Github Recap, Intermediate Github

Let's talk about version control! As soon as you start at your company working as a developer, you will be expected to understand version control. So you might as well be good at it.

We've established that Git is a version control system, similar but better than the many alternatives available. So, what makes GitHub so special? Git is a command-line tool, but the center around which all things involving Git revolve is the hub—GitHub.com—where developers store their projects and network with like minded people.

Changelogs

When multiple people collaborate on a project, it's hard to keep track revisions—who changed what, when, and where those files are stored. GitHub takes care of this problem by keeping track of all the changes that have been pushed to the repository.

GitHub Isn't Just for Developers

All this talk about how GitHub is ideal for programmers may have you believing that they are the only ones who will find it useful. Although it's a lot less common, you can actually use GitHub for any types of files. If you have a team that is constantly making changes to a word document, for example, you could use GitHub as your version control system. This practice isn't common, since there are better alternatives in most cases, but it's something to keep in mind.

Naming

While naming a repository try following the following conventions:

- Use lower case
- Use dashes i.e my-first-repo looks way better that myfirstrepo or my_first_repo.

Issues

This section is one of the most popular bug tracker in the world. It provides the owners of a repository the ability to organize, tag and assign to milestones issues. If you open an issue on a project managed by someone else, it will stay open until either you close it (for example if you figure out the problem you had) or if the repo owner closes it.

Pull Requests

Pull requests let you tell others about changes you've pushed to a GitHub repository.

Branches

By now you understand that git saves each version of your project as a snapshot of the code exactly as it was at the moment you committed it. Essentially creating a timeline of versions of a project as it progresses, so that you can roll back to an earlier version in the event disaster strikes.

The way git, and GitHub, manage this timeline — especially when more than one person is working in the project and making changes — is by using branches. A **branch** is essentially is a unique set of code changes with a unique name. Each repository can have one or more branches. The main branch — the one where all changes eventually get merged back into, and is called master. This is the official working version of your project, and the one you see when you visit the project repository at github.com/yourname/projectname.

Do not mess with the master. If you make changes to the master branch of a group project while other people are also working on it, your on-the-fly changes will ripple out to affect everyone else and very quickly there will be merge conflicts, weeping, rending of garments, and plagues of locusts. It's that serious.

Why is the master so important to not mess with? One word: the master branch is deployable. It is your production code, ready to roll out into the world. The master branch is meant to be stable, and it is the social contract of open source software to never, ever push anything to master that is not tested, or that breaks the build. The entire reason GitHub works is that it is always safe to work from the master.

Branching Out

Instead, everyone uses branches created from master to experiment, make edits and additions and changes, before eventually rolling that branch back into the master once they have been approved and are known to work. Master then is updated to contain all the new stuff.

Github Cheat Sheet

Git users can broadly be grouped into categories

Individual Developer (Standalone) commands are essential for anybody who makes a commit, even for somebody who works alone.

If you work with other people, you will need commands listed in the **Individual Developer (Participant)** section as well.

People who play the **Integrator role** need to learn some more commands in addition to the above.

Individual Developer (Standalone)

A standalone individual developer does not exchange patches with other people, and works alone in a single repository, using the following commands.

- git-init(1) to create a new repository.
- git-log(1) to see what happened.

- git-switch(1) and git-branch(1) to switch branches.
- git-add(1) to manage the index file.
- git-diff(1) and git-status(1) to see what you are in the middle of doing.
- git-commit(1) to advance the current branch.
- git-restore(1) to undo changes.
- git-merge(1) to merge between local branches.
- git-rebase(1) to maintain topic branches.
- git-tag(1) to mark a known point.

Examples

Use a tarball as a starting point for a new repository.

1. $ tar zxf frotz.tar.gz
2. $ cd frotz
3. $ git init
4. $ git add . <1>
5. $ git commit -m "import of frotz source tree."
6. $ git tag v2.43 <2>

English:

1. add everything under the current directory.
2. make a lightweight, unannotated tag.

Create a topic branch and develop.

1. $ git switch -c alsa-audio <1>
2. $ edit/compile/test
3. $ git restore curses/ux_audio_oss.c <2>
4. $ git add curses/ux_audio_alsa.c <3>
5. $ edit/compile/test
6. $ git diff HEAD <4>
7. $ git commit -a -s <5>
8. $ edit/compile/test
9. $ git diff HEAD^ <6>
10. $ git commit -a --amend <7>
11. $ git switch master <8>
12. $ git merge alsa-audio <9>
13. $ git log --since='3 days ago' <10>
14. $ git log v2.43.. curses/ <11>

English:

1. create a new topic branch.
2. revert your botched changes in curses/ux_audio_oss.c.
3. you need to tell Git if you added a new file; removal and modification will be caught if you do git

commit -a later.
4. to see what changes you are committing.
5. commit everything, as you have tested, with your sign-off.
6. look at all your changes including the previous commit.
7. amend the previous commit, adding all your new changes, using your original message.
8. switch to the master branch.
9. merge a topic branch into your master branch.
10. review commit logs; other forms to limit output can be combined and include -10 (to show up to 10 commits), --until=2005-12-10, etc.
11. view only the changes that touch what's in curses/ directory, since v2.43 tag.

Individual Developer (Participant)

A developer working as a participant in a group project needs to learn how to communicate with others, and uses these commands in addition to the ones needed by a standalone developer.

- git-clone(1) from the upstream to prime your local repository.
- git-pull(1) and git-fetch(1) from "origin" to keep up-to-date with the upstream.
- git-push(1) to shared repository, if you adopt CVS style shared repository workflow.
- git-format-patch(1) to prepare e-mail submission, if you adopt Linux kernel-style public forum workflow.
- git-send-email(1) to send your e-mail submission without corruption by your MUA.
- git-request-pull(1) to create a summary of changes for your upstream to pull.

Examples

Clone the upstream and work on it. Feed changes to upstream.

1. $ git clone git://git.kernel.org/pub/scm/.../torvalds/linux-2.6 my2.6
2. $ cd my2.6
3. $ git switch -c mine master <1>
4. $ edit/compile/test; git commit -a -s <2>
5. $ git format-patch master <3>
6. $ git send-email --to="person email@example.com" 00*.patch <4>
7. $ git switch master <5>
8. $ git pull <6>
9. $ git log -p ORIG_HEAD.. arch/i386 include/asm-i386 <7>
10. $ git ls-remote --heads http://git.kernel.org/.../jgarzik/libata-dev.git <8>
11. $ git pull git://git.kernel.org/pub/.../jgarzik/libata-dev.git ALL <9>
12. $ git reset --hard ORIG_HEAD <10>
13. $ git gc <11>

English:

1. checkout a new branch mine from master.
2. repeat as needed.

3. extract patches from your branch, relative to master,
4. and email them.
5. return to master, ready to see what's new
6. git pull fetches from origin by default and merges into the current branch.
7. immediately after pulling, look at the changes done upstream since last time we checked, only in the area we are interested in.
8. check the branch names in an external repository (if not known).
9. fetch from a specific branch ALL from a specific repository and merge it.
10. revert the pull.
11. garbage collect leftover objects from reverted pull.

Push into another repository.

1. satellite$ git clone mothership:frotz frotz <1>
2. satellite$ cd frotz
3. satellite$ git config --get-regexp '^(remote|branch).' <2>
4. remote.origin.url mothership:frotz
5. remote.origin.fetch refs/heads/*:refs/remotes/origin/*
6. branch.master.remote origin
7. branch.master.merge refs/heads/master
8. satellite$ git config remote.origin.push \ +refs/heads/*:refs/remotes/satellite/* <3>
9. satellite$edit/compile/test/commitsatellite$ git push origin <4>
10. mothership$ cd frotz
11. mothership$ git switch master
12. mothership$ git merge satellite/master <5>

English:

1. mothership machine has a frotz repository under your home directory; clone from it to start a repository on the satellite machine.
2. clone sets these configuration variables by default. It arranges git pull to fetch and store the branches of mothership machine to local remotes/origin/* remote-tracking branches.
3. arrange git push to push all local branches to their corresponding branch of the mothership machine.
4. push will stash all our work away on remotes/satellite/* remote-tracking branches on the mothership machine. You could use this as a back-up method. Likewise, you can pretend that mothership "fetched" from you (useful when access is one sided).
5. on mothership machine, merge the work done on the satellite machine into the master branch.

Branch off of a specific tag.

1. $ git switch -c private2.6.14 v2.6.14 <1>
2. $ edit/compile/test; git commit -a
3. $ git checkout master
4. $ git cherry-pick v2.6.14..private2.6.14 <2>

English:

1. create a private branch based on a well known (but somewhat behind) tag.
2. forward port all changes in private2.6.14 branch to master branch without a formal "merging". Or longhand below:

```
git format-patch -k -m --stdout v2.6.14..private2.6.14 | git am -3 -k
```

An alternate participant submission mechanism is using the git request-pull or pull-request mechanisms (e.g as used on GitHub (www.github.com) to notify your upstream of your contribution.

Integrator

A fairly central person acting as the integrator in a group project receives changes made by others, reviews and integrates them and publishes the result for others to use, using these commands in addition to the ones needed by participants.

This section can also be used by those who respond to git request-pull or pull-request on GitHub (www.github.com) to integrate the work of others into their history. A sub-area lieutenant for a repository will act both as a participant and as an integrator.

- git-am(1) to apply patches e-mailed in from your contributors.
- git-pull(1) to merge from your trusted lieutenants.
- git-format-patch(1) to prepare and send suggested alternative to contributors.
- git-revert(1) to undo botched commits.
- git-push(1) to publish the bleeding edge.

Examples

A typical integrator's Git day.

1. $ git status <1>
2. $ git branch --no-merged master <2>
3. $ mailx <3> & s 2 3 4 5 ./+to-apply & s 7 8 ./+hold-linus & q
4. $ git switch -c topic/one master
5. $ git am -3 -i -s ./+to-apply <4>
6. $ compile/test
7. $ git switch -c hold/linus && git am -3 -i -s ./+hold-linus <5>
8. $ git switch topic/one && git rebase master <6>
9. $ git switch -C pu next <7>
10. $ git merge topic/one topic/two && git merge hold/linus <8>
11. $ git switch maint
12. $ git cherry-pick master~4 <9>
13. $ compile/test
14. $ git tag -s -m "GIT 0.99.9x" v0.99.9x <10>
15. Misplaced & branch $branch < 12 > done$ git push --follow-tags ko <13>

English:

1. see what you were in the middle of doing, if anything.
2. see which branches haven't been merged into master yet. Likewise for any other integration branches e.g. maint, next and pu (potential updates).
3. read mails, save ones that are applicable, and save others that are not quite ready (other mail readers are available).
4. apply them, interactively, with your sign-offs.
5. create topic branch as needed and apply, again with sign-offs.
6. rebase internal topic branch that has not been merged to the master or exposed as a part of a stable branch.
7. restart pu every time from the next.
8. and bundle topic branches still cooking.
9. backport a critical fix.
10. create a signed tag.
11. make sure master was not accidentally rewound beyond that already pushed out.
12. In the output from git show-branch, master should have everything ko/master has, and next should have everything ko/next has, etc.
13. push out the bleeding edge, together with new tags that point into the pushed history.

In this example, the `ko` shorthand points at the Git maintainer's repository at kernel.org, and looks like this:

(in .git/config) [remote "ko"] url = kernel.org:/pub/scm/git/git.git fetch = refs/heads/*:refs/remotes/ko/* push = refs/heads/master push = refs/heads/next push = +refs/heads/pu push = refs/heads/maint

Lab Activity

1. As a group, decide on a website topic.

Examples

(Create a website for a coffee shop) (Create a website for a CBD shop) (Create a website for a Chinese restaurant)

2. Decide on each person role for the grouped
3. As a group, create this website and push to github
4. Using the guidelines in today's lesson, make sure to contribute to each other's code and make a working site.
5. Good luck!

References:

1. https://mirrors.edge.kernel.org/pub/software/scm/git/docs/giteveryday.html
2. https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/