Git Bootcamp

Mike Gildersleeve – amgilder@cs.unh.edu

Overview

Git is a distributed Version Control System (VCS) or Source Code Management (SCM) system
- Tools to track and manage changes to a collection of files (typically in a folder) over time
- Allow comparisons between versions, rolling back to previous versions, collaboration with others, and more

Standard workflow is best thought of as generating a series of checkpoints within a given project – each checkpoint is a snapshot of its current state
- Can roll back to or branch off from any checkpoint – branches are just additional series of checkpoints and may be merged

Git is a separate application run locally on your system and it can operate entirely locally

However, it's often used in combination with a network service such as BitBucket, GitHub or GitLab
- Typically, use GitHub (https://github.com/) or GitLab (https://about.gitlab.com/) when sharing or collaborating with the general public
- Typically, use the departmental GitLab (https://gitlab.cs.unh.edu/) for most of your coursework and academic projects
- Can also use Git locally on your own system whenever it suits your needs to do so

Git itself is a command-line tool, but there are many graphical user interface (GUI) options – these may even be built into your editor or IDE
- We're going to focus on using it from the command line, so make sure you're entering the commands that follow into a terminal

A repository (or repo) is a workspace for tracking and managing the contents of a folder – typically, there's one repo per project
- By default, a repo encompasses a folder and all its subfolders – typically, this will be the primary project folder

Setup

To see if Git is already installed and, if so, what version you have → **`git --version`**
- If necessary, download and install Git for your system from https://git-scm.com/

Check the name and email you have configured for Git – if nothing appears, no value has been set

```
git config user.name
git config user.email
```

Set them as necessary – use your full name and UNH email when working with the departmental GitLab

```
git config --global user.name "Your name in quotes"
git config --global user.email "Your UNH email in quotes"
```

Basic local workflow

0) Ensure your current folder is part of a Git repo, initializing a new repo if necessary
   - To make use of Git, you must first be within a folder that is part of a repo
   - To see if your current folder is already part of a repo → **`git status`** → will show a fatal error if you're not inside a repo
   - ***Before creating a new repo, always check that you're NOT already inside a repo!***
   - To create a new repo using the current folder → **`git init`** → will create a hidden .git folder inside that folder for repo info
     - ***Never edit the files stored in the .git folder directly!***
     - Deleting this .git folder deletes the local repo
1) Work on stuff – create, edit, and delete files and subfolders – nothing particularly Git-specific about this part
   - Just use your editor or IDE to create and manipulate files locally as you normally would
2) Add (or stage) your changes – group a set of changed files (typically containing logically related changes) to prepare for a commit
   - Files and subfolders are not automatically included in your repo – each must be added by you
     - Once added, a file is considered to be tracked by the repo, files that haven't ever been added are considered untracked
   - Add specific files → **`git add <filename1 filename2 …>`** → use as many add commands and list as many files as you wish
     - Adding a folder automatically adds everything currently within that folder
     - Even when a file is being tracked, you must still re-add it to stage any recent changes for the next commit
   - Add all tracked files that have been changed since they were previously added → **`git add -u`**
   - Add all files and subfolders within the current folder → **`git add .`**
     - ***Note that adding changed files only puts the changes in the temporary staging area, not in the actual repo!***
     - Generally, it's best to add individual files and folders, since there are likely to be files and folders that don't need to be added
     - Typically, you only want to add files, such as source and config files, that are needed to construct your project
3) Commit – take all the changes added (or staged) since the previous commit and store them in the repo
   - Commit all currently staged changes → **`git commit -m "Your commit message here"`**
   - Each commit is an identifiable step along the path towards completion of your project
     - Each is identified by a unique commit hash – these are long, but usually the first few characters are sufficiently unique
   - You can return (or rollback) your project state to any commit you wish at any time and/or branch from each commit
   - Try to make your commits atomic – that is, each commit should encompass only a single feature, change, or fix
     - Results in frequent, relatively small commits rather than occasional, complex commits
   - Try to phrase your commit messages as if ordering Git to effect that feature, change, or fix
4) Repeat as needed from step 1

To instruct Git to ignore certain files or subfolders, list them, one per line, in a text file named .gitignore in the primary project folder

- End folder names with a slash (**/**) – lines can include patterns, such as **`*.exe`** to encompass all files with a **`.exe`** extension
- There are many .gitignore templates available for common situations at places like https://github.com/github/gitignore

Getting info about a repo

Check the current state of the repo that you are in → **`git status`**

- Reports things like the current branch, commit status, staged files, tracked files with changes, untracked files, and more
- Generates an error if used when not inside a folder that's part of a Git repo
- You'll find that this is by far the most frequently used of all the git commands

List all commits made to the repo → **`git log`**

- For each commit, the output includes its full hash, its author, when it was made, and the commit message, with the newest at the top
- For a more concise report → **`git log --oneline`**

Branching

Branches are separate timelines within a project – each repo can have many branches

- What is committed to one branch does not impact any others unless they are later merged – handy for "what if" explorations

At any given time, Git can only be focused on a single branch – generally, the master or main branch by default – often the "source of truth"

Branch pointers are like bookmarks that refer to distinct branches – technically, they point to specific commits within a branch

- HEAD is a branch pointer that indicates your current commit, which is always on your current branch and typically the latest on that branch

View available branch pointers → **`git branch`** → the active branch will be indicated by an asterisk (*)

Create a new branch off of HEAD → **`git branch <branch-pointer>`** → just creates the branch, doesn't move you to it

- Keep branch names to a single word that's descriptive of the branch's purpose

To move HEAD to a different branch → **`git switch <branch-pointer>`** → often follows the creation of a new branch

To rename the current branch → **`git branch -m <new-branch-pointer>`**

Merging

Branches can be thought of as different what-if scenarios that are completely independent of each other and the master or main branch

- Often, however, those branches need to be merged with other branches
- Common workflow is to treat the master or main branch as the "source of truth"
- Work on new features and experimental scenarios is done on divergent branches
- Those branches then get merged into the master or main branch later if their outcomes are deemed acceptable

A merge always combines a branch with the current HEAD – to change the current HEAD → **`git switch <new-head-branch>`**

To merge a branch with the current HEAD → **`git merge <branch-to-merge>`**

"Fast-forward" merges are the simplest – they just involve catching one branch up to another

- Arise when no additional commits have been made to the HEAD since the branch was diverged – just need to extend HEAD with changes
- Such merges will not produce any conflicts

Some merges, however, are potentially more complicated – may have conflicts that need to be resolved

- If no such conflicts exist, such a merge proceeds automatically
- But if conflicts do exist, they must be resolved manually before the merge can be completed
  - Git notifies us that conflicts exist and which files they're in – files will be marked up indicating conflicts
  - Our responsibility is to edit and save those files to resolve the conflicts – removing the conflict markers as we go
  - When we commit those changes to the repo, the merge should automatically complete
  - Some editors have tools to directly support this process

Traveling back in time

Git allows you to take your project back in time to any previous commit – all files in the working directory are set as they were at that time

First you need to know the hash of the commit you wish to target – usually you only need the first seven characters of a hash

- To find the commit hashes → **`git log --oneline`**

To undo all changes made since a previous commit → **`git reset --hard <commit-hash>`**

To undo only the changes made by a specific commit → **`git revert <commit-hash>`**

- Adds a new commit at the end of the branch that undoes whatever changes were made by the specified commit

Reverting is often preferable to resetting when you are using Git to collaborate with others

Remote repos

Although it's possible to keep everything local, Git is often used with remote repositories (aka, remotes)

- Commonly hosted at places like GitHub and GitLab
  - The department has its own GitLab server at https://gitlab.cs.unh.edu/ – login using your UNH credentials under the Wildcats tab
- Allows sharing and collaboration on a scale ranging from an individual to the entire globe – particularly common for open-source projects

Generally, the first step in working with a site such as GitHub or GitLab is to establish your account

- For gitlab.cs.unh.edu, you should receive an invitation email at your official UNH address with instructions

Once you have your account setup, the next step is to establish SSH keys so you don't need to use credentials

- The details vary slightly from system to system, but the process has some commonalities
  - First you need to have a pair of SSH keys on your local system – these are easily generated
    - `ssh-keygen -t ed25519 -C "`***`yourusername`***`@gitlab.cs.unh.edu"`
    - When prompted for the file name, you can generally press Enter to accept the suggested pathname
    - Specify a passphrase (or leave it blank, if your system is secure) – you'll need to enter this often, so make it easy to type
  - Then, you need to add that public key to your GitLab account
    - See: https://docs.gitlab.com/ee/user/ssh.html#add-an-ssh-key-to-your-gitlab-account

Once logged into your departmental GitLab account, use the **+** icon in the menu bar to create a new project – start with a blank project
- Name the project and choose its visibility – ***keep coursework private*** unless instructed otherwise by your instructor
- Can ignore other settings unless instructed otherwise
- Further instructions for several options appear in the empty repo page – use last option if you already have a local Git repo established
  - If the second command line generates an error regarding 'no such remote,' just ignore it and proceed

Cloning, pushing, and pulling

Cloning is the process by which one initializes a local repo that is linked to a remote repo
- Might be used to work locally on a remote repo published by others
- Also the easiest way to establish a link between your own local repo and a remote equivalent
  - Set up the remote repo on GitHub or GitLab using the web interface first, then clone it locally
    - This automatically configures the remote-local links, which can be done manually but can be tedious
  - Just put all project files into the cloned local directory, add them, commit, and eventually push (see below)

Typically remote repos are cloned using a URL to identify them – get this URL from the web interface for the remote repo

On your local system, make sure you're in the directory that you wish to be the parent of the newly cloned directory
- Check that it's not already part of local repo → **`git status`**
- Clone a remote repo → **`git clone <repo-url>`**
- Results in a new local directory appearing to act as the root directory of the cloned repo

Within that project root directory, you can add and commit as you would with any other local repo

But if you want those changes to be reflected in the remote repo, you must first push them to the remote repo
- Push all commits made since cloning (or previous push) to the remote repo → **`git push`**

If you want to update your working directory with changes made to the remote repo, you need to pull them yourself
- Pull all commits and their associated changes from the remote repo and update the local working directory → **`git pull`**

Diff'ing things

Git makes it easy to see the differences between two things, such as versions of a file, branches, or commits
- Show all unstaged changes → **`git diff`**
- Show all staged changes → **`git diff --staged`**
- Show all changes, both staged and unstaged → **`git diff HEAD`**
- Show all changes that exist between two different branches → **`git diff <branchname1>..<branchname2>`**
- Show all changes that exist between two different commits → **`git diff <commithash1>..<commithash2>`**
- Each of the above can be limited to a single file by adding the filename → **`git diff <filename>`**

Many Git users discipline themselves to diff each file before staging them, as this can help avoid erroneous or incomplete commits

Reading diff output is a bit complicated
- Results are divided up into sections, one per thing being diff'd
- First line of each section starts with the word 'diff' and shows the things being compared in that section, such as versions of a single file
  - One version is labeled a and the other b
- Second line shows the hashes of the two things being compared and some advanced info about the mode of that file
- Third and fourth lines show the markers (- for a and + for b) associated with each of the two things being compared
- The remainder of each section consists of one or more chunks
  - Each chunk shows a change with a bit of its before and after context
  - Each starts with a chunk header that contains two pairs of numbers between '@@' indicators
    - The first number in each pair indicates on what line of the file the chunk's extraction begins
      - Although they look like positive and negative values, the – and + are just the markers from above being utilized
    - The second number indicates the number of lines extracted for the chunk
  - In the chunk itself
    - Lines beginning with a space (and displayed in white) are unchanged lines provided as context
    - Lines beginning with a – (and displayed in red) are changed lines from version a
    - Lines beginning with a + (and displayed in green) are changed lines from version b

To leave the diff results, press the q key