GIT

CSE5006 – LAB 1

TABLE OF CONTENTS

1. INTRODUCTION TO GIT

1.1. WHY USE VERSION CONTROL?

The most straightforward way of organizing the codebase for a software project is to simply have an ordinary directory somewhere containing all of the source files. When you want to make a change, you simply open a file and change it in your favorite text editor, saving to overwrite the previous version. This way of working should be familiar to you from the labs and assignments in previous subjects. But what happens if you introduce a bug and want to go back to an old version of the software? You could start copy-pasting files and folders, but this quickly becomes tedious and awkward.

Version control systems (VCS) are designed to help you keep track of changes to your code as it is developed. When using a VCS, you are able to commit changes as you develop, which is a bit like saving a snapshot of your project. With a VCS, you can:

- Develop your code without fear of messing everything up - you can always revert to an earlier commit.
- Produce detailed backups of the work you do.
- Go back through your project history to determine which code change introduced a bug.
- Analyze the development history of your project.
- Permit other people to contribute to your project - you can integrate changes made by multiple people at once.
- Work simultaneously on multiple different versions of your project - you can work on new features separately, then merge them later.
- Identify which developer introduced a particular line of code.

These days, you will find that almost all software is developed using version control, with Git being the dominant tool used to facilitate this. We will be using Git as our VCS in this subject.

1.2. CONFIGURING GIT

Before we continue any further, we should configure Git so that it knows who is editing code on this computer. This is useful when multiple people are working on the same project, and later on, you need to find out who made what change. Run the following commands, but fill in your own email and name instead of the **`<your_student_email>`** and **`<your_full_name>`** placeholders.

```
git config --global user.email "<your_student_email>"

git config --global user.name "<your_full_name>"
```

2. STARTING A GIT REPOSITORY

In Git, the term "repository" is used to refer to all of the files in your project along with its complete history of all changes ever made. There are two ways in which you can get a new repository on your machine. The first way is to clone an existing Git code repository, usually from the Internet. The second way is to start a completely original repository from scratch by initializing Git in a local project.

2.1. CLONING AN EXISTING REPOSITORY

Firstly, open up a terminal. Make a new directory for this lab and change into it, as shown below (I will use the dollar sign to represent the prompt, don't actually type in a "$"):

```
mkdir -p ~/Documents/Labs/Lab01

cd ~/Documents/Labs/Lab01/
```

Let's say that you want to download a local copy of the "marked" project. In order to do this, you need to know the Git clone URL for the repository - in this case, it is https://github.com/chjj/marked.git.

```
git clone https://github.com/chjj/marked.git
```

Great, the repository has now been downloaded to your computer. Let's take a look at what we just grabbed from the Internet.

```
ls
```

The output of **`ls`** shows that a directory called "marked" has been created and populated with the most recent version of all project files. Let's change into the "marked" directory and explore a little bit more.

```
cd marked

git log --abbrev-commit --pretty=oneline
```

The **`git log`** command prints a summary of commits that were made to the repository. More on commits later, for now, just understand that we have access to the entire project history, not just the current files. Hit "q" to quit the log.

We don't actually need a copy of the "marked" project right now, so you can go ahead and delete it.

```
cd ..

rm -rf marked
```

2.2. STARTING A NEW REPOSITORY

Now we'll create a new project repository from scratch. Begin by making a new directory for the project and changing into it.

```
mkdir my-awesome-project

cd my-awesome-project
```

Create a new file so that the project isn't empty.

```
echo "My project is awesome, and I'm awesome." > README
```

Now we have a README file with some humble text inside. Let's create a new Git repository so that we can start tracking changes.

```
git init .
```

Read this command as "initialize a Git repository in this directory". Now we'll ask Git about the current status of our newly created repository.

```
git status
```

Oh no, red text! Don't worry, it's not that bad - Git is simply telling us that the README file is untracked. What we need to do is tell Git that we are interested in storing this file in our repository.

```
git add README
```

Note that you can also add entire directories, which is handy if your initial project has a lot of files.

```
git status
```

Ah, much better - green text. We say that changes highlighted in green like this are "in the staging area". This means that they are ready to be committed into the latest chapter of our repository's history.

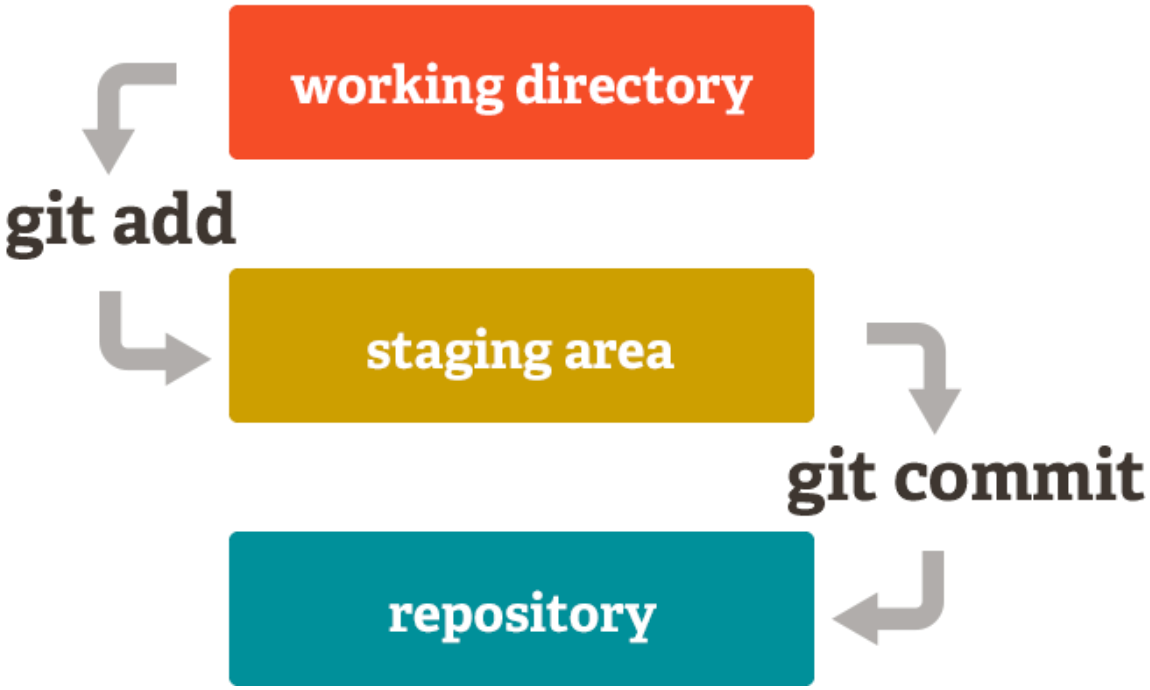
```
git commit -m "First commit"
```

Read this command as "commit all of the changes in the staging area into the repository with the message 'First commit'." Commits are the most common way of updating a repository and describe the addition, removal, or modification of one or more files. It is a good idea to take some time to write concise and informative commit messages that summarize the changes made since the previous commit. In this case, there is no previous commit, so it's not as important.

```
git status
```

Git will now report that the working directory is clean. Congratulations, you have successfully saved a snapshot of the current project state into the Git repository.

If you are having trouble understanding the different "areas" of Git, it may be easier to visualize things as a diagram.



When you edit, create, or delete a file, that change is made in the working directory. To tell Git about a change, it must be added to the staging area. Adding a change to the staging area does not affect the repository itself, and you are free to use `git add` multiple times as you are working (though this is not usually necessary). Once you have accomplished some complete unit of work, you can commit all changes in the current staging area to the repository. This will create a "checkpoint" in the project's history that you can revert back to later if need be. Making a commit effectively "clears" the staging area, allowing you to repeat the process again.

3. GITHUB

In the previous section, we created a new Git repository locally. However, in most cases, we want to store a copy of the repository on the Internet so that other people can download the code to use or collaborate. There are a few options available when it comes to hosting Git repositories on the Internet. We will be using GitHub.

3.1. CREATING YOUR GITHUB ACCOUNT

Go to https://github.com and create a new account or use an existing account if you would like.

3.2. LOGIN ON THE COMMAND LINE

3.2.1. CREATING A GITHUB PERSONAL ACCESS TOKEN

1. Go to the GitHub website https://github.com/ and log in to your account.
2. Click on your profile picture in the top-right corner of the screen and select Settings.
3. On the left sidebar, click on Developer settings.
4. In the left sidebar of the next screen, click on Personal access tokens and then tokens classic.
5. Click on the Generate new classic token button.
6. Give your token a descriptive name in the Note field.
7. In the Select scopes section, select the repo checkbox to add all repo related permissions.
8. Scroll down and click on the Generate token button.
9. You will now see your new token. Make sure to copy the token and store it safely; you will not be able to see it again!

3.2.2. LOGGING IN WITH GITHUB USING GIT VIA COMMAND LINE

Now that you have your personal access token, you can use it to authenticate your Git operations via the command line.

1. Go to https://github.com/new to create a new repository.
2. Set the name as 'my-awesome-project'.
3. Make sure the repository is public.
4. Click the create repository button.
5. Open the terminal/command prompt.
6. Navigate to your local 'my-awesome-project' Git repository.
7. Add your GitHub repository as a remote repository with the command replacing **username** with your GitHub username:

```
git remote add origin https://github.com/username/my-awesome-project.git
```

8. When you need to perform an operation that requires authentication (e.g., git push), you will be asked for your username and password:

```
git push --set-upstream origin master
```

9. Enter your GitHub username as the username.
10. Use your personal access token as the password.

Remember, your personal access token should be kept secret, just like your password. If you believe your token has been compromised, you can easily create a new one and delete the old one from the GitHub settings.

3.3. CREATING A REPOSITORY

3.3.1. CREATE THE REPOSITORY

Just like before, except with a few steps to help us with cloning!

1. Sign into your GitHub account at https://github.com/.
2. Once you're logged in, click the '+' icon in the upper right corner next to your profile picture and select 'New repository'.
3. Name your repository in the 'Repository name' field. This name will be used in the URL for the repository on GitHub. Optionally, you can add a description in the 'Description' field.
4. Decide if you want your repository to be public (visible to everyone) or private (visible only to you and people you invite).
5. Initialize your repository with a README.
6. Click the 'Create repository' button.

3.3.2. CLONE THE REPOSITORY

Now that you have a repository on GitHub, you can clone it to your local machine using Git from the command line.

1. Navigate to the main page of your repository on GitHub.
2. Above the list of files, click the 'Code' button. This will open a dropdown.
3. In the 'Clone' section, click the clipboard icon to copy the clone URL for the repository. This URL should look like https://github.com/username/repo.git, where 'username' is your GitHub username and 'repo' is the name of your repository.
4. Open your terminal/command prompt.
5. Navigate to the location where you want the cloned directory to be made.
6. Type git clone, paste the URL you copied in step 3, and press enter. The command should look like git clone https://github.com/username/repo.git.
7. Your local clone will be created.

Remember, if the repository is private, you'll need your GitHub credentials to clone it. If you're using token-based authentication, as described in the previous instructions, you'll use your personal access token as the password when prompted.

3.4. FORKING A REPOSITORY

Forking a repository on GitHub creates a copy of that repository under your GitHub account. This allows you to freely experiment with changes without affecting the original project.

Here are the instructions on how to fork the "https://github.com/CSE5006/lab-1" repository:

1. Open your web browser and navigate to the repository's URL: https://github.com/CSE5006/lab-1.
2. At the top-right corner of the page, you'll see a button labeled Fork. Click on this button.
3. Wait for GitHub to create the fork. This process usually takes a few seconds.
4. Once the forking process is complete, you'll be automatically redirected to your new forked repository. The web address will look something like "https://github.com/YourUsername/lab-1", where "YourUsername" is your GitHub username. You can confirm that the fork was successful by looking at the description above the file list, which should say "forked from CSE5006/lab-1".

Now you have a forked copy of the original repository in your GitHub account. You can clone it to your local machine, make changes, and push updates just like any other repository you own.

3.4.1. EXERCISE 1

Currently, the word "red" on the web page is displayed as blue text. Open up the file **styles.css** in a text editor (double click on it) and fix this "bug" by changing the color to red. Save the file and refresh the web page to ensure that the color is correct. If all is well, use the following commands to commit the change and push it to the remote repository:

```
git add styles.css

git commit -m "Fixed red text colour"

git push
```

3.5. USING BRANCHES

One of the most powerful features of Git is the concept of branches. You can think of a branch as a parallel universe for your code. You can have many different branches at the same time containing different versions of your project. These don't represent past versions of your project like old commits do, they represent alternate realities. The main branch in Git is called "master" or for newer projects, “main”, and is the default branch that we have used so far.

Branches may sound really complicated and a bit useless at first, but they make it super easy to manage adding new features to a project - especially when other developers are involved. Let's create a new branch for adding a new section to the web page.

```
git checkout -b new_section
```

Read this command as "check out a new branch called 'new_section'." "Checking out" is Git terminology for switching to a different branch.

Right-click on **`index.html`** and select "Open With" and then choose **`gedit`**. Immediately below the line containing the comment `TODO: Add new section here`, insert the following HTML:

```
<div id="cool-section"></div>
```

This HTML will add a new container element to the page with ID "cool-section". We can use this ID from within the CSS file to give the section some visual styles. Add the following code to the styles of the file.

```
#cool-section {
    border: 4px solid gray;
    background-color: yellow;
    height: 300px;
}
```

Refresh the web page in the browser and ensure that the new section is visible. Now we can use a variant of the **`git add`** command which allows us to review all file modifications and selectively add them to the staging area, ready to commit.

```
git add -p
```

When asked whether to "Stage this hunk," enter "y" for yes - this will tell Git that you want to add each change to the staging area. Now that the changes are staged, you can commit them and then push them to the remote repository on GitHub.

```
git commit -m "Added a cool new section"

git push -u origin new_section
```

Voilà! The **`new_section`** branch has been uploaded to the remote GitHub repository. Now we are going to switch back to the **`main`** branch.

```
git checkout main
```

Note that we do not need the **`-b`** option because the **`main`** branch already exists. Reload the page in the web browser again.

Oh no! Where are my changes! Git overwrote my work! My boss is going to fire me! I'm going to be a nerf herder for the rest of my life! Woah, slow down there buddy, there's no need to panic. Your changes are still all there, they just exist in a different branch.

```
git checkout new_section
```

Refresh again. See? No changes were lost. Now back to the **`main`** branch.

```
git checkout main
```

In a team scenario, fellow project collaborators would now take a look at the code in the **`new_section`** branch, and hopefully approve the changes. Once the new feature is approved, it then needs to be merged back into the main branch, **`main`**. Run the following command to merge **`new_section`** into **`main`** (this does not need to be specified explicitly since **`main`** is the current branch):

```
git merge new_section
```

Confirm that the **`<div id="cool-section">...</div>`** is in **`index.html`**, then push to GitHub.

3.5.1. EXERCISE 2

Create a new branch called **`add_textarea`** and checkout that branch. Make the following change to **`index.html`** so that a text area is shown inside the **`<div id="cool-section">...</div>`**:

```
    <div id="cool-section"><textarea></textarea></div>
```

Once you have confirmed that the web page now shows a text area within the yellow section, use Git to add and commit the change. Checkout the **`main`** branch, merge the **`add_textarea`** branch into it, and push to GitHub.

4. SUMMARY

You can ask Git to print out a visualization of the history of your project, including branching and merging (don't forget that you can hit 'q' to quit when you're done).

```
    git log --graph --abbrev-commit --pretty=oneline
```

Congratulations on reaching the end of today's Git journey!

4.1. GIT QUICK REFERENCE

Command	*Description*
git clone <url>	Download a local copy of the Git repository.
git init .	Initialise a new Git repository in the current directory.
git status	Display the current status of the working directory. Shows files that are unstaged or uncommitted.
git add <file>	Stage a file.
git add -p	Go through all modifications and choose whether to stage them.
git commit -m <message>	Commit staged changes.
git remote add origin <url>	Tell Git where the remote repository is located.
git push -u origin <branch>	Update the remote branch specified to match current local branch state.
git push	Same as above after the above command has been used once.
git pull	Update the current local branch to match the remote counterpart.
git merge my_branch	Merge my_branch into the current branch.
git log	Show the repository's commit history.