GIT INTRODUCTION

Maseeh College of Engineering & Computer Science

CS202

Karla Fant

2022 rev 1

Student Name:______________________________________

Git Introduction

CS202

2022

Revision 1

Computer Science

Karla S Fant

Student Name:____

PSU ID #: ____

CS Login Name: ______________________

Computer Science

Contents

Expectations and Overview

IMPORTANT

Everyone will need an MCECS (CS) account.

This account is needed for
both lab work and programming assignment work.

Visit FAB 88 to obtain your account for the first time.

This needs to take place PRIOR to the first lab!

Lab Session Policies - Expectations

(pending additions)

Git Background Information

What is Git?

Git is a version control system. A version control system is a set of tools for tracking changes to files you regularly update. Some benefits of a version control system are more organization of large code bases, the ability to coordinate collaboration, and a safety net in the event of lost data or development mistakes. Although version control can be important for any kind of project, you will be learning how to use it with your programming work.

For example, let's say you're working on your programming assignment on Linux at 3:00 am, but in the morning you realize that you really wish you could go back to how it was before any of your changes. What can you do? If you used Git, then you can restore that previous version. Git can also be used for group projects where you're all working on the same code, to turn in assignments, and as well as other uses that we won't get into.

What is GitLab?

GitLab is a web-based Git repository that provides free open and private repositories. Essentially, this is where everything you "backed up" or everything you've "committed" goes to. There are similar hosted platforms – such as GitHub or Bitbucket – that offer many of the same features as GitLab. Learning how to use GitLab now will better prepare you to use other platforms later.

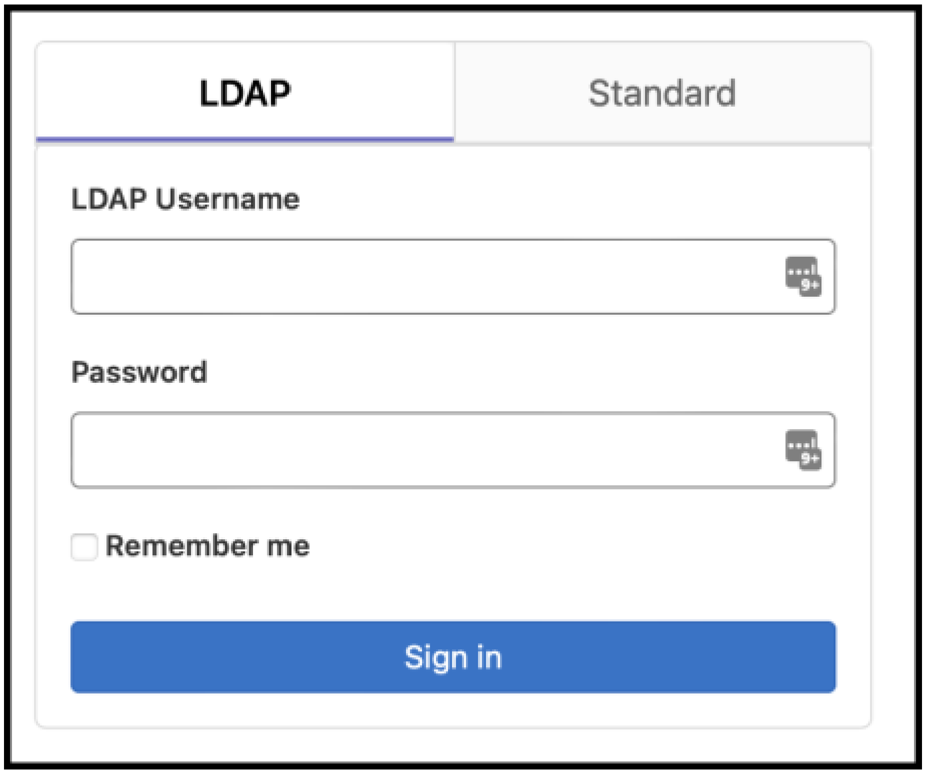
NOTE: In this document some screenshots have been modified to remove sensitive information. However, you will still be typing your own information when prompted.

Exercise 1: Creating Your GitLab Repository

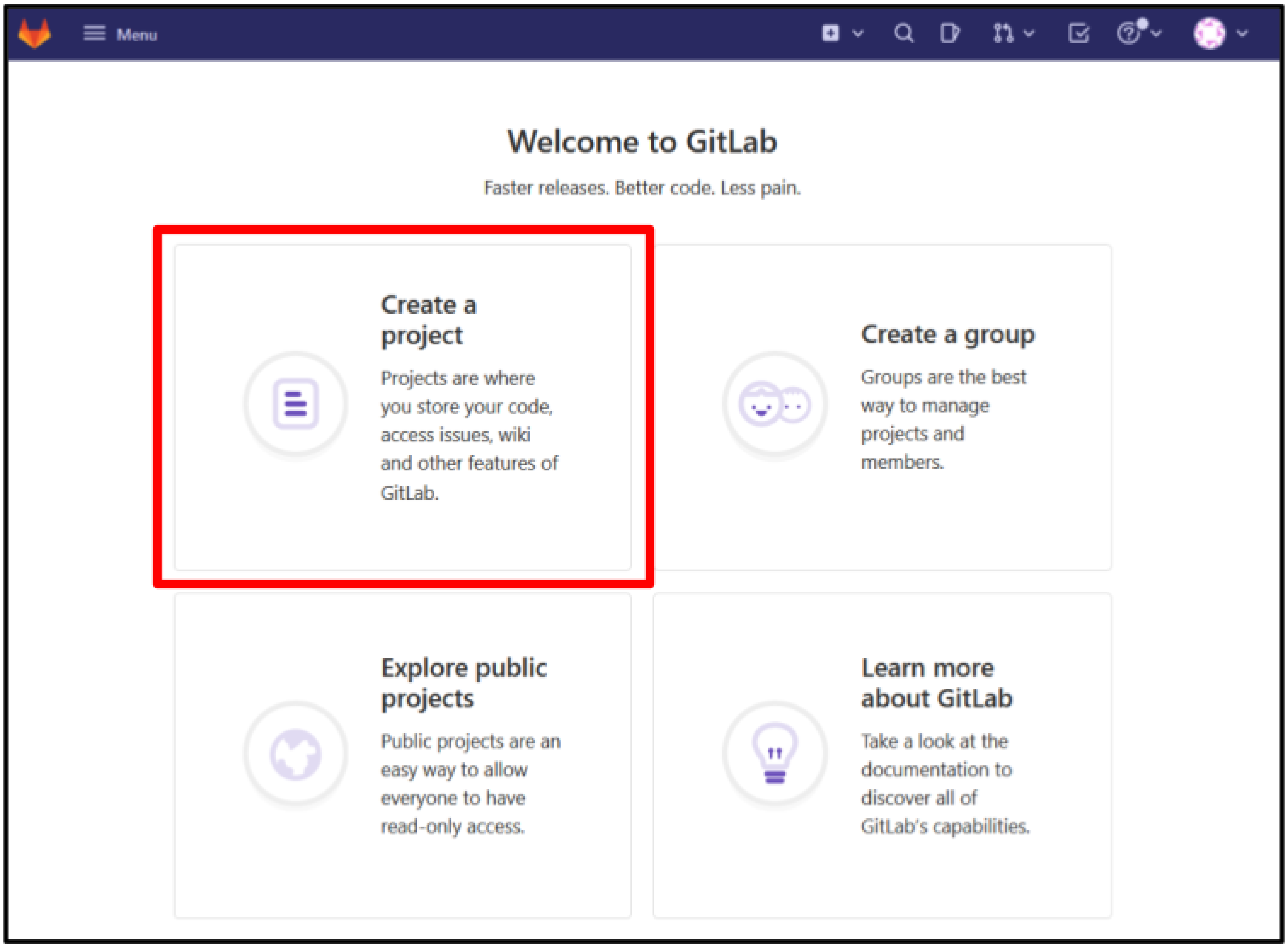
1. Log into gitlab.cecs.pdx.edu with your LDAP credentials

   (MCECS username/password)

   https://gitlab.cecs.pdx.edu/users/sign_in



2. Create a new project.



Welcome to GitLab

Faster releases. Better code. Less pain.

Create a project

Projects are where you store your code, access issues, wiki and other features of GitLab.

Create a group

Groups are the best way to manage projects and members.

Explore public projects

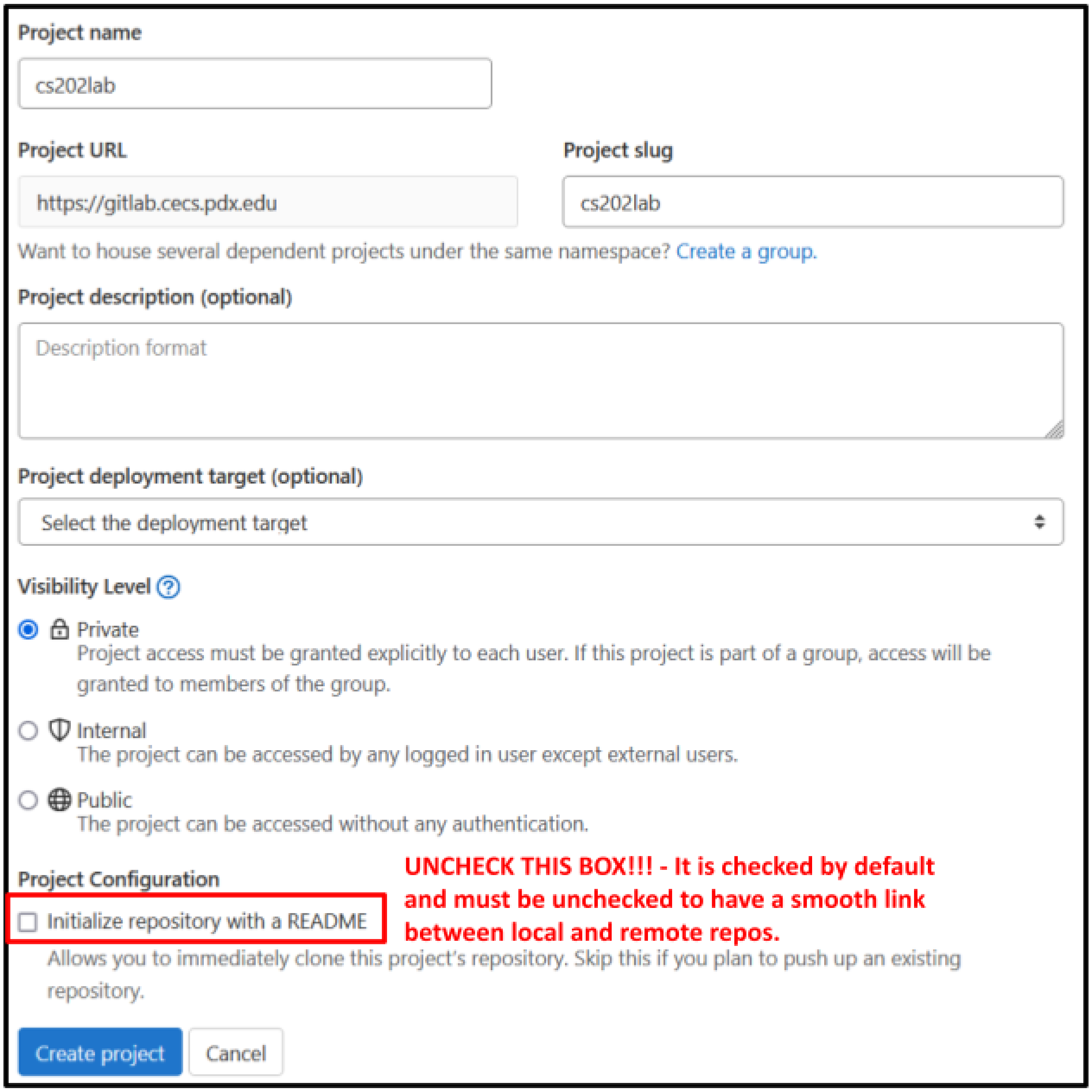
Public projects are an easy way to allow everyone to have read-only access.

Learn more about GitLab

Take a look at the documentation to discover all of GitLab's capabilities.

3. Give your new project some information.

NOTE: You MUST uncheck the "Initialize repository with a README" - You will do this later manually.



Project name

cs202lab

Project URL

https://gitlab.cecs.pdx.edu

Project slug

cs202lab

Want to house several dependent projects under the same namespace? Create a group.

Project description (optional)

Description format

Project deployment target (optional)

Select the deployment target

Visibility Level

● Private
Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.

○ Internal
The project can be accessed by any logged in user except external users.

○ Public
The project can be accessed without any authentication.

Project Configuration

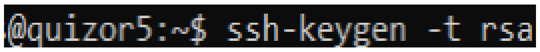
☐ Initialize repository with a README
Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

UNCHECK THIS BOX!!! - It is checked by default and must be unchecked to have a smooth link between local and remote repos.

4. Create an SSH key while logged into cs202lab.cs.pdx.edu.

NOTE: It is very important to follow these next few instructions accurately. Generating an SSH key allows you to add it to your GitLab to skip entering your username and password each time you push, pull, fetch, etc.

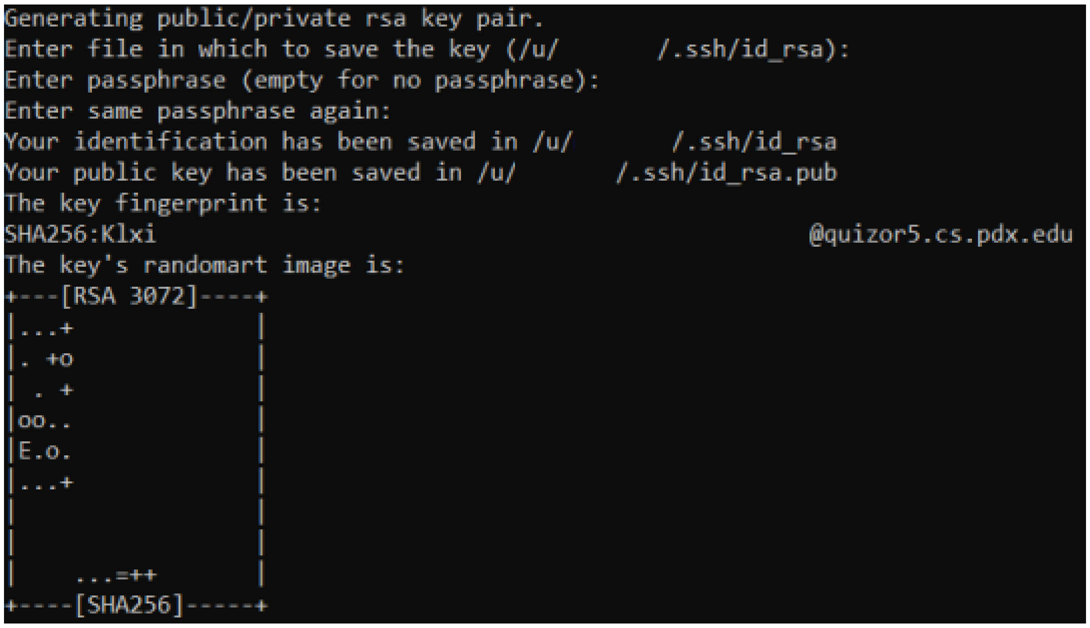
4.1. Type `ssh-keygen -t rsa`



```
@quizor5:~$ ssh-keygen -t rsa
```

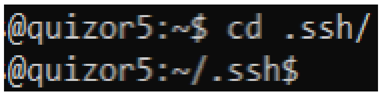
4.2. Once the `Enter file in which to save the key (location):` has been displayed, **hit enter 3 times.** This sets the next few prompts to a default value - or no value - and for this tutorial they are **not required.**

NOTE: A few prompts will automatically post informing you of where the keys have been saved as well as the fingerprint and the randomart image. The only thing you will want to keep in mind is where the keys were saved..



```
Generating public/private rsa key pair.
Enter file in which to save the key (/u/[redacted]/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /u/[redacted]/.ssh/id_rsa
Your public key has been saved in /u/[redacted]/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:Klxi[redacted]@quizor5.cs.pdx.edu
The key's randomart image is:
+---[RSA 3072]----+
|...+             |
|. +o             |
| . +             |
|oo..             |
|E.o.             |
|...+             |
|                 |
|                 |
|    ...=++       |
+----[SHA256]-----+
```

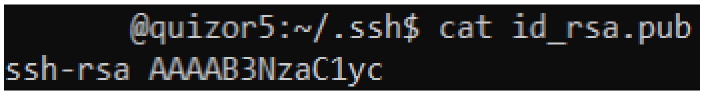
4.3. From your home directory (`username@mylab:~$ `)Type `cd .ssh` to move into the .ssh directory where your keys are stored.



```
@quizor5:~$ cd .ssh/
@quizor5:~/.ssh$
```

4.3.1. **Lab server required step:** type `chmod 600 id_*` upon entering the `.ssh` directory. This changes the permissions for the public and private key files. If we do not do this step, GitLab will produce an error later.

4.4. Once inside the SSH directory type `cat id_rsa.pub`.



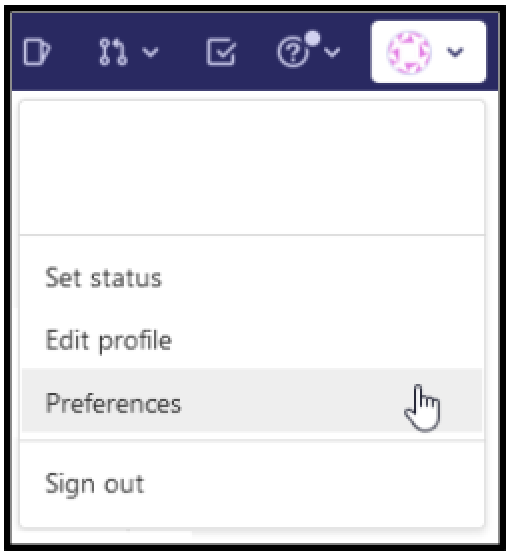
```
       @quizor5:~/.ssh$ cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc
```

NOTE: Yours will be *much* longer.

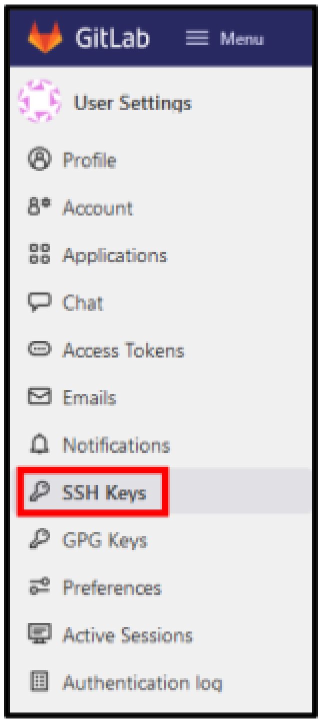
4.5. Copy this entire string to your clipboard. This will vary from system to system how this is performed.

5. Give your GitLab account the public SSH key.

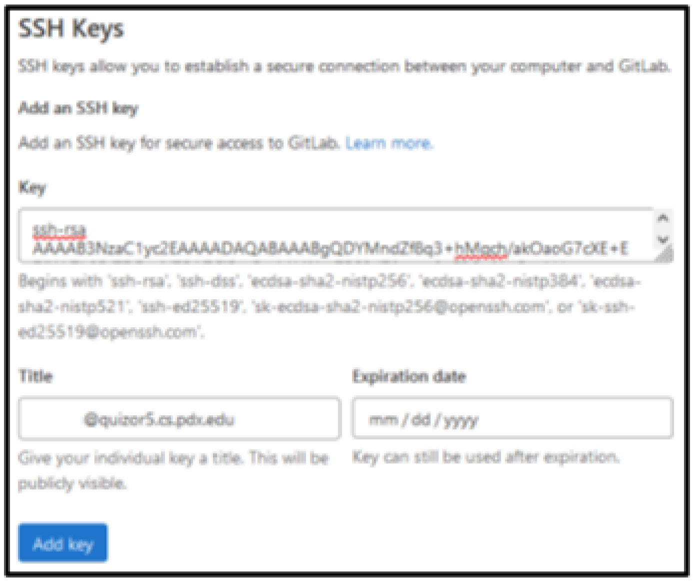
5.1. Go to your GitLab Preferences.



5.2. Click on **SSH Keys.**



5.3. Paste the RSA key you copied earlier into the **Key** block. Then click **Add Key.**



SSH Keys

SSH keys allow you to establish a secure connection between your computer and GitLab.

Add an SSH key

Add an SSH key for secure access to GitLab. Learn more.

Key

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABgQDYMndZf8q3+hMqch/akOaoG7cXE+E
```

Begins with 'ssh-rsa', 'ssh-dss', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', 'ssh-ed25519', 'sk-ecdsa-sha2-nistp256@openssh.com', or 'sk-ssh-ed25519@openssh.com'.

Title: @quizor5.cs.pdx.edu

Give your individual key a title. This will be publicly visible.

Expiration date: mm / dd / yyyy

Key can still be used after expiration.

Add key

Congratulations! From here you are ready to now connect your lab directory to this empty remote repository. **You can repeat this process on the linux.cs.pdx.edu environment!** Next you will set up your local repository that will connect to this new empty repository on GitLab.

Exercise 2: Linking Repos

This section demonstrates how to set up and link a **local repo** (files on your system) to a **remote repo** (a repository on a service such as GitLab, Github, Bitbucket). It's important to keep in mind that using git for version control is to maintain good working code as it evolves, and with this the ability to return to a more stable version of the code in the event your current working tree runs afoul and you need to roll back.

To learn more information about a particular command, type `git help <cmd name>` without the angle brackets. For this section of the tutorial, use your existing GitLab repo made in the previous section. You will be making a complete backup of all your lab files.

<u>Connecting Local and Remote Repositories</u>

1. Log into the **cs202lab.cs.pdx.edu** environment.

2. Navigate to the **CS202** folder.

3. Type `git init` - this will create a local repository on the linux system.

  3.1. Type `ls -a` to verify there is now a new directory called `.git` - this is the directory that stores all the information related to the local repo.

NOTE: If at any point there is a critical error in this process and you are unable to proceed, simply remove the .git directory by typing `rm .git -fr` and start from step 3. **DO NOT DO THIS FOR ALL FUTURE GIT REPOS YOU USE OUTSIDE OF LAB.** There are safer and cleaner ways to recover from an issue, but since you are working and learning in the lab environment this is okay for now. You will absolutely need to avoid this in the future as it can throw off version control in a very terrible and dangerous way.

4. Next, type `git add .` - Using the '.' as a filename will add **all** of your lab files. This will also add any changes you make to the repo that can be **staged** prior to a **commit**. Staging means preparing which files you would like to send your revisions to in the remote repository.

    4.1. It is good practice to only stage files you are certain you want to push that are in a good working state. If you are still working on one file, but another is finished for now, only add that finished file. To add individual files, type `git add <filename>`.

    4.2. To see what was added, type `git status` - you should see all files waiting to be committed and pushed. If you only see directories listed, know that it includes all files in that directory.

5. Type `git commit -m "First commit"` - this commits all our **staged** files that you added.

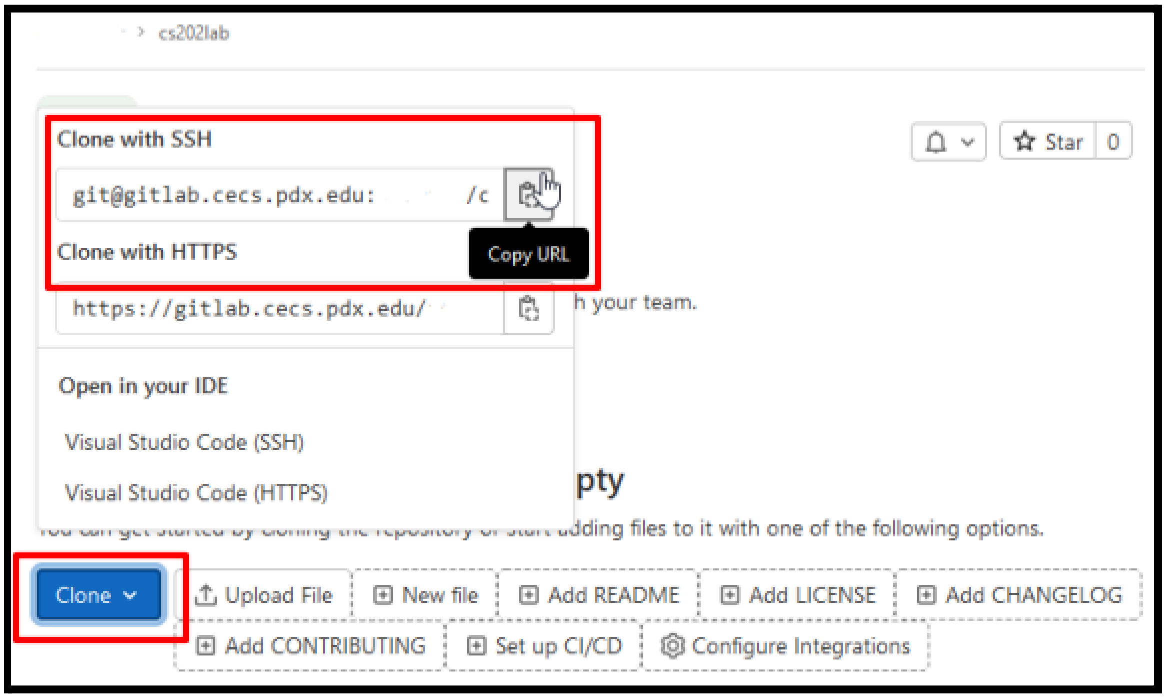
    5.1. Commit means we have saved these changes locally and are ready to push the files to the remote repository. The `-m` allows a message to be written along with the commit. You should always explain briefly which changes were made to the file.

6. With your first commit completed, you now have your first working **branch**. A branch allows you to differentiate versions of the project if desired.

    6.1. You **MUST** rename this branch to match what you have on GitLab. GitLab will always set the first branch as **main**. Your **local repo branch must have the same name.**

    6.2. Type `git branch -M main` - this will change the local repos branch name to **main.** Next you will connect up your local repo to the remote one.

7. First, return to your GitLab repo and click on **Clone,** then click on the copy button for **Clone with SSH**.



```
git@gitlab.cecs.pdx.edu:        /c
https://gitlab.cecs.pdx.edu/
```

7.1. Return to your terminal and type `git remote add origin` then **paste** the link you just copied from GitLab. It should look like this.

```
@quizor1:~/CS202$ git remote add origin git@gitlab.cecs.pdx.edu:      /cs202lab.git
```

NOTE: No news is good news! If you have received an error message, notify a TCSS.

8. Now you will **push** your committed files to the remote repo on GitLab.

8.1. Type `git push -u origin main` - This sets up the association between the origin you set up in **step 7.1** and the current branch in your local repository.

8.2. Return to GitLab and refresh the page on your project. Your files should now be uploaded. If there was an error, get help from a TCSS.

If all went well, **congratulations!** You now have a local repo on the lab server that is able to push and save changes to a remote repository on GitLab!

Good Git Tips

- Always `git pull` before you start coding. This will pull any changes you made somewhere else to another machine. While in the lab this isn't going to typically apply, but if you start working on personal projects and jump from machine to machine **always** do a pull before you start.
    - This applies for both working alone or in a group.
    - You can perform a `git fetch` beforehand if you aren't quite sure if you want to pull right away.
- Not sure if you are behind or ahead of the current branch? Type `git status`. This will let you know your current position in the branch.
    - This will also tell you what files have been changed/removed that need to be added prior to a commit!
- Only add changes you really need to push. Sometimes you will add all changes and that's okay, but it's also good practice to avoid doing this to prevent conflicts that could arise.
- Need a refresher on what changes you made previously? Type `git log` to see all previous commits to the repo.
    - Want less information? Do `git log —-oneline` instead.

Exercise 3: Changes and Version Recovery

Navigate to the CS202 lab directory. Create and edit a new C++ file, name it whatever you like, and put this code in it:

```
#include <iostream>
using namespace std;
int main()
{
      cout << “LET’S GIT GOIN\n”;
      return 0;
}
```

Save the file and close it. From the Linux prompt enter `git status` again. This time, a new message shows up. Git noticed that a new file has shown up in the directory and is letting you know that it needs to be added to the stage. Let’s add this file and check the status again:

```
git add *.cpp
```

```
git status
```

The new file is now **staged**. A staged file is one that is ready to be **committed**, and a commit is like a snapshot of the current version of the project. Staging and committing will be used very often when working with git.

When making a commit, it is a very good idea to document it. You will want to have a record of what changes were made since the last commit. This will help if you need to revert to an old version, and it helps with communication among collaborators.

Now let’s commit this new file. In the command below you’ll see the -m flag followed by quoted text. If you run the command ‘git commit’ without the flag and text, git will open your default text editor for you to enter your commit message. If you do this, just enter your message, save, and close. With the -m flag, you can enter the message on the command line:

```
git commit -m “New .cpp file with code”
```

If you call ‘git status’ now, you should see a message similar to the very first time you called it in this tutorial. *Consider this*: If you were now to compile this c++ file then call ‘git status’ what would it say? Think about it, then try it out.

With git, you can see *exactly what has changed* in your files since your last commit. To do this, first make a change to the program you created:

```
Replace   cout << “LET’S GIT GOIN\n”;
with      cout << “GIT PUNS ARE EASY\n”;
```

Save and close the file, then let’s see how git shows us the changes we just made:

```
git diff
```

You should see a snippet of code from the c++ file with the original code (denoted with a ‘-‘) and the new code (denoted with a ‘+’).

Since you made a change to a tracked file, you will need to add it to the stage again before you can commit it. Go ahead and try to commit before you add. It will give you the same output if you called `git status`. This is git’s way of telling you that you have something else to do before it will commit. Once you’ve tried that, add the file and commit. To add all files in the repo to the stage you can call `git add -A` or `git add .` (the period represents the current directory).

To see all the commits you’ve made so far, call `git log` and a list of all commits by all authors will be shown with the commit message, date-stamp and unique commit identifier. We can use the log output to experience some of the real power of a version control system. We’ll get the unique id for one of our commits then do some time traveling:

```
git log --oneline
```

There should be a condensed commit log with a unique id and the commit message you entered for each commit. Either write down- or make sure you can refer to- the *earliest* commit id. This will be the last one on the list. We’ll use this number to **checkout** an old commit. Checking out is how you use git to switch to different versions of a project.

Before we visit an earlier version, make sure you have *saved and closed* your file, then run this command, replacing ‘my_program’ with your filename:

```
cat my_program.cpp
```

You should see an output of your code in the Linux shell. We’re doing this as a basis for comparison for later. Take note of what it says or make sure you can see it later.

Note: `cat` is a Linux tool for concatenating and printing files. It’s not specific to git and we are only using it to view code in this example.

To go to an old version of your code, run the following command, replacing `#######` with the commit id from before, *and do not forget the period at the end as this pulls the previous files at the moment of that commit*:

```
git checkout ####### .
```

Now run the same `cat` command from before to see what your code looks like. It should show code from the first version of the program. You can even open the file in your text editor and change it again. Let's do that to get a feel for working on multiple versions of the same program:

Replace `cout << “LET’S GIT GOIN\n”;`
with `cout << “LET’S GIT STARTED\n”;`

Then save, close, stage (add) and commit. You now have three versions of this program, and you can use the `git checkout` command to move from one to the other.

NOTE: To return from a checkout if there were no changes made:

```
git checkout main
```

Your Average/Expected Workflow:

This is a general outline of your basic interactions with git while you develop your projects:

1. Create your project directory and initialize it for git
2. Create your project files
3. Add project files to the stage and commit changes
4. Work on your project
5. Repeat 3, 4.

Along the way, check the status and log as needed, and try checking out older commits to help understand development problems better.