Using Git with friends

(or coworkers, if you don't have friends)

Distributed vs centralized version control

Git is a *distributed* version control system (DVCS):

- History is stored locally with every copy of a repository
- New commits and objects become part of the local history first
- Sharing commits and objects is an explicit operation
- No authoritative copy of a repo (except socially)

Compare to *centralized* VCSes like Subversion, CVS, and Perforce:

- A repository's state and history are stored on a single server
- Clients talk to that server to fetch files and make changes
- New commits are automatically visible to everyone

Git sharing subcommands

`git remote`: Tell Git what other copies of the repository you care about.
Set up or view *remotes*, aliases for repositories stored elsewhere that you can exchange objects with. Git can access a remote stored on the local filesystem, on an HTTP server, or on any system accessible via SSH.

`git fetch`: Copy branches from a remote to your local repository.
Branches from a remote are prefixed with the remote's name and a slash. For example, `origin/main`.

`git push`: Copy a branch from your local repository to a remote.
Modifies branches on the remote, so requires write access.

What questions do you have?

git remote

Add, remove, modify, or query remotes.

With no arguments, lists remotes. `-v` shows URLs.

`add`: Adds a remote with the given name and URL.

`remove`: Removes a remote.

`rename`: Changes a remote's name.

`set-url`: Changes a remote's url.

```
$ git remote add origin /comp/50ISDT/examples/git-zoo/
$ git remote
origin
$ git remote add tom /h/thebb01/git-zoo/
$ git remote -v
origin    /comp/50ISDT/examples/git-zoo/ (fetch)
origin    /comp/50ISDT/examples/git-zoo/ (push)
tom    /h/thebb01/git-zoo/ (fetch)
tom    /h/thebb01/git-zoo/ (push)
$ git remote rename tom max
$ git remote set-url max /h/mberns01/repos/zoo/
$ git remote -v
max    /h/mberns01/repos/zoo/ (fetch)
max    /h/mberns01/repos/zoo/ (push)
origin    /comp/50ISDT/examples/git-zoo/ (fetch)
origin    /comp/50ISDT/examples/git-zoo/ (push)
$ git remote remove max
$
```

`git fetch`

Copy objects and branches from a remote to local remote-tracking branches.

Updates *remote-tracking branches* for the given remote. Doesn't affect your local branches, index, or working tree.

Can take a list of one or more branch names to fetch; if none are given, fetches all branches.

`--all`: Fetches from all remotes at once.

```
$ git init  # New, empty repository
$ git remote add origin /comp/50ISDT/examples/git-zoo/
$ git fetch origin
remote: Enumerating objects: 36, done.
<...>
From /comp/50ISDT/examples/git-zoo
 * [new branch]       add-file3   -> origin/add-file3
 * [new branch]       add-symlink -> origin/add-symlink
 * [new branch]       main        -> origin/main
$ git branch main origin/main
Branch 'main' set up to track remote branch 'main' from
'origin'.
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
$ ls
directory1  directory2  file1  file2  file3  missing-link
$
```

git push

Copy one or more branches, and all their objects, to a remote.

Takes a remote name followed by one or more branch names. Creates or updates the named branches on the remote from matching local branches.

If local and remote branches have different names, specify them both separated by a colon.

Specifying a remote branch preceded by a colon deletes that branch on the remote.

```
$ mkdir ../tmp/ && cd ../tmp/ && git init && cd -
Initialized empty Git repository in /h/thebb01/example/tmp/.git/

$ git remote add example ../tmp/
$ git push example main
Enumerating objects: 33, done.
Counting objects: 100% (33/33), done.
Delta compression using up to 6 threads
Compressing objects: 100% (21/21), done.
Writing objects: 100% (33/33), 3.48 KiB | 209.00 KiB/s, done.
Total 33 (delta 9), reused 0 (delta 0)
To ../tmp
 * [new branch]       main -> main

$ cd ../tmp/
$ git branch -v
  main 513f37f Add a longer file in directory2
$
```

What questions do you have?

SSH and HTTP(S) remotes

```
laptop$ mkdir zoo && cd zoo && git init
Initialized empty Git repository in /home/thebb/zoo/.git/

laptop$ git remote add origin thebb01@homework.cs.tufts.edu:/comp/50ISDT/examples/git-zoo

laptop$ git fetch origin
remote: Enumerating objects: 36, done.
remote: Counting objects: 100% (36/36), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 36 (delta 10), reused 32 (delta 8)
Unpacking objects: 100% (36/36), 3.66 KiB | 85.00 KiB/s, done.
From homework.cs.tufts.edu:/comp/50ISDT/examples/git-zoo
 * [new branch]      add-file3   -> origin/add-file3
 * [new branch]      add-symlink -> origin/add-symlink
 * [new branch]      main        -> origin/main

laptop$
```

`git clone`

Shorthand for `git init` + `git remote add` + `git fetch` + `git checkout -b`

Takes a URL, which will be set as the `origin` remote, and an optional directory name to clone to.

```
$ git clone git@github.com:tekknolagi/isdt.git
Cloning into 'isdt'...
remote: Enumerating objects: 464, done.
remote: Counting objects: 100% (464/464), done.
remote: Compressing objects: 100% (284/284), done.
remote: Total 464 (delta 257), reused 334 (delta 141), pack-reused 0
Receiving objects: 100% (464/464), 271.34 KiB | 3.01 MiB/s, done.
Resolving deltas: 100% (257/257), done.
$
```

What questions do you have?

Integrating changes from remotes

Just like integrating changes from a local branch!

Three main strategies:

- **Fast-forward**: If no new local commits and no history-rewriting remote commits, add all the new remote commits to the local branch.
- **Merge**: If new local commits and new remote commits, use `git merge` to create a merge commit derived from both sets of changes.
- **Rebase**: If local commits have not been shared anywhere, can alternatively rebase them on top of new remote commits.

git pull

Fetches changes from the remote branch that your checked-out branch tracks (see git branch -vv), then merges or rebases those changes into the local branch.

Warning: `git pull` relies on a lot of implicit configuration (like remote branch mappings) and can have different results (fast-forward vs merge commit) depending on the state of the remote. It's often better to explicitly run `git fetch` followed by your desired integration command.

`--ff-only`: Forces a fast-forward merge. Fails if not possible.

`--rebase (-r)`: Rebases local changes on top of new remote state instead of merging.

Conflicts

What happens if a merge, rebase, or cherry-pick involves two commits that changed a single base version of a file in two different ways?

- All these operations call `git merge-file`
- Does its best to reconcile the changes on its own by splitting into hunks
- If two hunks conflict, enters *conflict resolution*:
  - Conflicting hunks are marked in the working tree
  - Special `<<<<<<<` lines indicate conflicts
  - Up to you to resolve manually
  - Use `--abort` if there are too many conflicts
- GUI tools for merging files: WinMerge, Meld

What questions do you have?

Common open-source development workflow

- One repository copy designated as source of truth
  - e.g. torvalds/linux
- *Maintainers* are allowed to push to this copy
- Anyone else wanting to contribute communicates their changes to maintainers out-of-band
  - For Linux, patches on a mailing list
  - For many other project, GitHub *pull requests*

What questions do you have?

Maintaining a codebase

Limitations of collaboration via `git push`

- Requires you to fully trust every contributor.
- Easy for bad code to be committed.
- No built-in way to comment on a commit.
- No way to enforce code review policies.

Solution: merge/pull requests and patches

Out-of-band way for a contributor to propose a change to a repository.

- **Merge request** (a.k.a. pull request): Instead of a contributor pushing their feature branch upstream, they push it to their own *fork* and ask the upstream maintainer to pull it from there.
- **Patch**: No pushes or pulls happen at all. The contributor formats each commit they've made as a textual *patch* using `git show` or `git format-patch`, sends it to the maintainer (e.g. in an email), and the maintainer *applies* that patch using `git am` or `git apply`.

Either way, maintainers can comment on the change via the same communication channel it was sent over and only accept it if they approve of it.

What questions do you have?

Revising proposed changes

Two schools of thought:

1. Add new commits containing fixes on top of the original ones
    - Pros: easy to see what changed in each revision, commit message can explain what got fixed
    - Cons: no way to "fix" a bad commit message, makes `git blame` harder
2. Rewrite history to produce new, fixed versions of each commit
    - Pros: results in clean, readable commits at the end of the day
    - Cons: requires maintainers to deal with rewritten history prior to acceptance

(See https://github.com/tekknolagi/isdt/pull/15#issuecomment-920592835.)

What questions do you have?

Bare vs full repositories