/

Warnings and Tips

A hackathon is an unusual coding environment in several ways. You will likely be making many commits to a handful of files very rapidly over the weekend, so merge conflicts may be more frequent and more troublesome than you are accustomed to if you have used Git in the context of a more standard software development cycle. If you're new to Git, this can be a really rough introduction to the system, so here's a few tips to ease that pain a little:

1. Use a good Git client such as SourceTree (Mac, Windows), TortoiseGit (Windows), or some combination of these tools (Linux). It might not be as hardcore as using a command line, but being able to visually track your changes could save a lot of mayhem.
2. Stage your commits carefully. Don't commit files that aren't ready to be pushed, aren't necessary for your patch, or that you haven't made meaningful changes to. Use the "Ignore Files" feature described above to avoid staging files that are automatically generated or specific to you.
3. Try to avoid working on the same file as someone else at the same time. You're

surrounded by your collaborators, so talk to them to figure out who's working on what. Try not to start fixing typos or weird indentation styles in a file that one of your teammates has been working on (but do make a note of the issues and resolve them once the file is free, or let the person who's working on the file handle the changes). Merging is easiest when you don't create conflicts in the first place.

4. If you find that your team is routinely generating conflicts despite the previous tips, you might have too many people working independently on the same problems. Consider using a pair-programming approach; a second set of eyes can be really helpful during marathon coding sessions.
    a. To get the most out of pair programming, you need to do a bit more than just sit two programmers in front of the same computer. The Wikipedia page on the topic is a good place to get a brief overview of the practice, but for a more in-depth guide, I'm partial to the article "All I Really Need to Know about Pair Programming I Learned in Kindergarten".
5. Be *very careful* about overwriting versioning history or rejecting someone else's changes. If someone has committed code that breaks the build for everyone, communicate with your group to decide how to fix the problem and who will do it. If everyone independently tries to fix their local repositories, you're in for major merging headaches.