1 What is a Commit?

1.1 Lecture

Git Repositories have only a few kinds of fundamental objects that can be manipulated in only a few ways.

Since this lecture is about those things, it is very important: Everything else in this course will build on what we learn today.

1.1.1 Git Objects

- Only three kinds of objects in Git: commits, trees, and blobs
- Every object is addressed (and uniquely identified) by a hash
  - The hash depends on the entire contents (and metadata) of the object, making them all immutable.
  - Each is stored in `.git/objects/<hash[0:2]>/<hash[2:39]>`
  - Hashes are all 40 characters
    - But, "As a convenience, Git requires only as many digits of the hash id as are necessary to uniquely identify it within the repository" (Git from the Bottom Up, Introducing the Blob).
- Immutability means that:
  - Git always knows when the Repository has changed because changes to objects will always result in new hashes for the modified versions and thus new objects for them.
  - Deduplication is easy: two objects with the same contents+metadata are always the same object in storage.
    - Kinda like if `a == b` implied `a is b` in Python

Blobs Blobs are content * Each represents a particular version of a file * Like files in a filesystem * Metadata: size (derivable from content)

Trees Trees are trees of blobs * Each represents a directory in the working directory * Like directories in a filesystem * Metadata: name and permissions of each child

Commits Commits each point to a single tree representing the entire working directory * Each represents a point in the working directory's history * Or, a node in the graph that is the Repository * Metadata: author, committer, message, references to parent commit(s), time of authoring, time of committing * Take a snapshot of the working directory (tree) and incorporate it into its history * Answers some basic history questions about the snapshot: * Who made it (committer, author) * When it was made (time of authoring) * Why it was made (message) * The context in which it was made (references to parent commit(s))

Teaser for next time

- Branches are references to commits
- Branches move to include new commits when they're checked out
- `HEAD` points to the currently checked-out branch

1.2 Lab/Homework

Make a repository with a few commits using only low-level (plumbing) Git commands. Adapted from the Git Book, section 10.2.

1.2.1 Viewing tools

The following are suggestions for viewing the Repository's state throughout the procedure below. * **List all objects in the repository:** `find .git/objects -type f` * Objects are listed as files with names corresponding to their hashes * **View the contents of an object:** `git cat-file -p <object-hash>` * For **blobs**, prints its contents * For **trees**, lists its child objects with name, type, and hash * For **commits**, lists its tree, message, author, and committer * **See a diagram of all objects:** `python /path/to/git-graph.py /path/to/lab_1_2_repo` * **View the type of an object:** `git cat-file -t <object-hash>` * **View the contents of the Staging Area:** `git status`, or `gitui` * **View the history of a commit:** `git log <commit-hash>` * **NB:** `git log` without arguments will always be empty in this lab because it shows the history of `HEAD`, which is never set.

1.2.2 Procedure

1. Initialize repo
   1. `git init lab_1_2_repo`
   2. `cd lab_1_2_repo/`
   3. Use `find .git/objects` to see that it only has a couple of empty directories

Make Blobs

1. Create a couple of blobs from different versions of a file
   1. `echo 'version 1' > test.txt`
   2. `git hash-object -w test.txt`
   3. `echo 'version 2' > test.txt`
   4. `git hash-object -w test.txt`
2. Try recovering both versions of the file `test.txt` from Git using `git cat-file`:
   1. `rm test.txt`
   2. `git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt` should give you "version 1" in the file
   3. `git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt` should give you "version 2" in the file

      1. **NB:** I know the hashes of these blobs because *they depend only on the blob's contents.*
3. Try making a duplicate blob
   1. `echo 'version 1' | git hash-object -w --std-in`
   2. You should see that this blob's hash is the same as the first version of `test.txt` you made above.
      1. Remember that a blob's hash depends only on its contents, so it doesn't matter what file they're in, or even if they're in a file to begin with.
   3. You should also see that no new objects have been made with this operation. Since all objects are identified by the hash of their contents+metadata, this means that no existing object's contents or metadata have changed, either.
   4. Since the Repository is fundamentally just the set of all Git objects in a given `.git` directory, the Repository as a whole hasn't changed.
   5. It's safe to conclude that *making duplicate objects in Git is impossible.*

Make Trees from Blobs

1. Git writes trees from the Staging Area (Index), so you have to stage your blobs to make trees out of them.
   1. `git update-index --add --cacheinfo 100644 83baae61804e65cc73a7201a7252750c76066a30 test.txt`
      1. **NB:** You needed to name the blob (and add permissions for it) when adding it to the Staging Area because Git *doesn't store file metadata in blobs, just contents.*
2. Write the contents of the Staging Area to a tree with
   1. `git write-tree`
3. Now, do `the` same `thing`, `but` with the second version of `test.txt`, and a new file that will be added from the Working Tree rather than the Repository.
   1. `git update-index --cacheinfo 100644 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt`
   2. `echo 'new file' > new.txt`
   3. `git update-index --add new.txt`
   4. `git write-tree`
4. Lastly, make a nested tree by adding the first tree to the Staging Area as a subdirectory.
   1. `git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579`
      1. The `--prefix` option gives Git a name for the subdirectory represented by the given tree.
   2. `git write-tree`

Make Commits from Trees

1. Make a commit for each tree you made in the last section, in order

1. `git commit-tree -m 'First commit' d8329f`
2. `git commit-tree -m 'Second commit' 0155eb -p <first-commit-hash>`
3. `git commit-tree -m 'Third commit' 3c4e9c -p <second-commit-hash>`
4. **NB:** I *don't* know the hashes of these commits because they depend on the author, committer, and the time the commit was made.

Now you have a commit history, viewable with `git log <third-commit-hash>`, made only from plumbing commands!