Git: A brief overview

AUTHOR
Christian Osendorfer

Introduction

Note

Disclaimer: Most of the following text is taken from https://inst.eecs.berkeley.edu/~cs61b/fa19/docs/using-git.html

Version control systems are tools to keep track of changes to files over time. Version control allows you to view or revert back to previous iterations of files. Some aspects of version control are actually built into commonly used applications. Think of the undo command or how you can see the revision history of a Google document.

In the context of coding, version control systems can track the history of code revisions, from the current state of the code all the way till it was first tracked. This allows users to reference older versions of their work and share code changes with other people, like fellow developers.

Version control has become a backbone of software development and collaboration in industry. In this class, we will be using Git. Git has excellent documentation so we highly encourage those who are interested to read more about what will be summarized in the rest of this guide. Intro to Git

Git it a distributed version control system as opposed to a centralized version control system. This means that every developer's working copy of the code is a complete repository that can stand apart from a shared, centralized server. This concept leads to our ability to use Git locally on our own computers.

Your computer already has Git installed (see the first homework sheet!) as a command line tool. You also should be able to log into studilab.

First, read through the following documentation, then start your own (almost empty) code repository on studilab **under your username**, as described to the end of this document and clone it to your own local computer. Only then use the other commands (`add`, `commit` and `push`) to get your code from the first programming practical onto studilab.

Local Repositories

Initializing Local Repositories

Let's first start off with the local repository. A **repository** stores files as well as the history of changes to those files. In order to begin, you must initialize a Git repository by typing the following command into your terminal while in the directory you wish to make it a local repository.

```
        git init
```

When you initialize a Git repository, Git actually creates a `.git` subdirectory that is a *hidden folder* on most operating systems (indicated by the `.` as the first character in the folder name). But it's there. The UNIX command `ls -la` will list all directories, including your `.git` directory, so you can use this command to check that your repo has been initialized properly.

Tracked vs. Untracked Files

Git repos start off not *tracking* any files. In order to save the revision history of a file, you need to track it. The Git documentation has an excellent section on recording changes.

Files fall into two main categories:

- untracked files: These files have either never been tracked or were removed from tracking. Git is not maintaining history for these files.
- tracked files: These files have been added to the Git repository and can be in various stages of modification: unmodified, modified, or staged.
  - An unmodified file is one that has had no new changes since the last version of the files was added to the Git repo.
  - A modified file is one that is different from the last one Git has saved.
  - A staged file is one that a user has designated as part of a future commit.

The following Git command allows you to see the status of each file:

```
        git status
```

The git status command is extremely useful for determining the exact status of each file in your repository. If you are confused about what has changed and what needs to be committed, it can remind you of what to do next.

Staging & Committing

A **commit** is a specific snapshot of your working directory at a particular time. Users must specify what exactly composes the snapshot by staging files.

This command lets you stage a file (called `FILE`). Before staging, a file can be untracked or tracked & modified. After staging and committing it, it will be a tracked file.

```
        git add FILE
```

Once you have staged all the files you would like to include in your snapshot, you can commit them as one block with a message.

```
        git commit -m MESSAGE
```

Your message (which should be a sentence surrounded by `" "`) should be descriptive and explain what changes your commit makes to your code. You may want to quickly describe bug fixes, implemented classes, etc. so that your messages are helpful later when looking through your commit log.

In order to see previous commits, you can use the `log` command:

```
        git log
```

The Git reference guide has a helpful section on viewing commit history and filtering log results when searching for particular commits. It might also be worth checking out gitk, which is a GUI prompted by the command line.

As a side note on development workflow, it is a good idea to commit your code as often as possible. Whenever you make significant (or even minor) changes to your code, make a commit. If you are trying something out that you might not stick with, commit it (perhaps to a different **branch**, something we don't discuss in this note!).

> **Important**
>
> If you commit, you can always revert your code or change it. However, if you don't commit, you won't be able to get old versions back. So commit often!

Undoing Changes

The Git reference has a great section on undoing things. Please note that while Git revolves around the concept of history, it is possible to lose your work if you revert with some of these undo commands. Thus, be careful and read about the effects of your changes before undoing your work.

- **Unstage** a file that you haven't yet committed:

  ```
          git reset HEAD [file]
  ```

  This will take the file's status back to modified, leaving changes intact. Don't worry about this command undoing any work.

  Why might we need to use this command? Let's say you accidentally started tracking a file that you didn't want to track. (.class files, for instance.) Or you were doing some work and thought you would commit but no longer want to commit the file.

- **Amend** latest commit (changing commit message or add forgotten files):

  ```
          git add [forgotten-file]
          git commit --amend
  ```

  Please note that this new amended commit will replace the previous commit.

- **Revert** a file to its state at the time of the most recent commit:

  ```
          git checkout -- [file]
  ```

  This command is useful if you would like to actually undo your work. Let's say that you have modified a certain file since committing previously, but you would like your file back to how it was before your modifications.

> **Warning**
>
> **Reverting** is potentially quite dangerous because any changes you made to the file since your last commit will be removed. Use this with caution!

Remote Repositories

Thus far, we have worked with local repositories. We'll now discuss remote repositories, Git repositories that are **originally** not on your own computer. They are on other machines, often servers of sites that host repos like GitHub or GitLab. These are website services that can store your Git repositories, they **are not equivalent to Git**. Rather, it is a convenient way to store your code online. They also have many features that make sharing and developing code collaboratively more simple and efficient. At HAW Landshut, we run our own GitLab instance on-premise, available under studilab.

Private vs. Public Repos

By default, repositories on studilab are private. Don't change that for the time being!

Creating a repository on studilab

The following is taken from the GitLab documentation. Read closely through the information shown on the website while following the steps. Use your own user when asked for e.g. specifying owners or the path of the repository.

In order to create a blank project:

1. On the left sidebar, at the top, select **Create new (+)** and **New project/repository**.
2. Select **Create blank project**.
3. Enter the project details:
   - In the **Project name** field, enter the name of your project. The name must start with a lowercase or uppercase letter (`a-zA-Z`), digit (`0-9`), emoji, or underscore (`_`). It can also contain dots (`.`), pluses (`+`), dashes (`-`), or spaces.
   - In the **Project slug** field, enter the path to your project. The GitLab instance uses the slug as the URL path to the project. To change the slug, first enter the project name, then change the slug.
   - Set the **Visibility Level** to *private*.
   - To create a `README` file so that the Git repository is initialized, has a default branch, and can be cloned, select **Initialize repository with a README**.
4. Select Create project.

In the repository, investigate what you can do with the `README` file.

Cloning a Remote

There are often remote repos with code that you would like to copy to your own computer. For example a repository of yours on studilab. In these cases, you can easily download the entire repo with its commit history by **cloning the remote**:

```
        git clone [remote-url]
        git clone [remote-url] [directory-name]
```

The top command will create a directory of the same name as the remote repo. The second command allows you to specify a different name for the copied repository.

When you clone a remote, the cloned remote is *associated with your local repo* by the name **origin**. This is by default because the cloned remote *was the origin for your local repo*.

Pushing Commits

You may wish to update the contents of a remote repo by adding some commits that you made locally. You can do this by **pushing your commits**:

```
        git push [remote-name] [remote-branch]
```

For example, let's say that I cloned a repo then made some changes on the *master branch (we don't know about branches, so this is okay to ignore for now). I can give the remote repository my local changes with this command:

```
        git push origin master
```

Fetching & Pulling Commits

There are also times that you'd like to get some new commits from a remote that are not currently on your local repo. For example, you may have cloned a remote created by a partner and wish to get their newest changes. You can get those changes by **fetching** or **pulling** from the remote.

- **fetch**: This is analogous to downloading the commits. It does not incorporate them into your own code.

  ```
          git fetch [remote-name]
  ```

  Why might you use this? Your partner may have created some changes that you'd like to review but keep separate from your own code. Fetching the changes will only update your local copy of the remote code but not merge the changes into your own code.

  For a more particular example, let's say that your partner creates a new branch on the remote called `fixing-ai-heuristics`. You can view that branch's commits with the following steps:

  ```
          git fetch origin
          git branch review-ai-fix origin/fixing-ai-heuristics
          git checkout review-ai-fix
  ```

  The *second command* creates a new branch called `review-ai-fix` that **tracks** the `fixing-ai-heuristics` branch on the `origin` *remote*.

- **pull**: This is equivalent to a fetch + merge. Not only will pull fetch the most recent changes, it will also merge the changes into your HEAD branch. `{.bash}      git pull [remote-name] [remote-branch-name]` Let's say that my boss partner has pushed some commits to the master branch of our shared remote that fix our AI heuristics. I happen to know that it won’t break my code, so I can just pull it and merge it into my own code right away.

```
        git pull origin master
```

Outlook

If you look around the Git documentation, you will find information to other, slightly more advanced topics like branching, merging and adding a remote repository.