Lab 2 – Git Introduction

Git is a distributed version control system that helps developers to collaboratively author files (often source code, but also documentation, learning material, websites, this worksheet, etc).

There are many GUIs (graphical user interfaces) for Git, e.g., Github Desktop, SourceTree and GitKraken with very similar functionalities. However, the core Git tool can be run via the command line without the need of a graphical front-end. The main command is simply `git`, followed by whatever options you may want. In this tutorial we will go over some of the key Git usage scenarios.

General Instructions

Along this lab, you will be provided with a series of `git` commands and a description of what the command does. As you go along, you are expected to run the command by yourself.

This lab sheet should be self-contained, but you may use the Git Manual as an additional reference. Should you run into trouble, ask the TA for help.

In this lab, imagine you are part of a software development team that is assessing the software quality of an early version of Microsoft Word. The open-sourced code for MS Word 1.1 (from 1989) happens to be available on Github:

https://github.com/BlastarIndia/msword

Steps

This lab requires the use of the command line. If you are not familiar with the command line, it is recommended that you go through some introductory material to the use of a terminal:

- Windows terminal
- MacOS command line primer
- Linux

1. Setup

Open a terminal and create a directory within which you will complete this lab, e.g., within your user home directory:

```
mkdir ~/co7095-lab2/
```

and change directory into the one you have just created:

```
cd ~/co7095-lab2
```

2. Cloning the MS Word 1.1 repository

Unlike other version control systems such as SVN, with Git, the whole repository is stored on your local computer. This includes every previous change made to the system throughout its development history. The action of pulling a repository from its original location into a local directory is called *cloning*. You can clone a repository by typing:

```
git clone <URL>
```

So in our case, type:

```
git clone https://github.com/BlastarIndia/msword.git
```

The output of this command should look like this:

```
Cloning into 'msword'...
remote: Enumerating objects: 933, done.
remote: Total 933 (delta 0), reused 0 (delta 0), pack-reused 933
Receiving objects: 100% (933/933), 4.42 MiB | 7.16 MiB/s, done.
Resolving deltas: 100% (64/64), done.
Updating files: 100% (910/910), done.
```

Navigate into the newly created directory:

```
cd msword
```

3. Inspecting the code

This module isn't about programming, so we're not going to analyse the source code in detail. It is however interesting to skim through some of the code files and look at what the Microsoft developers wrote in the comments!

For example, we can search the files for particular words such as "why" using the `grep` (Linux and MacOS) or `findstr` (Windows) commands:
`

```
grep -r "why" .
```

```
findstr /s why *.*
```

Both commands do exactly the same. The dot at the end of the `grep` command tells grep to search in the current directory; if you omit the dot, the command may hang and you'll have to abort it with Crtl/Cmd+C.

By having a look at the output of this command, you might notice some gem comments such as `i'm not sure why this is being done, but I think it will be ok`. Try searching for other words apart from "why" that you might be curious to know if they exist in the codebase, e.g., "hack".

4. Change a file

Notice that by cloning the repository in step 2, you have downloaded a full copy of the entire codebase to your local machine. Any changes you make will remain local, unless you submit a "pull request" to the main repository; because the project is now discontinued, there may not be anybody there to accept your pull request.

Pick a file, and open it in your favourite text editor, e.g., `gedit <filename>`, `vim <filename>`, or Visual Studio Code). Make a change (e.g. remove a comment), and save the file.

5. Check the repository status

Git keeps track of which files have been changed. When working on collaborative, shared repositories, you are encouraged to do this regularly. You can check this with following command:

```
git status
```

The output should mention the file that has been modified in the previous step.

6. Commit your change to your local repository

Consolidating your change into the repository is done in two steps. First, you need to tell git which files you want to commit. This can be done with the `git add` command:

```
git add .
```

or

```
git add <filenames>
```

This command will simply add all of the files that have been changed or the list of files `<filenames>` to the list of files (one or several) to be committed.

Once you have done this, you need to actually commit these files to the repository. This is done via the `git commit` option.

```
git commit -m “<message>”
```

where `<message>` is a string of characters that summarises your change. For example:

```
git commit -m “removed obsolete comment”
```

or

```
git commit -m “refactored loop”
```

or

```
git commit -m “renamed method”
```

7. Dealing with multiple repositories

Let’s assume that another member of your team has also started working on the project. In reality, she would be working on a local copy on her own computer, but for the purpose of this lab, you will pretend to lend her your computer and execute all the commands for her.

Open a second terminal for your teammate and use it to create a new directory for her (anywhere, e.g., `mkdir ~/co7095-teammate/` , except within your own `msword` directory). Move into this directory (`cd ~/co7095-teammate/`), and repeat step (1) to clone the msword project from github into her directory.

8. Pulling changes

Now, this new developer is not up to date with the work that you’ve been doing, so she needs to update her version of the repository with your recent changes. She can do this by *pulling* all of the changes you’ve made onto her own directory with the `git pull` option:

Change directories into the newly checked out msword directory `cd ~/co7095-teammate/msword`. Then execute:

```
git pull <path_to_msword_directory_with_the_file_you_recently_edited>
```

If you have followed the previous steps to the dot, it should be `git pull ~/co7095-lab2/msword`
Now she and you both have identical versions of msword.

9. Logs

You can inspect the various changes that have happened over the course of the various commits by executing

```
git log
```

And if you want to limit the number of commits to be shown

```
git log -n <number of commits you wish to see>
```

For example, to verify that your teammate's repository includes your commit from Step 5, execute:

```
git log -n 1
```

You should get exactly the same output if you execute that command in her or your `msword` directory.

10. Conflicts

Conflicts between different developers in the same project are undesirable but do occur in practice and resolving them can be challenging. You will simulate a conflict and its resolution now.

Let us assume you don't like the word "buggy" in `Opus/dialog2.c` and want to change it to "unreliable". In your own copy of the repository (*in the first terminal you opened*), open that file, and replace "buggy" with "unreliable". Add the change, and then commit it using the instructions in Step 5.

Unbeknownst to you, your teammate had precisely the same reservations about the word "buggy". In her version (*second terminal*), edit the same file, but instead of changing "buggy" to "unreliable", change it to "suboptimal". Again, add the change and then commit it using the instructions in Step 5.

Now, *back to your own terminal*, you want to move on to work on a different coding task. In that situation, it is recommended to ensure that your version of the repository is up to date. To make sure of this, you decide to pull your teammate's latest version:

```
git pull <path_to_new_devs_msword_directory>
```

If you have followed the instructions in full details, this would be

```
git pull ~/co7095-teammate/msword
```

At this point, Git highlights that there is a conflict with the following output.

```
Auto-merging Opus/dialog2.c
CONFLICT (content): Merge conflict in Opus/dialog2.c
Automatic merge failed; fix conflicts and then commit the result.
```

This message means that Git did not know how to combine the changes you both have made, because you both have changed the same text in different ways.

11. Resolving conflicts

To assist you in resolving the conflict, Git has edited the file containing the conflict. Open up `Opus/dialog2.c` again. You should be able to find the conflicting changes:

```
<<<<<<< HEAD
  /* somewhat unreliable feature disabled for version 1.0 */
=======
  /* somewhat suboptimal feature disabled for version 1.0 */
>>>>>>> 6828cba90cba762b492d07872d8814c973a258bb
```

Git highlights conflicts in that way. The text in between the lines containing `<<<<<<< HEAD` and `=======` is your local version. Everything after the line containing `=======` and the line containing `>>>>>>> <hash code of remote version>` is the text in your teammate's version.

To resolve the conflict, edit the file so that it contains the version you want (this often involves discussing with the conflicting developer to agree on what version to keep), and remove the additional lines added by Git.

Now use `git add` and `git commit` as in Step 5 to commit your resolved version. If you were adding and committing specific files, you may come across the error `fatal: cannot do a partial commit during a merge.`. To avoid this, use the `git commit` command without specifying the files to commit.

12. Branching

Your teammate now decides that she is going to make more substantial changes in several files in the system, but she does not want her interim changes and commits to disrupt your work. To enable this, she decides to create her own *branch* in the repository where she can work on and commit to without having her changes automatically merged with your version when you try to pull her updates to the main development version.

In your second terminal again (in her `msword` directory), create a new branch:

```
git branch <branch name>
```

e.g.

```
git branch Refactoring
```

Now the branch has been created, but we are not actually operating in that branch (we are still in the main branch). You can check this by executing `git branch` (without branch name), which should output:

```
  Refactoring
* master
```

To work on the new branch, you have to switch to it by using the checkout command:

```
git checkout Refactoring
```

Try `git branch` again to confirm you are now on the `Refactoring` branch. Once you have done this, pick a couple of files, edit them, and commit these changes (again, `add` followed by `commit` as in Step 5).

13. Merging

Having made her edits in her branch, she is ready to merge them back in with the main version so that you are able to update your own repository with her changes.

To help her merge her changes into the main (master) branch, first change her working branch from `Refactoring` to `master`:

```
git checkout master
```

Then, merge the current version (master) with the changes in the Refactoring branch:

```
git merge Refactoring
```

Use the `git log` command to check that the changes made in the `Refactoring` branch are now in the `master` branch. After confirming this, we can safely delete the temporary branch we created (using the `branch -d` option):

```
git branch -d Refactoring
```

Now, go into your own repository (first terminal) and pull her recent changes again.

```
git pull ~/co7095-teammate/msword
```

14. Keeping track of changes

Remember that you can and should use `git log` regularly to keep track of the various changes that have happened over the course of a software development project. Doing so is an easy way of preventing conflicts in the first place.

Well done!

You have completed the Git lab!