COMP(2041|9044) 25T1 — GIT

https://www.cse.unsw.edu.au/~cs2041/25T1/

Things developers want

- Tell me what changes were recently made to this file?
- Tell me who added this line of code? When? Why?
- Take all files back to the way they were 2 weeks ago
- 2 coders have been working independently on the system - combine their work safely
- Develop new system features in parallel but still incorporate bug fixes being made to the main release
- Allow me to propose this bug fix and get comments from other developers
- Record that these code changes fix this bug report.

Git

- `Git` is a Version Control System (VCS)
  - Track changes to a file or set of files over time
    so that you can recall specific versions later
- `Git` is open source under the GPLv2 licence
  - Git git repo
  - Created for and still used by Linus Torvalds for the linux kernel

Other VCS

- SCCS
- RCS
- CVS
- Subversion
- Mercurial
- Fossil
- etc.

VCS terminology

- Repository (repo)
- Branches
  - Default Branch (master/main/trunk)
- Tags
- Commits
- Index
  - Staging
- Working Directory

Repository

Many VCSs use the notion of a *repository*

- store all versions of all objects (files) managed by VCS
- may be single file, directory tree, database,...
- possibly accessed by filesystem, http, ssh or custom protocol
- possibly structured as a collection of *projects*

Git Repository

Git uses the sub-directory `.git` to store the repository.

Inside `.git` there are (among other things):

- **Objects**
  - **Blobs** are file contents
    - no file names, permissions, links, etc.
  - **Trees** are directory listings
    - model the file system
    - this is where: file names, permissions, links, etc. live
    - trees can also point to other trees to store subdirectories
  - **Commits** are snapshots
    - represents the state of the working directory at a particular time
    - has a list of parent commits
    - stores meta info: author, committer, message, etc.
    - points to a tree that represents the file structure at the time of the commit
- **Refs** are pointers
  - **Branches**
    - branches provide dynamic pointers to the commits we care about
    - contain hex strings referencing the Object ID of a commit
  - **Tags**
    - tags provide static pointers to historic commits
    - contain hex strings referencing the Object ID of a commit

Inside a Git Repository

A new git repository is created with `git init` will have the following structure:

```
$ tree .git
.git/
├── config
├── HEAD
├── objects
└── refs
    ├── heads
    └── tags
```

Some files are not shown as they are not relevant for us.

- `branches/` is a deprecated implementation of `heads/`
- `description` is only used by the gitweb program
- `hooks/` is used for git hooks (very useful, but not relevant for us)
- `info/` is used for git logs and metadata (not relevant for us)

Inside a Git Repository

Once we have added some files and made some commits the structure may look like this:

```
$ tree .git
.git/
├── config
├── HEAD
├── objects
│   ├── 63
│   │   ├── 438577f200a1323959c79c6bcbebd98b52f95c
│   │   ├── 8bd101ad9ff2a7f224fa89062a693d2afa4964
│   │   └── <more objects>
│   ├── 8d
│   │   └── <more objects>
│   ├── c2
│   │   └── <more objects>
│   ├── ff
│   │   └── <more objects>
│   └── <more objects>
└── refs
    ├── heads
    │   ├── master
    │   ├── develop
    │   ├── feature
    │   │   ├── feature1
    │   │   ├── feature2
    │   │   └── <more branches>
    │   └── <more branches>
    ├── remotes
    │   └── origin
    │       ├── HEAD
    │       └── master
    └── tags
        ├── v1.0
        ├── v1.1
        ├── v1.2
        ├── v2.0
        ├── v2.1
        └── v3.0
```

Inside a Git Repository

HEAD is a special file that points to the current ref

- This is usually a branch
- But it can also be a tag or a specific commit

`refs/heads/` contains all the branches `refs/tags/` contains all the tags `refs/remotes/` contains all the remote branches

- refs are simply pointers to commits

Inside a Git Repository

`objects/` contains all the objects

each object is a 20 byte SHA1 hash of the object contents
stored as a 40 character hex string.

the first two characters of the hash are used as a directory name
with the remaining 38 characters as the file name.

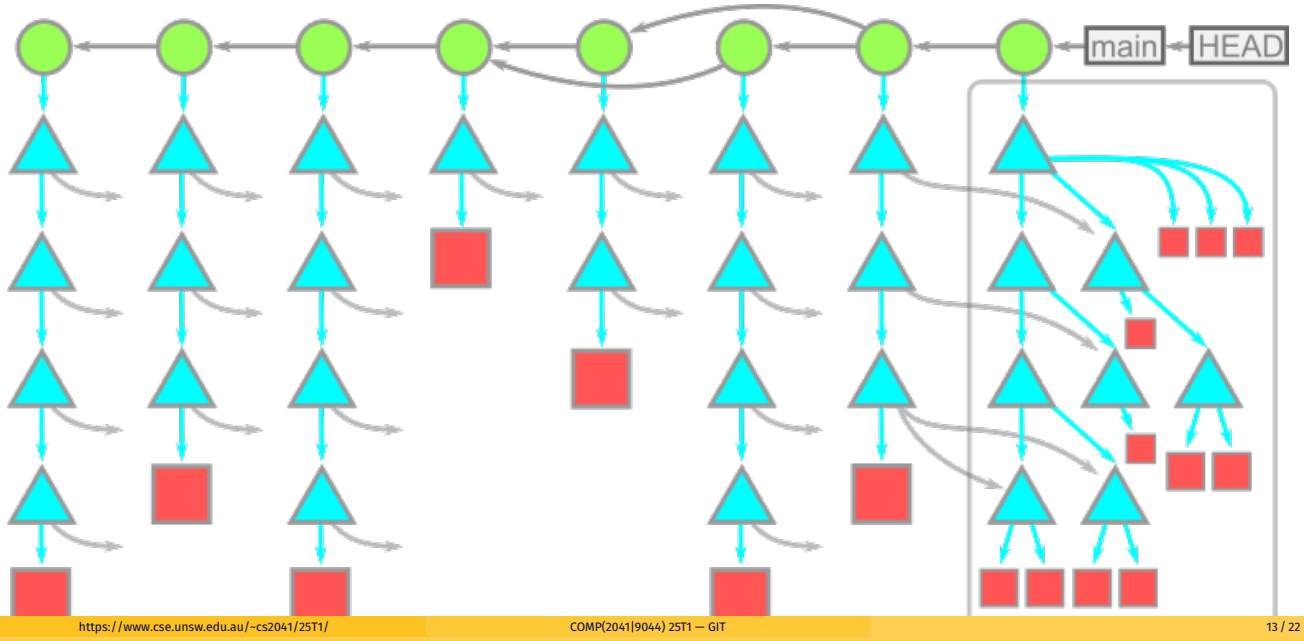
- objects are stored compressed, so can't be read directly

Inside a Git Repository

- `git ls-files -s`
  - lists all objects in the index
- `git cat-file -t <object>`
  - prints the type of the object
- `git cat-file -p <object>`
  - prints the contents of the object
- `git cat-file --batch-check --batch-all-objects`
  - list all objects, their type and size
- `git rev-list --objects --all`
  - list all objects and their name (if they have one)

Git Repository Overview

Boxes are blobs, Triangles are trees, Circles are commits



Git Repo Hosting

Some of the best known Git repo hosting services

- GitHub
- GitLab
  - UNSW CSE GitLab
- BitBucket
- SourceForge
- etc.

Why Git?

- distributed VCS - multiple repositories, no oracle
- every user has their own repository
- created by Linus Torvalds for Linux kernel
- external revisions imported as new branches
- flexible handling of branching
- various auto-merging algorithms
- not better than competitors but better supported/more widely used (e.g. github/gitlab/bitbucket)
- at first stick with a small subset of commands
- substantial (exponential) time investment to learn to use Git's full power

git commands

The **80/20 rule**:
80% of the time you run the same 20% of the available commands.

The **BIG** 7:

- `git init [<name>]` or `git clone <URI>`
- `git status`
- `git add <file>...`
- `git commit [-m "<message>"]`
- `git pull`
- `git push`

The others:

- `git branch <branch>`
- `git checkout <branch>`
- `git fetch`
- `git log`
- `git stash`
- `git cherry-pick`
- `git bisect`

git init

git-init - Create an empty Git repository

How every repository starts.

```
git init [options]       # turn the current directory into a git repo
git init [options] <dir> # create a new directory `dir` that is a git repo
```

Has some very rarely used options:

- `--bare` repo without a working directory, can't commit to the repo.
- `--template` files to copy into `.git` upon creation.
- `--separate-git-dir` create a working directory for a repo located elsewhere
- `--shared` share the repo amongst several users

Reads some very rarely used environment variables:

- `$GIT_DIR` if set use `$GIT_DIR` not `.git` as the name of the base of the repository
- `$GIT_OBJECT_DIRECTORY` store object files here instead of `$GIT_DIR/objects`

99% of the time you will use `git init` without options.

git clone

git-clone - Clone a repository into a new directory

How repositories are shared.

```
git clone [options] <repoURL>       # clone the git repo from `repoURL` into a di
git clone [options] <repoURL> <dir> # clone the git repo from `repoURL` into a di
```

Has many (rarely used) options:

- `--bare` similar to `git init --bare`
- `--sparse` start with only the files in the root of the repository
- `-o/--origin <name>` use `<name>` instead of `origin` for the upstream repository
- `-b/--branch <name>` checkout the `<name>` branch instead of `master/main`
- `--recurse-submodules` initialize and clone submodules
- `-j/--jobs` the number of fetches to do at the same time

85% of the time you will use `git clone` without options.
Another 10% will just use the `--recurse-submodules` option.

git status

git-status - Show the working tree status

How you know the state of a repository.

```
git status [options]
git status [options] <path> ...
```

Has many options.
The most used options being:

- `-s`/`--short` output in "short-format"
- `--long` output in "long-format" (default)
- `--porcelain [<version>]` easy-to-parse format for scripts, with the API `<version>`
- `-v`/`--verbose` show the textual changes that are staged to be committed

Tracking a Project with Git

- Project must be in single directory tree.
- Usually don't want to track all files in directory tree
- Don't track binaries, derived files, temporary files, large static files, secrets, etc.
- Use **.gitignore** files to indicate files never want to track
- Use `git add <file>` to indicate you want to track *file*
- Careful: `git add <directory>` will recursively add **every** file in **directory**

git add

git-add - Add file contents to the index

```
git add [options] <path> ...
```

- `-n`/`--dry-run` don't actually add anything, just show what would be added
- `-f`/`--force` add ignored files
- `-i`/`--interactive` add interactively
- `-A`/`--all` add all files already in the index
- `-N`/`--intent-to-add` mark files as tracked but don't save their contents

git commit

git-commit - Record changes to the repository

```
git commit [options] [-m <message>] [--] <path> ...
```

- `-m/--message <message>` use `<message>` as the commit message (almost always used)
- `-a/--all` automatically stage all tracked files before committing
- `-C/--reuse-message <commit>` use the commit message from `<commit>`
- `--amend` replace the previous commit with a new one
- `--author <author>` use `<author>` instead of the current user
- `--date <date>` use `<date>` instead of the current date
- `--allow-empty` allow empty commits (useful for CI/CD pipelines)

if `--message` is not used, `git commit` will open an editor for you to write the commit message.
This allows you to write a longer, multi-line, commit message.