SENG2021/SENG3011: Using GitHub

1. Why GitHub

Version control systems have become common practice amongst most software companies and utilising it within a team setting is a key skill to have. In this workshop teams will be required to use git for version control and github as their remote repository service. Submissions will be done using this service. **Teams will be required to add their mentors into their repository as soon as possible.**

Watch the following videos by Hussein to familiarise yourself with git

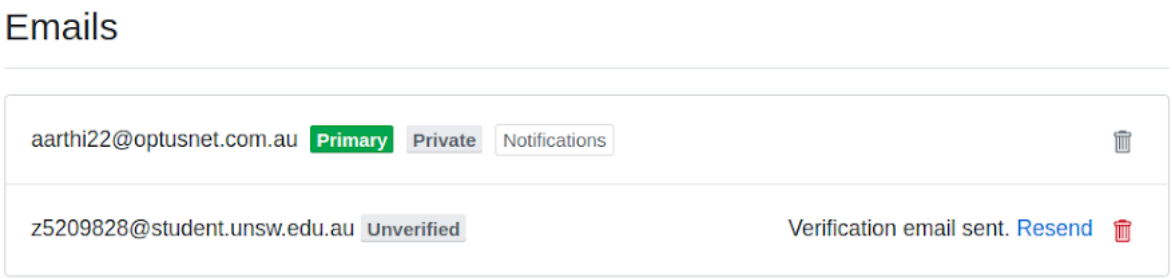
Git videos - Google Drive

2. Sign up for GitHub

GitHub is a version control platform (like Bitbucket or GitLab) that uses git. It is a great way for developers to collaborate with one another. It will be the primary source of starter code distribution as well as where you will submit project source code and reports before being marked by your mentors. You will have to create an account here before you can start. Getting familiar with GitHub and how to use it is the aim of this lab.

Instructions:

- I DON'T have an account on GitHub: Create an account at GitHub using either your student email (zID@student.unsw.edu.au) or your personal email.
  - **Make sure to verify your email address. If you do not get an email straight away, go to https://github.com/settings/emails and click resend**



- I ALREADY have an account on GitHub: Continue to the next step.

3. Install git

Throughout the course you will need to be comfortable with git. It comes pre-installed on most linux releases and is already installed on the CSE machines. To check if git is installed on your local machine use the command

```
git status
```

If it is installed you will see something like

fatal: Not a git repository (or any of the parent directories):

If you do not have git installed, you will see something like

bash: git: command not recognized

If this is the case, you will have to set it up using the following instructions

- **Linux** - Follow instructions at https://git-scm.com/download/linux
- **Mac** - `brew install git`

Once it has been installed, we also have to configure git.

Instructions:

1. Try `git status` to check whether git is installed
   - If not, follow one of the above instructions depending on your OS
2. Configure git if you have not used it before with the following commands (including the quotes)

```
git config --global user.name "Your Name"

git config --global user.email "github_email@example.com"
```

4. Create and cloning your project repo

Creating and Cloning a **repository** (a repo is just a directory that is linked with git) is how the codebase is linked from GitHub to your local computer so changes you make can be saved and shared with others. It is the final step before you can start making changes and contributing. A repo can be cloned at any time by someone who has access, so they can start working whenever they want. When a repo is cloned, all code that is uploaded on the server is copied to a desired location on your local machine.

Go to your github home page and create new repository for your project with the name **seng2021-<team-name>**, adding all team members as contributors to the repository. Also, give access to your mentor and the course administrator, as they will be visiting the repo to evaluate your progress. Make sure only one member of the team is doing this to avoid multiple repos per team.

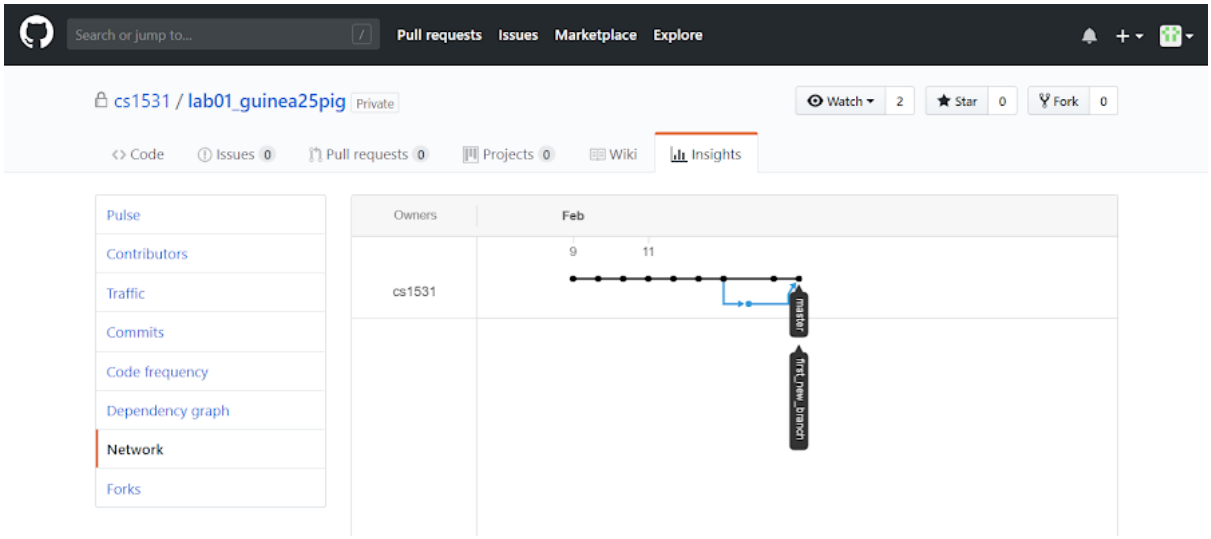
Once the repository is created all the team members can add it into their computer by following steps:

- Open a Terminal and navigate to the folder you want the repo to be in
- `git clone [link]` (Replace [link] with the repository URL)

- Type `ls` to ensure the folder has been copied correctly
- `cd [repo_name]` to navigate into the cloned repo
- Type `ls` again to see the copied files.

Quick Tip: Viewing your Git history on GitHub

GitHub provides a decent visualisation of your repo's commit history, which you can access at Your Repo > Insights > Network as shown below. Your initial history will look different to what is displayed here, but watch it evolve as you walkthrough each of the steps below (refresh the page after every pushcommand).



5. Make your first commit

Now that you have cloned the repo, you are ready to work on the codebase locally.

A commit is an update of the remote (on the GitHub's server) state of the repository. It saves changes that you have made and can be given a message to describe those changes. A good use of git involves a lot of commits with detailed messages.

Before you can commit, you have to do what is called staging your changes, which effectively tells git what changes you actually want to commit and what changes you don't at the moment.

Commits are often followed by pushing which is how git actually uploads your commits to the remote server.

The command to commit and push are as follows:

```
git add [files_to_commit] # Stage

git commit -m "Detailed message describing the changes" # Commit

git push # Push
```

Example:

1. Add a new file called `first.txt` in the repo directory
2. Add a line of text to the first.txt file using your favourite text editor and save.
3. Go back to your terminal and enter the following commands:

```
git add first.txt`

git commit -m "Added a line to the first file"

git push
```

4. MAKE SURE YOU UNDERSTAND THE PURPOSE OF EACH OF THE 3 ABOVE COMMANDS! If you are unsure about any of them, ask your tutor or rewatch the tutorial videos.
5. Go back to GitHub and confirm that your changes have been pushed to the server.

6. Do your first pull

Usually when you are using git, it is in a team. That means that you will not be the only one who is making the changes. If someone else makes a change and pushes it to the server, your local repo will not have the most up to date version of the files. Luckily, git makes it easy to update our local copy with the `git pull` command.

This command checks the remote server that your local repo is linked to and makes sure that all of your files are up to date. This ensures that you don't accidentally do things like implement the same thing someone else has already done and also lets you use other people's work (eg. new functions) when developing.

Pulling regularly is one of the **most important** practices in git!

Unfortunately, at the moment you are just working individually. But GitHub still gives us a nice way to practice a `git pull`.

Example:

1. View your repo on GitHub.
2. Click on the `first.txt` file
3. Click the small 'edit' pencil icon in the top right
4. Make any change you want to this file and click the **Commit Changes** button at the bottom of the screen
5. This will have changed the `first.txt` file on the server but not on your local environment (check it!). To fetch these changes use the `git pull` command from your terminal
6. Confirm that your version of `first.txt` now has the changes you made on the web page

7. Create your first branch

Branches are a vital part of git and are used so people can work on separate parts of the codebase and not interfere with one another or risk breaking a product that is visible to the client. Breaking something on one branch does not have an impact on any other.

Good use of git will involve separating parts of the project that can be worked on separately and having them in their own feature branch. These branches can then be merged when they are ready.

Useful commands for branches:

```
git checkout -b [new_branch_name] # Create a new branch and switch to it

git branch                        # List all current branches

git checkout [branch_name]        # Switch to an existing branch
```

Example:

1. Make your new branch with: `git checkout -b first_new_branch`
2. List your branches to see that you have indeed swapped (use the above commands)
3. Make another change to the `first.txt` file
4. Try to push your changes to the server using the commands you learnt in the *Make your first commit* section
5. The above step should have given you the following error:
   — `fatal: The current branch first_new_branch has no upstream branch.`
   This means that the branch you tried to make a change on doesn't exist on the server yet which makes sense because we only created it on our local machine.
6. To fix this, we need to add a copy of our branch on the server and link them up so git knows that this new branch maps to a corresponding branch on the server
7. `git push -u origin first_new_branch`

NOTE: The final step above must be run on the 1st push to every new branch that you have created on your local machine. After you have run this once, you should go back to simply using git push

8. Merge your two branches

Merging branches is used to combine the work done on two different branches and is where gits magic really comes in. Git will compare the changes done on both branches and decide (based on what changes were done to what sections of the file and when) what to keep. Merges are most often done when a feature branch is complete and ready to be integrated with the master branch.

Once we have finished all that we are going to do (and think there are no bugs) on the branch, (e.g. *first_new_branch*we) we can merge it back into master. It is a strong

recommendation to have a version of the code that at least runs on master so people are not completely blocked. (DO NOT PUSH BROKEN CODE TO MASTER)

Another recommendation is to merge master into your branch before merging your branch into master as this will ensure that any merge into master will go smoothly.

Commands for merging two branches

```
git merge [target] # Merge the target branch into current
```

NOTE: A successful merge automatically uses the commits from the source branch. This means that the commits have already been made, you just need to push these to the server (`git push`)

Example:

1. Switch back to the master branch using one of the commands fom the above section
2. Merge in the changes you made in the other branch
   `git merge first_new_branch`
3. Push the successful merge to the server to update the master branch on the server

9. Engineer a merge conflict :(

Merge conflicts are the one necessary downside to git. Luckily, they can be avoided most of the time through good use of techniques like branches and regular commits, pushes and pulls. They happen when git cannot work out which particular change to a file you really want.

For this step we will engineer one so you can get a taste of what they are, how they occur and how to fix them. This will be the LAST time you will want one. The process may seem involved but it is quite common when multiple people are working at a time.

Example: (All commands have been presented above)

1. Add a line to the top of the `first.txt` file (on *master branch*)
2. Add, commit and push your changes
3. Switch to your *first_new_branch*
4. Add a different line to the top of the `first.txt` file
5. Add, commit and push your changes
6. Merge master into your current branch
7. This sequence of steps should made a merge conflict at the top of the `first.txt` with the following output
   ```
   Auto-merging first.txt
   CONFLICT (content): Merge conflict in first.txt
   Automatic merge failed; fix conflicts and then commit the result.
   ```

10. Resolve your merge conflict :)

Resolving a merge conflict is as simple as editing the file normally, choosing what you want to have in the places git wasn't sure.

This is a very simple example, but merge conflicts can be large and in many different places across a file/repo. If possible, avoid merge conflicts. This can be done by regularly pulling from the server to update your local copy and by making your branches in such a way that they handle only one feature/section of the code.

A merge conflict is physically shown in the file in which it occurs.
<<<<<<< marks the beginning of the conflicting changes made on the **current** (merged into) branch.
======= marks the beginning of the conflicting changes made on the **target** (merged) branch.
>>>>>>> marks the end of the conflict zone.

E.g.,

```
This line could be merged automatically.

There was no change here either

<<<<<<< current:sample.txt

Merges are too hard. This change was on the 'merged into' branch

=======

Merges are easy. This change was made on the 'merged' branch

>>>>>>> target:sample.txt

This is another line that could be merged automatically
```

This above example could be solved in many ways, one way would be to just use the changes made on the target branch and delete those made on the current branch. Once we have decided on this we just need to remove the syntax. The resolved file would be as follows

```
This line could be merged automatically.

There was no change here either

Merges are easy. This change was made on the 'merged' branch

This is another line that could be merged automatically
```

We would then just commit the resolved file and the merge conflict is finished!

Example:

1. Open the `first.txt` file and decide which (or both) changes you want to keep
2. Remove the merge conflict syntax
3. Add, commit and push the resolved merge conflict
4. en up `first.txt` on *master* branch - you see that mMerging *master* branch into *first_new_branch*has only affected *first_new_branch*.
5. Now, merge *first_new_branch* into *master* to synchronise the two branches (this step should not cause any conflicts), then push the changes.
6. Check your git history visualisation on GitHub :)

11. Suggested Git workflow for the Project

Using git effectively and efficiently vastly improves a team's productivity. Teams in many software engineering firms will form git workflows to improve team collaboration and to prevent conflicts before they can occur.

Nominate one person from your group to manage the github repository. They will be responsible for creating the github repo, adding the start code and be the only one with push access to master.

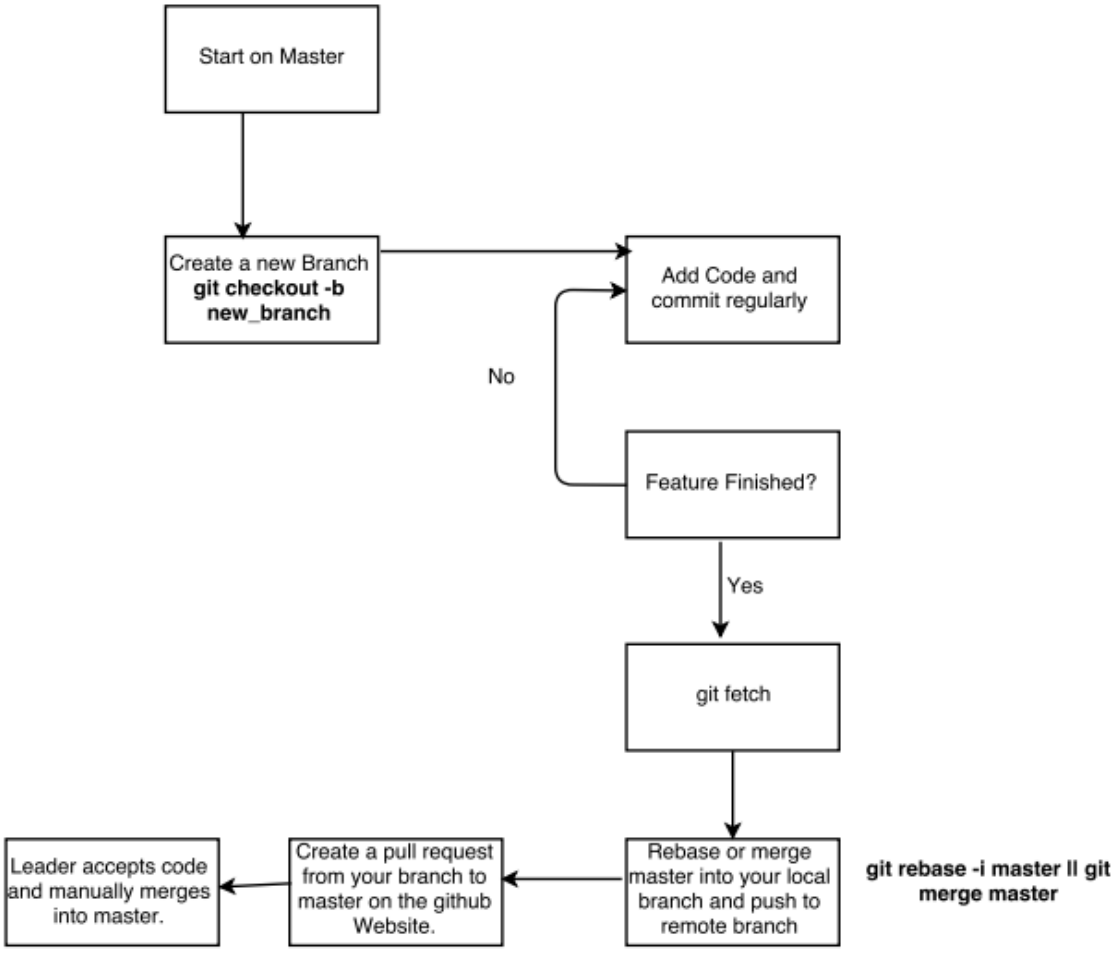
The workflow is as followed:

1. Create a separate branch from master (perhaps for a new feature to implement or a bug fix). Ensure your master is up to date before doing this.
2. Work on the feature or bug fix as normal and commit regularly to your new branch. Push this branch to the remote repository consistently (**git pull origin new_feature**)
3. When you have completed the feature or bug fix, perform a **git fetch** and **git merge master,** to ensure your branch is up to date with any new changes on the master branch. Ensure you resolve any merge conflicts before proceeding. Push your changes to the remote branch.
4. Go to your project page on github and create a pull request for your branch to be merged into master.
5. The leader will review the pull request and manually merge the feature branch into master. The branch used to work on the feature/bug fix may be deleted.

There are many ways to expand upon it such as including code reviews and unit tests that each feature must pass etc. The developer may also choose to rebase their branch before submitting a pull request for a cleaner commit history.

It is highly recommended that each team invests time in designing a workflow to ensure a smooth development process and less code and team conflicts.

A flow chart for the workflow is presented on the next page.



Account monitoring by mentors

We will facilitate submissions by having mentors check a team's repository. They will view the last commit timestamp before the set deadline for a given deliverable. This includes both prototypes and report submissions.

A small amount of marks can be gained for the final prototype based on a good usage of the repository between team members. Mentors will monitor progress in mentoring sessions by checking a team's repository and providing feedback on their usage.

Resources for learning git

https://learngitbranching.js.org/

http://jameschambers.co/writing/git-team-workflow-cheatsheet/

https://www.sitepoint.com/getting-started-git-team-environment/

https://www.atlassian.com/git/tutorials/comparing-workflows

https://www.atlassian.com/git/tutorials/merging-vs-rebasing