Faculty of Computer Science, Dalhousie University *5-Feb-2020*

CSCI 4152/6509 — Natural Language Processing

Lab 4: Git and GitLab Tutorial

Lab Instructor: Dijana Kosmajac, Tukai Pain
Location: rm1102, Mona Campbell building
Notes author: Vlado Keselj

Git and GitLab Tutorial

Lab Overview

You will learn and refresh your knowledge about:

– GitLab Web interface
– How to checkout projects in Git
– Adding and deleting files and directories to Git and GitLab
– Committing and pushing your changes
– Checking out previous commits
– Elements of collaborative work: creating branches
– Merging branches and resolving conflicts

Required operations and files to be submitted:

1. `README.md` file created in GitLab
2. directory `lab4g` and file `id_rsa.pub` in it
3. directory `lab4g` should contain `explore.pl` and `Shakespeare__Hamlet-25k.txt`, with the commits for versions 1.0 and 1.1. of `explore.pl`
4. branch `ada-main-program` is created, have required commits, and merged later into `master`
5. branch `bob-function-explore` is created, have required commits, and merged later into `master`

What is GitLab?

Most software development companies use some system for source version control, which enables efficient storage and retrieval of different versions of the software being developed and a way for different developers to collaboratively work on the software. The collaborative work means that they can write and test software independently at the same time, and there is a way to merge their changes easily in the final project. Subversion (SVN for short) and Git are examples of some popular version control systems. GitLab is a web-based system which is based on the Git source version control system and provides some additional functionality in a web-app style, similarly to another popular on-line platform — GitHub. Both GitHub and GitLab provide a Web interface to access and manage your Git repository.

GitLab is a Web-based DevOps platform, delivered as a single application, that provides a Git-repository manager providing wiki, issue-tracking and CI/CD pipeline features, using an open-source license, and developed by *GitLab Inc*. Continuous Integration (CI) is an established process to continuously provide integration of written code provided by a team in a shared repository. Developers share the new code in a Merge (Pull) Request. The request triggers a pipeline to build, test, and validate the new code prior to merging the changes within the repository. Continuous Delivery (CD) ensures the delivery of CI validated code to the app by means of a structured deployment pipeline. In general, CI helps the developers to catch and reduce bugs early in the development cycle, and CD moves verified code to the applications faster.

Some References

In this tutorial you should learn basic elements of Git and GitLab. For more information and tutorials on GitLab, you can refer to the official documentation:

`https://docs.gitlab.com/ee/README.html`

The following is a great interactive on-line tutorial for basic Git operations:

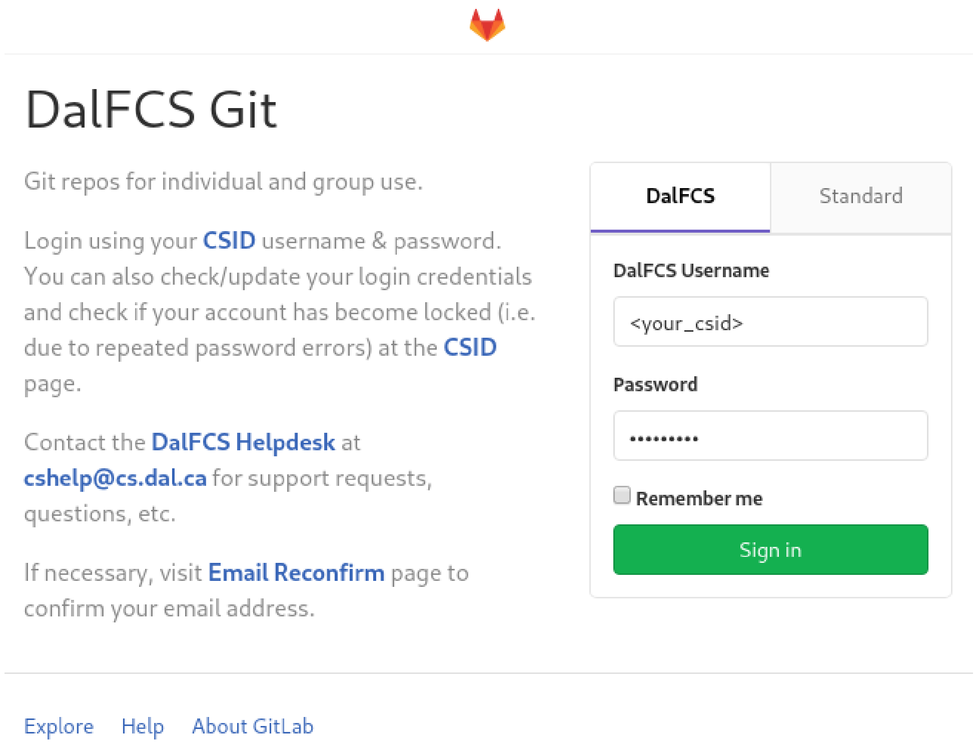
`https://learngitbranching.js.org`

We rely on material developed for several tutorials provided at Dalhousie:

– "Tools of the Trade" tutorial by Alex Brodsky for CSCI 2134,
– "Git Command Line Basics" tutorial by Sarah Meng Li and Robert Hawkey for CSCI 3130, and
– contributions by Dijana Kosmajac.

Step 1. Logging into DalFCS GitLab Website

Open your Web browser and go to the Dalhousie FCS GitLab web site: `https://git.cs.dal.ca` You should be able to see the Login screen, as shown in Figure 1. Login with your CSID login and password.



DalFCS Git

Git repos for individual and group use.

Login using your **CSID** username & password. You can also check/update your login credentials and check if your account has become locked (i.e. due to repeated password errors) at the **CSID** page.

Contact the **DalFCS Helpdesk** at **cshelp@cs.dal.ca** for support requests, questions, etc.

If necessary, visit **Email Reconfirm** page to confirm your email address.

Figure 1: Dal GitLab login screen.

A user of GitLab can participate in different *projects* and have different roles in them. All participants of a project are called *members*. The roles can be an *Owner*, *Maintainer*, *Developer*, *Reporter*, or *Guest*. We will refer to the

Dalhousie FCS GitLab installation as a repository of these projects, but we will also refer sometimes to your project as the repository. For example, since your course project is a part of the coursework, we may want to distinguish it from your GitLab project by calling your GitLab project your course GitLab repository. Hopefully, this will not be as confusing as it may sound, since you should be able to distinguish by context whether we are referring to the whole GitLab repository, or your own course repository.

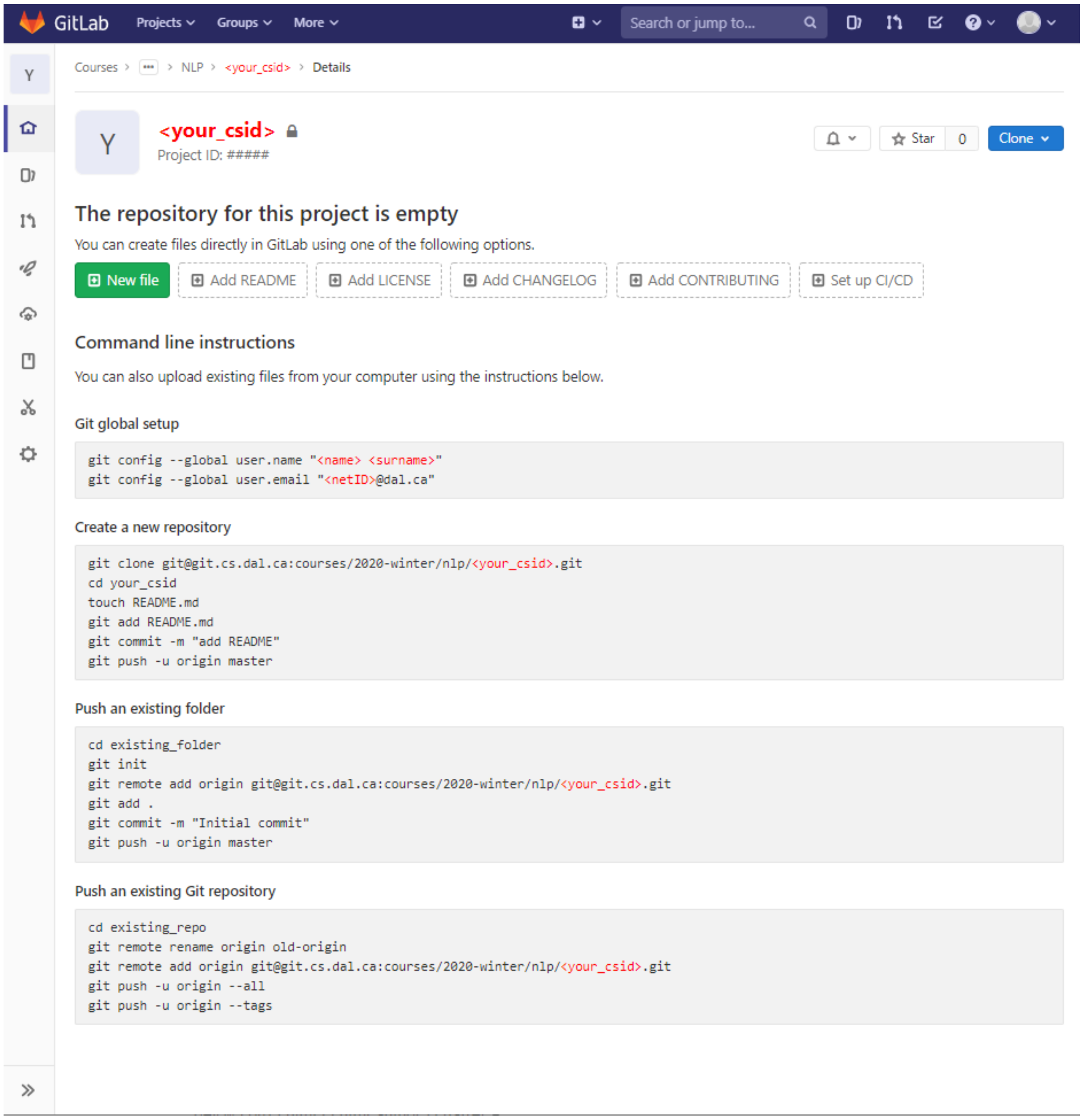
Since you can be a member of different projects in GitLab, you first need to find the project that is assigned to you within the NLP course. This project has the same name as your CSID and it is within the NLP project group, within this term (Winter 2020). In order to find it you can use the “Projects” or “Groups” menu option in the GitLab Web interface, or directly type in the URL of the project. Once you find it, the browser should show the following URL: `https://git.cs.dal.ca/courses/2020-winter/nlp/<your_csid>` where `<your_csid>` is your CSID. Figure 2 shows how the front page of this project should look like.

Step 2: Creating a README File in GitLab

Your course GitLab repository is empty at this moment, or at least it should be, and in this step we will see how to create and edit a file directly through the GitLab Web interface. It is not the common way to create and edit files in the repository but it can be handy. We will also learn some elements of Markdown—a lightweight markup language, which is conveniently formatted into nice HTML when viewing it.

A README file is very common for projects in general, since, as its name suggests, it is something that users unfamiliar with the project will first open to learn about the project. This is why there is a special button “Add README” in GitLab to create such file. *Click button “Add README”*. *Click* ⇒

A file named `README.md` will be opened for editing. Enter the following contents into it, almost all verbatim except the four lines at the beginning where you need to add your information:



The repository for this project is empty

You can create files directly in GitLab using one of the following options.

Command line instructions

You can also upload existing files from your computer using the instructions below.

Git global setup

```
git config --global user.name "<name> <surname>"
git config --global user.email "<netID>@dal.ca"
```

Create a new repository

```
git clone git@git.cs.dal.ca:courses/2020-winter/nlp/<your_csid>.git
cd your_csid
touch README.md
git add README.md
git commit -m "add README"
git push -u origin master
```

Push an existing folder

```
cd existing_folder
git init
git remote add origin git@git.cs.dal.ca:courses/2020-winter/nlp/<your_csid>.git
git add .
git commit -m "Initial commit"
git push -u origin master
```

Push an existing Git repository

```
cd existing_repo
git remote rename origin old-origin
git remote add origin git@git.cs.dal.ca:courses/2020-winter/nlp/<your_csid>.git
git push -u origin --all
git push -u origin --tags
```

Figure 2: Front page of your GitLab NLP repository.

README.md
```
# CSCI 4152/6509 Natural Language Processing (Winter 2020)
## GitLab Course Repository

### Student Details

Table example:

| Student Name:                | (enter your name) |
|------------------------------|-------------------|
| CSID:                        | (enter your CSID) |
| Banner Number (B00 Number): | ...               |
| Dal E-mail Address:          | ...               |

### Some Examples in MD (Markdown)

This is an example of an unordered list:
* The first item,
* The second item,
* and so on.

And this is how to create an ordered list:
1. The first item,
1. The second item (number 2 is generated even though
   we type 1 again)
1. The third item and so on.

If we want to include some code, we can use the "pre" tag:
<pre>
$ ls -l
$ pwd
</pre>
or
<pre>$ ls -l</pre>
<pre>$ pwd</pre>
```

You can leave the commit message as it is, or you can make it more descriptive. Do not change the Target Branch (it should be master), and click the button 'Commit changes' to save the changes into the repository; i.e., to 'commit' them. After that you should be able to see the contents of the `README.md` file nicely formatted in HTML. We will explain the `commit` operations and the `master` branch later.

Submit: The file `README.md` must be created in the GitLab repository as described. If you finished the given instructions it is submitted already.

Note: Even though we have shown how to edit a file directly using the GitLab Web interface, this should rarely if ever be done. It is very convenient here to make the first edit of the README file because we can immediately see how it looks like on the web, however, editing other files should be avoided. As we will see later, the proper way to work on the files is to create a branch and test the changes before making them generally available to the rest of the team.

Now we will switch into working on a terminal by logging in into the bluenose server. You should keep your browser open with the GitLab page so you can later check the new contents of your GitLab repository as you keep updating it.

Step 3: Logging in to server bluenose

As in the previous labs, login to your account on the server bluenose.

Change your directory to `csci4152` or `csci6509`, whichever is your registered course. This directory should have been already created in your previous lab.

Create the directory `lab4` and change your current directory to be that directory.

Check that you are in the right directly by running the command:

```
pwd
```

The output should show a path that looks like this: *your_home_directory*`/csci4152/lab4` or *your_home_directory*`/csci6509/lab4`.

This is the directory where you should keep files from this lab.

Step 4: Using HTTPS Address in Git

Step 4-a: Find GitLab repository address

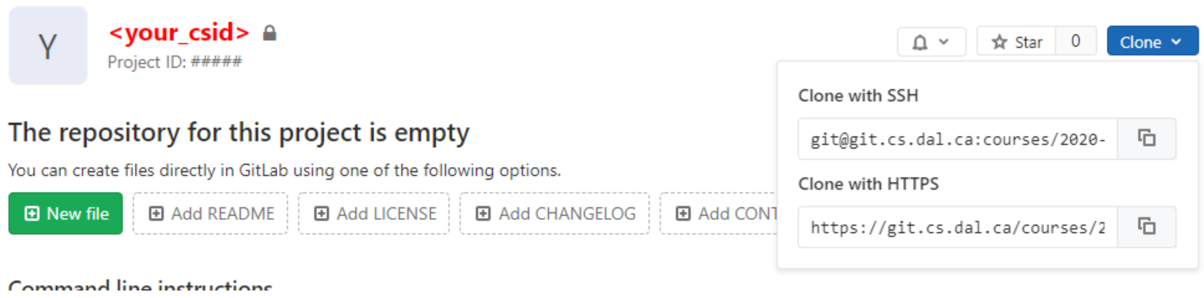
Our first step is to get a copy of your GitLab repository in our local directory. This operation in Git is called `clone` because we are copying (i.e., cloning) not only the latest versions of the files, but also history of their revisions in our local directory. The clone command requires the address of the remote repository and there are two variations of address that we can use: we can clone using SSH from an address that looks like this

```
git@git.cs.dal.ca:courses/2020-winter/nlp/<your_csid>.git
```

or using HTTPS from an address that looks like this

```
https://git.cs.dal.ca/courses/2020-winter/nlp/<your_csid>.git
```

where `<your_csid>` needs to be replaced with *your* CSID. You can also find and copy any of these addresses from your GitLab repository after clicking at the top blue button labeled as "Clone," as shown in Figure 3.



The repository for this project is empty

You can create files directly in GitLab using one of the following options.

Clone with SSH

git@git.cs.dal.ca:courses/2020-

Clone with HTTPS

https://git.cs.dal.ca/courses/2

Figure 3: Finding the Address of Your Repository

Both of these addresses, SSH and HTTPS, can be used to clone the repository, and later work with it. If we use the HTTPS option, then you will need to enter CSID and password to clone the repository and also repeat it later for any `git` command that communicates with the GitLab repository. A somewhat more convenient way is to use the SSH address, because we can set up our SSH key in such way that we do not have to use the password each time, and relay on private-public key authentication. You have an option to work with the repository in any way, but here we are going to show how to set up password-less access using the SSH address.

Step 4-b: Clone Repository via HTTPS

Make sure that you are in the right directory by running the command

```
pwd
```

The output should be:

```
~/csci6509/lab4
```

or

```
~/csci4152/lab4
```

where ~ is your home path directory. Now, you need to clone your GitLab repository using the command:

```
git clone https://git.cs.dal.ca/courses/2020-winter/nlp/<your_csid>
```

where `<your_csid>` is your CSID, as before. You will be prompted to enter your CSID and password, and after that you should be informed that the repository is successfully cloned. You can verify that by using the command `ls` to see that a directory named `<your_csid>` is created in your current directory. You should then run the command:

```
cd <your_csid>
```

to enter the copy of your repository, and using the command `ls` you can see that it contains the file `README.md`, which you created using the GitLab web interface.

Step 5: Prepare and Submit Public Key

Step 5-a: Create Directory and Check Keys
We will create a directory named `lab4g` as a part of the GitLab repository where we will store some more files that we will create in this lab. We name it differently than `lab4` in order to distinguish it from the directory `lab4`, which was previously created and which is not part of the GitLab repository. To create the directory, run the command:

```
mkdir lab4g
```

and make that directory your current directory:

```
cd lab4g
```

Now we need to prepare the public and private ssh key. It is possible that you created those keys before, so first check whether they are there using the command:

```
ls ~/.ssh/
```

If the command lists the files `id_rsa` and `id_rsa.pub` then the keys were created before; if not, then we need to generate them. Before we generate the keys, it is important that you understand the following security note.

Security note about attaching a password to a private key: From the security perspective, it is generally recommended to always have a good password attached to a private key; however, this is not required and in many situations it is more convenient not to have a password. We will assume in further instructions that you do not attach a password to the private key. If you prefer to create a password, you can do it—the only difference is that you will need from time to time enter your password, which we will not assume in the lab notes, and if you want to make it a bit more convenient you will need to use an ssh agent. In any case, you should know what are trade-offs in risks and convenience in using password or not. If you attach a password to your private-public key pair, it means that whenever you use your private key you need to enter the password. In this way, if someone gets a copy of your private key, they still can not use it until they get the password, so it is more secure. On the other hand, we use the private-public ssh key here in order to have password-less communication with GitLab, so having a password in a way defeats the purpose of using a public key for access. We assume that you will keep your private key secret, and that guarantees that no one else can access your GitLab repository identifying as you. You should remember that you must not share your private key and make sure others do not have access to it (e.g., via file permissions, backups, etc.).

If you are interested how to have a password attached to a key and not to have to type it often, you can look into using an ssh agent. It is used by running a command at the beginning of your login session to bluenose where you need to type your key password, and during that session you do not need to retype it.

Step 5-b: Generate Keys If Needed
Now, going back to key generation, if we need to generate the keys, we will use the following command:

```
ssh-keygen -t rsa
```

You should respond by pressing the Enter key to all questions: first, keep the file names as default offered names, and on prompt for password, just press an Enter in order not to use the password. If you prefer, you can use the password, but then you will need either to type it for many git operations, or you will need to use an ssh agent, as we stated in the note above.

The `ssh-keygen` command generates two files: `id_rsa` contains your private key, and you should keep it confidential and not share with anyone; and `id_rsa.pub` contains your public key, which you can share, and which we will use to provide password-less access to GitLab when accessing it from your bluenose account. You can check again that the key files are indeed generated by typing:

```
ls ~/.ssh/
```

Now you should copy the public key file `~/.ssh/id_rsa.pub` into your current directory, which should be `lab4g`. You can do it using the command:

```
cp ~/.ssh/id_rsa.pub .
```

Step 5-c: Adding Files in Git
We are now going first to commit the file to our local Git repository. In Git, we first need to add any files that we want to commit to the so-called *staging* area by using the command '`git add`' and then we use '`git commit`' to *commit*; i.e., to save files into our local Git repository. The commit command requires a log message, and in order to prevent an editor to be automatically opened to enter this message, we will provide it in the command line. To do all this, use the following two commands:

```
git add id_rsa.pub
git commit -m'Commit id_rsa.pub'
```

Our public key file is submitted to our local git repository, and now we need to 'push' it to the remote GitLab repository. This git operation is called *push* and we run using a '`git push`' command. The remote GitLab repository address is saved and known as 'origin', and our current branch is 'master', so we are going to use these in the command without more explanation on them at the moment. Run the following command and be ready to enter your CSID and password:

```
git push -u origin master
```

Enter your CSID and password at the following prompts:

```
Username for 'https://git.cs.dal.ca': <your_csid>
Password for 'https://<your_csid>@git.cs.dal.ca': <your_password>
```

Now, go to your Web browser window, which should have the GitLab site still open and logged in, and you should be able to find your public key file there. First, if you reload the page you should now see the directory `lab4g` and when you open it there should be a file named `id_rsa.pub`, which you should also open. The file should start with `ssh-rsa` and then followed by a long seemingly random string of letters and digits, which encode information of your public ssh key. With this, you successfully fished the Step 5.

Submit: The directory `lab4g` and the file `id_rsa.pub` should be submitted in your GitLab repository by this time.

Step 6: Setting up SSH Key in GitLab

We continue now from the ending of the previous step, where in your Web browser you have GitLab repository open, and in particular, you opened your public key file `id_rsa.pub`. In the top corner, you should be able to notice a double-square icon, one square over another, and if you hover over it with your mouse, it should show the tooltip "Copy file contents". Click it, and in this way you will have the contents of the file `id_rsa.pub` saved in your computer clipboard buffer (same as if you pressed Ctrl-C for 'copy').

Click on the icon in the top right corner of the GitLab page, which represents your avatar. If you already set up your avatar (e.g., photo of you), it should be shown here, or by default it will show a person's profile shadow. After clicking on it, you should see a pull-down menu and you should click on "Settings" on it. Now, on the left menu, you should see the "SSH Keys" option, and you should click on it. You will se a textbox under the "Key" label, with an explanation above that starts with "Paste you public SSH key…"'. Click on the box and press Ctrl+V combination to paste the key. You should remove any possible empty lines around the key, so that the key remains by itself in one line. When you remove these empty lines, the button below with the label "Add key" should become green. You can modify the Title filed if you want, and then you can click the green button "Add key". With this you have set up your SSH key in GitLab and should be able now to work with the repository from your bluenose account without a need to enter your CSID and password frequently.

Step 7: Clone with SSH

Go back to the bluenose terminal, in other words click on your terminal window, where you are logged in in the bluenose server. You should check your working directory using the command `pwd`. You are probably in the directory with path ending with `/lab4/<your_csid>/lab4g`. We need to be now in the directory `lab4`, and you can achieve this with the command: `cd ../..`

Rename your previously cloned directory named `<your_csid>` to `<your_csid>-https` using the command:

```
mv <your_csid> <your_csid>-https
```

In this way, it will not conflict with a new cloned copy with the same name, which we will make using the SSH address. Clone the repository using the SSH command as follows:

```
git clone git@git.cs.dal.ca:courses/2020-winter/nlp/<your_csid>.git
```

Assuming that you have set up your SSH key correctly, this command should run without prompting you for a password, and after executing it you should see the directory named `<your_csid>` in your current directory. If you run the command `ls` you should actually see two directories `<your_csid>` and `<your_csid>-https`. You can now delete your previously cloned directory `<your_csid>-https` by using the command:

```
rm -rfv <your_csid>-https
```

Be careful to run exactly the above command! You must be always **very careful** with the command `rm` in order not to remove a directory that you do not want to remove.

Step 8: Preparing Files `explore.pl` and Shakespeare

One useful function of Git, as well as other source version control systems, is that it keeps history over various versions of source files, so in case that we want to go back and retrieve older version of a file or set of files we can easily do that. We do need to make a decision when to take these snapshots of the files and commit them, because only the committed versions can be retrieved later. In this step we are going to explore this functionality.

Make sure that you are in your main directory of your cloned project; i.e., in the directory path that ends with `lab4/<your_csid>`. Type in the command:

```
ls -la
```

and you should see an output like this:

```
drwx------. 8 <your_csid> <your_csid> 14 Feb  2 08:58 .git
drwx------. 2 <your_csid> <your_csid>  3 Feb  1 22:36 lab4g
-rw-------. 1 <your_csid> <your_csid>  5 Feb  2 08:58 README.md
```

You should remember that we created the file `README.md` and the directory `lab4g`. The hidden directory `.git` is the directory where our local repository is stored and you should not edit it directly. Git also keeps there information about our remote repository and the address of the GitLab, where our remote repository is kept. Git is a distributed source version control system, which means that our local repository keeps the files history, but it is also kept at the remote repository. We do not generally have to have a remote repository and we can have more than one. We can save and retrieve files to and from each of them separately. For comparison, Subversion (or SVN) is a centralized system where we work only with a working copy of the files and the history is kept only at one central repository, which can be either remote or somewhere local.

Change your directory to `lab4g`. Let us assume that you want to start working on a Perl program to be used to explore a textual file by providing an interface to print all lines containing a given word. We open our emacs editor (or any other local bluenose editor that you like) and prepare the following file named `explore.pl`. If you prefer not to type, the file is available on bluenose and you can copy it with the command:

```
  cp ~prof6509/public/lab4-1-explore.pl explore.pl
```

The file content is as follows:

```
explore.pl
```

```
#!/usr/bin/perl

my $fname = shift;
open(F, $fname) or die;
my @lines = <F>; close F;
print "File $fname loaded.\n";

while (1) {
  print "Enter a search word (0 to exit): ";
  my $w = <>; chomp $w;
  last if $w =~ /^\s*0\s*$/;
  &explore($w);
}

sub explore {
  my $w = shift;
  print "TODO: exploring $w\n";
}
```

Save the file (remember that its name is `explore.pl`) and make it user-executable (in other words use the command `chmod u+x explore.pl`). Copy also a file for testing with the command:

```
cp ~prof6509/public/Shakespeare__Hamlet-25k.txt .
```

which is some first 25 kilobytes of the famous Shakespeare play “Hamlet”. Run the command:

```
./explore.pl Shakespeare__Hamlet-25k.txt
```

to test your program. Remember that you can use Tab key after typing `./expl` to complete the command name, and again use Tab key after typing `Shake` to complete file name, so that you do not have to type it completely. This feature is called auto-completion in the bash shell and if you are not familiar with it you may read about it on Internet. After running the above command you can try some words, like ‘hamlet’, but the program is not finished and will not do much. You should also test entering ‘0’ in order to finish the program.

Step 9: Commit Files explore.pl and Shakespeare

Let us say that you decided that you did enough work today and you would like to save what you have in your local, but also your remote repository. Other than having an additional backup copy, another benefit of saving it in a remote repository is that you can later clone the project at your home computer, or a laptop and continue working there. This is easier that copying around the files directly from bluenose. In order to do this, we first need to *add* the files to Git, as we did with `id_rsa.pub`, if you remember. We decide that we want to save in Git the files `explore.pl` and `Shakespeare__Hamlet-25k.txt` so we run the following command:

```
git add explore.pl Shakespeare__Hamlet-25k.txt
```

If you want to check which files are *staged*; i.e., ‘added’ to be committed in the next commit, you can issue the command:

```
git status
```

The output may vary depending on what exactly you were doing, but it should look something like this:

```
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   Shakespeare__Hamlet-25k.txt
        new file:   explore.pl

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  explore.pl~
```

You may or may not have the file `explore.pl~`. This file is a backup file created by emacs, so we will not worry about it and we will not add it to the Git for saving. Commit these files locally by running the command:

```
git commit -m'explore.pl version 1.0'
```

If you did not set your name and email, the commit command may respond with the following output:

```
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author
```

If you want to set your name and email for git to use, you can use the above commands to do it.

Let us suppose that after this we realized that we could have written a comment line to describe the file name and the version, and we do that by inserting in the file `explore.pl` a line like the red line below:

```
#!/usr/bin/perl
# explore.pl, version 1.1

my $fname = shift;
...
```

Now, we save the file, add it to git and commit it using the commands:

```
git add explore.pl
git commit -m'explore.pl version 1.1'
```

Step 10: Explore Previous Commits

We see now how we can save several historical versions of the file that we are working on. To see this history, run the command:

```
git log
```

and we can see the history of our commits. It is run through the program `less` so we can browse through it by pressing Space bar to page down, or key 'b' (for back) to go page up. We exit the viewing by pressing the key 'q' (quit). In order to go back in time, we can *checkout* any of the committed versions. A very direct way is to checkout a previous version is notice the lines in the 'git log' output that start with the word `commit` and followed by a long sequence of hexadecimal digits. These are SHA-1 checksums of the commits and they uniquely identify the made commits. You can use them to bring back any earlier commit. The command to bring back an earlier commit is 'git checkout' and instead of giving the full SHA-1 checksum, you can use only a few hexadecimal digits at the start. For example, if you see the following line in the 'git log' output:

```
commit 67bbc0e4ce0c969a1e888259e8001a519563f620
```

with the commit that you want to bring back, then you can issue the command

```
git checkout 67bb
```

using only the first four digits, and it will probably work. If it does not work than you can use more digits, or all of them. If the command is successful you will see an appropriate output with a line at the end that looks like:

```
HEAD is now at 67bbc0e explore.pl version 1.0
```

which means that we have now the previous version of the program `explore.pl`. You can browse it using the command '`more explore.pl`' to see indeed that this is an older version of the program. In case that we need, we can save somewhere this older version, but generally we should not be working on this older checkout, but instead bring back the newest version with the command:

```
git checkout master
```

You can take a look and check that your file `explore.pl` is the newest, version 1.1, contents.

Step 11: Push Changes to GitLab

All these changes and versions of the file `explore.pl` are saved in your local Git repository and they are not yet visible in your remote GitLab repository. If you want, you can check in your Web browser your GitLab repository and you will see that it does not contain the `explore.pl` file nor the start of the Hamlet that we saved. In order to save this in the remote repository, we need to push the changes using the command:

```
git push -u origin master
```

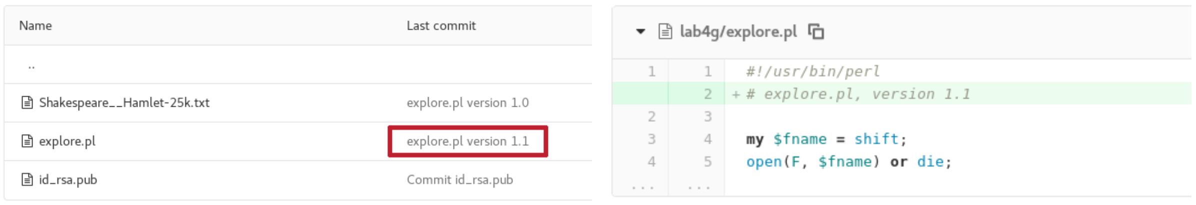
Run this command in your bluenose terminal. You should get the output that looks something like this:

```
Counting objects: 13, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (13/13), 12.42 KiB  3.11 MiB/s, done.
Total 13 (delta 2), reused 0 (delta 0)
To git.cs.dal.ca:courses/2020-winter/nlp/<your_csid>.git
   eb98362..2347063  master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Some numbers will be different.

Now, go to your browser window and reload the view of your repository. If you open the directory `lab4g` you will see the new files as shown in Figure 4. If you click the link marked with the red rectangle in the left part of Figure 4, then you can see the file `explore.pl` with the latest changes labeled as shown in the right part of the same figure.

Submit: The directory `lab4g` in GitLab repository should now also contain the files `explore.pl` and `Shakespeare__Hamlet-25k.txt` and there should be at least two versions of the file `explore.pl` (ver-



Name	Last commit
..	
Shakespeare__Hamlet-25k.txt	explore.pl version 1.0
explore.pl	explore.pl version 1.1
id_rsa.pub	Commit id_rsa.pub

lab4g/explore.pl

```
 #!/usr/bin/perl
+# explore.pl, version 1.1

 my $fname = shift;
 open(F, $fname) or die;
...
```

Figure 4: GitLab view of the files and latest changes

sions 1.0 and 1.1) saved in separate commits as described in the lab.

Step 12: Creating Branches: Directories Preparation

Another important functionality of the source version control is collaboration between multiple people on the same project. Until now, we worked on a single line of changes, which is by default called the `master` branch. Since it was a single branch, there was no particular reason to call it a branch actually. If several people work on their copies of the files and frequently commit their changes to the master branch, we can see that we could easily end in a chaotic situation with conflicting changes made on various versions of the files. In order to address this, an important rule in working with Git collaboratively is that each developer should start with a copy of the master branch, and then work independently on their sequence of commits called a *branch*. Only once they finish implementing the feature they are working on, and they are sure that it passes all tests and seem to work well, they should merge the branch back into the master branch. Git is quite good in merging changes, but in some cases where a conflict is generated, it allows us to jump in, fix the conflict, and still continue with the merge.

We will go through an exercise of simulating this working environment using your GitLab project. Let us say that one person named Ada starts working on a Perl program. It is the very program `explore.pl` that we already used in this lab. In order to remember that the person's name is Ada, we are going even to change name of the directory that contains our cloned repository. Make sure that you are in the directory `lab4` and change the name of your cloned directory `<your_csid>` with the command:

```
mv <your_csid> ada
```

Ada did the work that we did so far, which means that she wrote the main framework of the program and some functionality, but the function `explore` still needs to be written. At this point, Bob joins Ada's team and offers that he writes the function `explore` and maybe starts some work on the project documentation. Ada adds Bob to the GitLab team, and Bob now creates his own clone of the project. To simulate this, you should be in the `lab4` directory and run the following commands:

```
git clone git@git.cs.dal.ca:courses/2020-winter/nlp/<your_csid>.git
mv <your_csid> bob
```

You can check with the command `ls` that the directory `lab4` now contains two subdirectories: `ada` and `bob`.

Step 13: Creating Ada's Branch

Ada would like to make some more changes to the program, but she remembers that Bob is now working on the project too, so she needs to create her own branch to work on it. The branches have names and in some projects developers follow a convention that they start the branch name with their userid, so Ada decides to call her branch `ada-main-program` since she will be working on improving the main program.

Note: The convention of starting a branch name with the developer's userid is used in some projects, but this is not a universal convention. There could be more than one developer working on the branch so this would not be convenient. More frequently, companies have convention to label the branch with a bug report id, and similar. It is also important to remember that branch names may or may not be case sensitive depending on a system where developer is working. We will follow a convention here that the branch names are lowercase with words separated by a minus character.

In order to simulate Ada's operations, you do the following: First, go to the Ada's cloned repository:

```
cd ada
```

Check that you have the latest update of the `master` branch:

```
git checkout master
git pull
```

Create and checkout a new branch named `ada-main-program`. This can be done with two commands: `git branch` and `git checkout` but we are going to do it with one command as follows:

```
git checkout -b ada-main-program
```

Verify that the branch is created and it is your current branch by running the command:

```
git branch
```

You will see an asterisk * next to your current branch.

Ada can now work on `explore.pl` and other files if needed, without worry that the changes will collide with changes that Bob is making, or that Bob will get some incomplete copy of `explore.pl` that is in the middle of changes. Change your current directory so that you are in the directory `lab4g` in Ada's cloned copy. You will probably need to run the command:

```
cd lab4g
```

Ada decides to improve `explore.pl` but handling the case when the user forgets to give a file name. For example, of we type:

```
./explore.pl
```

the program will simply die without much explanation. Ada addresses this by adding the following code at the beginning of the file, and updating the file version number as follows:

```
#!/usr/bin/perl
# explore.pl, version 1.2

if ($#ARGV == -1) {
  die "Usage:  $0 filename\n";
}

my $fname = shift;
```

...*(the rest of the file)*
This modified file is also available as `~prof6509/public/lab4-ada-explore.pl` on bluenose if you want to make sure you modified it correctly. If we run

```
./explore.pl
```

we should get a more informative message about how to use our program. You can notice in the code the use of a special Perl variable `$0` which is assigned the program name in the way it was called. Ada is quite happy with this change and for now decides to save it locally and also to push it to the remote GitLab repository. As we saw before, she first need to add the file and then commit it with some useful log message as follows:

```
git add explore.pl
git commit -m'Ada: added usage message'
```

In order to push this branch to the remote GitLab repository, known with a short name as `origin` to git, we need to run the following command:

```
git push --set-upstream origin ada-main-program
```

The option `--set-upstream` needs to be done only once per new branch, we do not have to specify anything after `git push` later. Actually the option `--set-upstream` is not necessary in this case, but it makes some later commands shorter.

Step 14: Creating Bob's Branch

Let us now see how Bob is doing. Go to his directory `lab4g.` which you can do with the command:

```
cd ../../bob/lab4g
```

Have in mind that this command depends on your current directory, so do not follow it blindly, but frequently check in which directory you are by using the commands `pwd` and `ls`.

Bob would like to start working on the function `explore` so knowing well how to work with Git, he also first makes sure he has the newest version of the `master` branch with the following commands, which you will do now:

```
git checkout master
git pull
```

We can see that Bob will notice that Ada created her branch, but his work is not yet affected by it. Bob continues and creates his own branch and checks it out using the command:

```
git checkout -b bob-function-explore
```

and now he can work on the file `explore.pl`. Bob decides to make two main changes to the file. First, he updates the version number. He realizes that this may later conflict with what Ada is doing, so he adds a short `bob` note by it. After that he implements properly the function `explore` by replacing the previous version with the changes below:

```
#!/usr/bin/perl
# explore.pl, version 1.2 bob
: :  (the previous code is here)
# Bob wrote this function
sub explore {
  my $w = shift;
  print "Lines containing $w:\n";
  for (@lines) {
    print if /\b$w\b/i;
  }
}
```

You can also find the version of the `explore.pl` file the way it should look at this point on bluenose in `~prof6509/public/lab4-bob-explore.pl`. Before saving the file, Bob should test the file, and so should you. Run the program with:

```
explore.pl Shakespeare__Hamlet-25k.txt
```

and try entering different words, for example `hamlet`, `the`, `Denmark`, and finish with `0`.

Bob also decides to start working on documentation for the project, so he edits the file `report.txt` and adds the following two lines: (Again, you are simulating Bob, so go on and create and save `report.txt` with the following two lines.)

```
report.txt
```
```
The Manual for explore.pl
-------------------------
```

Save the file. At this point, Bob is happy of his contribution so far, and decides to save his branch and push it to the remote GitLab repository, so he issues the following commands:

```
git add explore.pl report.txt
git commit -m'Bob: added explore function, some documentation'
git push --set-upstream origin bob-function-explore
```

If we forget at any point what was the exact name of our branch (as Bob just did), we can always check it with '`git branch`'.

If we go to the GitLab browser window and refresh it, we can see that all three branches, the master one, the Ada's, and the Bob's one are saved and exist in the repository.

Step 15: Ada Merges Her Branch

We will go now and see what Ada is doing and simulate her actions. Change your directory to the Ada's `lab4g` and do a `git pull` to have the latest changes:

```
cd ../../ada/lab4g
git pull
```

After this command Ada can notice that there a Bob's branch at the remote repository, although this branch is not pulled to her repository locally.

Ada decides that it is time to make her changes part of the `master` copy, and this is done by *merging* her branch with the `master` branch. In software development in general, it is important that the merge with the `master` branch is done only once we are sure that our changes work well; i.e., they pass testing and they do not break some other parts of the code. In order to do the merge, Ada checks out the `master branch` and then merges her `ada-main-program` branch with the following commands:

```
git checkout master
git merge ada-main-program
```

This merge operations should go smoothly and you should get the output like this approximately:

```
Updating 2347063..0616dbf
Fast-forward
 lab4g/explore.pl | 6 +++++-
 1 file changed, 5 insertions(+), 1 deletion(-)
```

You can check the `explore.pl` file and you can see that Ada's changes are now updated in it, even though we are in the `master` branch. In order to save this merge in the remote repository as well, Ada finally runs:

```
git push -u origin master
```

Step 16: Bob Merges His Branch

Now, we are going to simulate Bob's actions: First go to his `lab4g` directory:

```
cd ../../bob/lab4g
```

Bob decides as well that he should merge his changes into the `master` branch. He first makes sure that his `master` branch is up to date with respect to the remote repository, so he needs to checkout this branch and run a pull command:

```
git checkout master
git pull
```

Bob notices that the `master` branch was changed. Normally, he should not worry about those changes but make sure that he finishes his changes, and then merge his branch back into the `master` branch and see that it all works well. However, if we work on a branch for a very long time, it may be useful to see if the lastest changes to the `master` branch have an affect on our work. Because of this, we may want to apply the changes in the `master` branch to our own branch. This operation is called `rebase`, and we will show it here by looking into how Bob would would do it. First, Bob needs to checkout his `bob-function-explore` branch and then apply `rebase` as follows:

```
git checkout bob-function-explore
git rebase master
```

This operation will not go without issues since, as we may remember, Bob and Ada both made edits to the 'version' line of the program `explore.pl`. If nothing else, there is no way that Git can tell which of these lines should be kept in the program. Because of this, the output of the last operation will look something like this:

```
First, rewinding head to replay your work on top of it...
Applying: Bob: added explore function, some documentation
Using index info to reconstruct a base tree...
M     lab4g/explore.pl
Falling back to patching base and 3-way merge...
Auto-merging lab4g/explore.pl
CONFLICT (content): Merge conflict in lab4g/explore.pl
error: Failed to merge in the changes.
Patch failed at 0001 Bob: added explore function, some documentation
The copy of the patch that failed is found in: .git/rebase-apply/patch
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git
rebase --abort".
```

We can see in the output that Git was not able to merge different versions of the file `explore.pl` and suggests that we resolve this conflict manually, also giving some instructions about how to proceed after that.

Since the file `explore.pl` is cause of the conflict, Bob needs to edit this file. In a way, we are lucky it is only this file, otherwise, we would need to edit several files in the 7conflict. Open the `explore.pl` file and we should see the contents like this (without colors). We will mark with red color the lines that Bob (meaning your) should delete, with blue a minor change to be made, and save the file:

explore.pl

```
#!/usr/bin/perl
~~<<<<<<< HEAD~~
# explore.pl, version ~~1.2~~ 1.3

if ($#ARGV == -1) {
  die "Usage: $0 filename\n";
}
~~=======~~
~~# explore.pl, version 1.2 bob~~
~~>>>>>>> Bob:  added explore function, some documentation~~

my $fname = shift;
open(F, $fname) or die;
my @lines = <F>; close F;
print "File $fname loaded.\n";

while (1) {
  print "Enter a search word (0 to exit): ";
  my $w = <>; chomp $w;
  last if $w =~ /^\s*0\s*$/;
  &explore($w);
}

# Bob wrote this function
sub explore {
  my $w = shift;
  print "Lines containing $w:\n";
  for (@lines) {
      print if /\b$w\b/i;
  }
}
```

If you take a look at the file, you see that Git left a labeled part of conflict showing the part coming from the 'HEAD' which is the latest commit in the master branch, and the part coming from the Bob's branch. It is up to Bob now to decide what to do about it, and he accepts the new code implemented by Ada with the usage message, and removes the old code coming from his branch. He also consolidates the version number by setting it to 1.3. You can also see how this file should look like after editing in bluenose at ~prof6509/public/lab4-bob-resolve-explore.pl.

Bob now needs to commit this change and invoke continuation of the rebase command. Type the following commands:

```
git add explore.pl
git commit -m'Bob: conflict resolved'
git rebase --continue
```

The rebase command should now finish successfully. After this, Bob makes the merge by first checking out the master branch and issuing the merge command:

```
git checkout master
git merge bob-function-explore
```

We can verify that we have the newest version of the explore.pl file and the file report.txt should also be there.

Finally, Bob pushes these changes to the remote GitLab repository with:

```
git push -u origin
```

We can go now to Ada's `lab4g` directory and make sure that she has the latest update as well:

```
cd ../../ada/lab4g
git pull
```

We are finished with this lab. At this point in practice Ada and Bob would probably delete their branches, which are sometimes called feature branches as there were used to implement certain features. You **must** not delete these branches because they need to be checked by the marker to verify that you finished the lab completely. Just for your information, a merged branch can be deleted with the command `git branch -d branchName`. A merged branch can also be deleted on the GitLab web site.

Actually, the merge operation can also be done on the GitLab web site. For simplicity reasons you were instructed to make branch merge in command line, but in practice there could be more rules around it. For example, it is usually agreed in a project that when a developer wants her or his branch to be merged to the `master` branch, they need to create a "merge request" on GitLab, also called "pull request" (PR) on GitHub. In this request, he or she will also invite some other developers to review the changes. If everything looks agreeable, someone else other than the branch developer should make the merge. Once the merge is finished, everyone in the project should pull the latest `master` branch, and they should `rebase` their development branches to be sure they are synchronized with the latest changes.

Submit: The three branches should be up to date in GitLab by this point with appropriate merges. GitLab will probably show that Ada's branch is merged and not Bob's branch, which is okay, because GitLab does not identify merged branch right away but only after additional commits are made.

This is the end of Lab 4.