Tutorial: Version Control

Source control, intro to Git, simple git commands

Introduction

For this tutorial, you will explore the version control features of Git. You will first add a file to the repository, then you will add information to the file while commiting changes along the way. Last, you will reset the file to a previous commit. Follow the tutorial and submit the last output from **git log** to learning suite by the end of class.

Definitions

Directory - means a folder on your computer
Terminal - the black box on mac or linux, on Windows it is called the command prompt

Commands You Will Learn

- git add
- git commit
- git init
- git log
- git reset

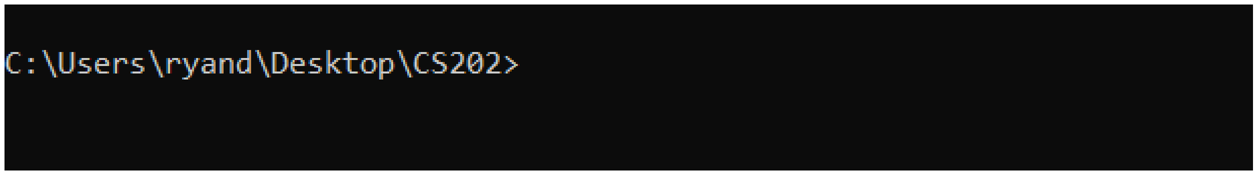
Tutorial

Part 1: Adding a File to a New Repository

We will start by creating a new repository and adding a file to it. Ask the instructor, a TAs or a classmate if you get stuck.

1. Create a new directory on your computer named "CS202"

2. Navigate to the directory using terminal, terminal should now look something like this:



```
C:\Users\ryand\Desktop\CS202>
```

3. Type **git init** to create a new repository at that directory

This is the first step in creating any project that you want to use git with. Git commands only work on git repositories. If you forget to initialize the repository, git will respond with this message: “fatal: not a git repository (or any of the parent directories): .git”

4. Create and save a text document called “Text.txt” to your folder
    a. In Windows, you can create the file by right clicking inside your folder and clicking “New Text Document”. From Mac or Linux, you can create the file by typing “touch Text.txt” from within the terminal.
5. Add the file to your next commit by typing in **git add <filename>** (“Text.txt” is our filename)

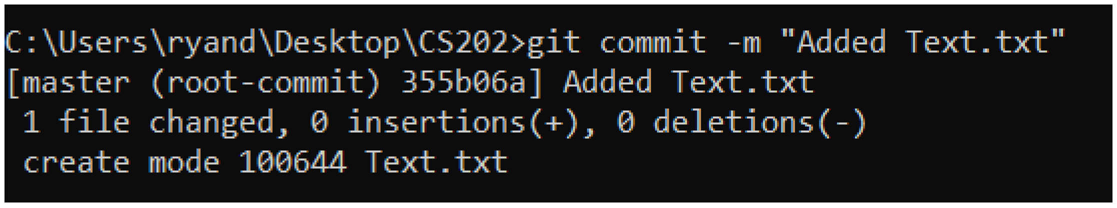
    This command marks that you have added or changed a file and want it to be part of the next commit. Typing **git status** tells you any of the files that you have changed and might want to add.

6. Commit the file(s) you add to your repository with the command

    git commit -m “<message>” (with “Added Text.txt” as your message)

    Git commit makes a new version of your project. We want our commit messages to be as clear as possible so if something goes wrong and we have to roll back to a previous version, we will be able to tell what version we want.

    You should see something like this after commiting your file.



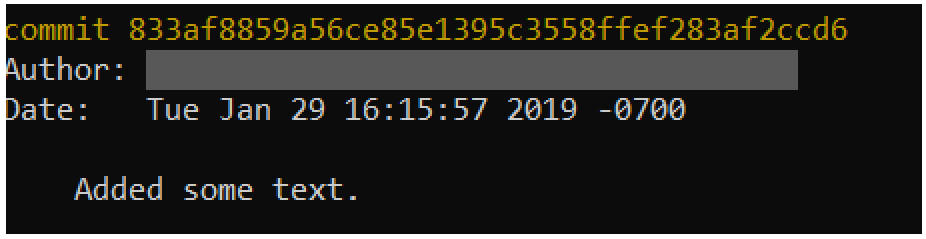
```
C:\Users\ryand\Desktop\CS202>git commit -m "Added Text.txt"
[master (root-commit) 355b06a] Added Text.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Text.txt
```

Part 2: Using Version Control to Save and Revert Changes

Next, we will add some text to a file, commit it, then make some unwanted changes to the file, and finally reset the file back to a previous state.

1. Modify Text.txt by adding the text “Text I want.” to the file.
2. Save Text.txt.
3. **Add** the file to your commit like you did in the previous example.
4. **Commit** the changes to your repository with the message “Added some text.”

5. Check to make sure your file has the text “Text I want” in it.
6. Modify your file by adding the following text “Text I don’t want.”
7. Save your file.
8. **Add** the file to your repository.
9. **Commit** the file with the message “Bad text.”
10. Check to make sure your file has the text “Text I don’t want” in it.
11. Type **git log** to view all the commits you have made to your repository.
12. You should see three commits, each with these respective messages.
    a. “Bad text”
    b. “Added some text”
    c. “Added Text.txt”
13. Now we will get rid of the unwanted text in our file by resetting our repository to the commit before we added the bad text.
14. Find the *hashcode* (long string of numbers/letters) for the commit that you want to reset to. In our case, it is the commit with the message “Added some text.”



```
commit 833af8859a56ce85e1395c3558ffef283af2ccd6
Author: [redacted]
Date:   Tue Jan 29 16:15:57 2019 -0700

    Added some text.
```

15. Hard reset your repository to a previous version by typing **git reset --hard <hashcode>** using the hashcode from the previous step. (you can just type the first 6 characters of the hashcode)

    Note that there are different ways of getting rid of a bad commit.

    - **git reset --hard <hashcode>** will delete all file changes and commits after the chosen commit.
    - **git reset --soft <hashcode>** will delete all commits, but keep all file changes after the chosen commit.

    - **git revert <hashcode>** will delete a chosen commit and its file changes, but will keep the commits before and after the deleted commit. (This is slightly more complicated, but worth learning)

16. Confirm that your reset worked by checking the file and verifying that “Bad text” is no longer in your text file.
17. Type **git log** to see that your bad commit is no longer in the git log.

Before You’re Done

Before you’re done, we want you to see how git is valuable in software engineering. Read the text below, and then copy the output from the last **git log** command and submit it to learning suite. You may submit a screenshot of your git log output if you prefer.

Git in Software Engineering

When developers create big projects, they will often use a version control system (like git) to create “checkpoints” of their progress.

For example, these are some of the commits you might see in a project.

- Added main.cpp
- Created user menu
- Created Item class
- Implemented the “add item” option on the user menu
- etc.

Sometimes changes that a developer makes will break the existing code. In the worse cases, the developers might not even remember how to get the project back to a working state. Fortunately with git, developers have the option of resetting their project back to a working state if they need to. This also allows developers to experiment with different potential changes and revert back to a state from before the changes were made.

Git has many more features that were not covered in this tutorial, including features that support group collaboration on file changes. These features are essential for development teams working on the same code base, where multiple developers may edit the same files. These group collaboration features will be explored in the next tutorial.