Workshop 2: Git Hands-On Activity

DSA3101 AY 25/26 Sem II

Contents

1 Preparations before WS2: Setting up Git

1.1 Name and Email Configuration

Once you have installed Git, it is good to carry out some initial configuration. Open up Git Bash terminal, or Windows Command Prompt (CMD), and enter the following commands. They will store your username and email.

```
git config --global user.name "yourname"
git config --global user.email "yourname@domain.com"
```

Without this step, you will not be able to make commits. Instead, you will encounter this error:

```
Author identity unknown
*** Please tell me who you are.
Run
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
to set your account's default identity.
Omit --global to set the identity only in this repository.
```

If you are not sure whether you have set it up appropriately, you can enter the following command.

```
git config --global --list
```

1.2 Text Editor Configuration

When we resolve merge conflicts, or when we need to enter a merge message, Git will open up a text editor. By default, it uses `vim`. However, this is not the preferred choice for everyone. With this next command, you can set it to your preferred text editor, e.g., Notepad++, Atom, etc.

```
git config --global core.editor notepad
```

You can enter the following command again to confirm your name, email address, and editor have been set up appropriately.

```
git config --global --list
```

Once your username, email, and default core editor are displayed correctly, you're ready to continue.

1.3 Passwordless Connection to GitHub

When working with version control software, we are encouraged to make small, modular changes frequently. Pushing and pulling to/from GitHub will be frequently called. As we do not want to be entering our password every time we do so, we shall set up SSH keys that allow us to secure the connection to GitHub.

SSH keys work in this way. Each individual generates a public and a private key pair. In order to encrypt a message, only the public key is necessary. However, in order to decrypt a message, both the public and private keys are needed. This set-up is used to secure the connection between GitHub and our machines.

To set this up, we need to generate a public-private key pair, and then share the public key with GitHub. The instructions can be found at this link: https://docs.github.com/en/authentication/connecting-to-github-with-ssh.

For more details, refer to the E-learning video 2-3: Setting up SSH on Canvas to complete the setup.

1.4 Cloning a Repository

Open up your preferred terminal, and navigate to the directory you wish to save the Git repository (which is essentially just another folder).

For example, I want to save all files in the following directory within the DSA3101 folder.

If you use Git Bash, enter the following:

```
cd /c/Users/liuyq/DSA3101/
git clone https://github.com/dabainiu617/dsa3101-2510.git
```

Note that to paste a copied text in Git Bash, you could press "Shift + Insert".

If you use CMD, enter the following:

```
cd C:\Users\liuyq\DSA3101\
git clone https://github.com/dabainiu617/dsa3101-2510.git
```

Note that to paste a copied text in CMD, you could press "Ctrl + V".

We can use `git log` and `git show` to see what changes were recently made to the repo.

```
git show -3     # shows last 3 commits

git log --since="1 day" # Lists commits created within the last day
```

We will try more variations of "git log" later.

2 In-class Activity 1 (ICA1): Making Commits

There are two in-class activities for WS2. The structure is as follows.

Activity	Topic	Key Skills & Commands
ICA1	Basic Actions	`git status`, `git pull`, `git add`, `git commit`, `git push`
	Edit Files, Differences, and Recovery	`git diff`, `git log`, `git checkout`, `git restore`
ICA2	Branching	`git branch`, `git checkout`, `git switch`
	Merge without Conflicts	`git merge`
	Merge with Conflicts	Manual conflict resolution

Table 1: Git Activities and Associated Skills

2.1 Start

Open up Git Bash terminal, or Windows Command Prompt (CMD), and navigate to the directory containing the WS2 materials for today.

For Git Bash:

```
cd /c/Users/liuyq/DSA3101/dsa3101-2510/workshop_materials/WS2
```

For CMD:

```
cd C:\Users\liuyq\DSA3101\dsa3101-2510\workshop_materials\WS2
```

2.2 create a new file and push it to Git Repository

First, check whether your local repository is synchronised with the latest version of the Git repository.

```
git status
```

It is good if you see the following message:

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

nothing to commit, working tree clean
```

Otherwise, enter the command:

```
git pull origin main
```

Navigate to the practice sub-folder, create an empty .txt file, and name it using no more than 10 characters (letters and numbers) that you can easily remember. The following code uses the file, testYQ.txt, as an example. please replace it with your own name for the remaining code.

Open the txt file, and type “version 1:”.

Or you could enter:

```
cd pratice
echo version 1: > testYQ.txt
```

Now we are ready to publish and push it to the Git repository. Ensure you are in the practice subfolder. We first git add and git commit with an appropriate message.

```
git add testYQ.txt
git commit -m "Initial commit: version 1"
```

Typically, we perform a git push immediately after a git commit. However, with over 150 of us working together, attempting to push at the same time may lead to certain issues. We’ll practice it and experience this firsthand at the end of Activity 1.

2.3 Git Diff and Git Log

Next, we simuate we made some changes, call it version 2, and ready to commit it. Note that we will not push it for now.

```
echo version 2: > testYQ.txt
git add .
git commit -m "Second commit: version 2"
```

Now, let us compare them. We first view the logs of this file.

```
git log -- testYQ.txt
```

You will likely see two commits similar to the ones below.

```
commit 8d82291ee8e4de0befd7bf8a348d068041be1432 (HEAD -> main)
Author: YQ <dabainiu@gmail.com>
Date:   Thu Jul 17 13:37:12 2025 +0800

    Second commit: version 2

commit 415b3180940f079ec9a198b00eb96a6c4a710f5a (origin/main, origin/HEAD)
Author: YQ <dabainiu@gmail.com>
Date:   Thu Jul 17 13:24:33 2025 +0800

    Initial commit: version 1
```

The commit id is the sequence starting from 8d8229 and 415b318. We could refer to one commit using only the front few characters.

Then we could compare two commits to see the changes.

```
# git diff ID1..ID2 -- filename
git diff 415b318..8d8229 -- testYQ.txt
git diff 415b318..8d8229
```

As these two commits only changed one file, you could waive the file part. The red color part represents the deletions, while the green color part represents the insertions.

The order of the two commit IDs matters. Typically, the ID that appears first represents an earlier commit than the one that follows.

If you only want to compare the differences between the two most recent commits for a particular file, you can also use the following command:

```
git diff HEAD~1 HEAD -- testYQ.txt
```

Now, let us display the logs related to this particular file.

```
git log --stat -- testYQ.txt
```

This helps us identify who committed which file, the commit message, and how many lines were changed.

Take a moment to consider the text inside the parentheses, origin/main and HEAD -> main, and reflect on what each part means.

If too many commits are displayed, press Q to exit the viewer mode.

There are various ways to view Git logs. The following command is commonly used for a quick overview of key changes.

```
git log --pretty=oneline -- testYQ.txt
```

The following command is useful for visualising the commit history as a tree structure.

```
git log --graph --all -- testYQ.txt
```

At this point, since we haven't created any additional branches, the commit history appears as a straight linear sequence.

To get help on any of these commands, run it the command with the help option. For instance:

```
git log --help
git diff --help
```

2.4 Recovery of files: Git Checkout and Git Restore

One of Git's key strengths is its ability to track the full history of changes, making it easy to recover any previous version.

For example, if you wish to recover to version 1, you could first check the commit id and then use Git Checkout to recover it.

You could use either of the following way to find the commit id.

```
git log -S"version 1" -- testYQ.txt
git log -L1,1:testYQ.txt --no-patch
```

The first command searches for commits that mention the text "version 1" either in the commit messages or within the content of files changed by those commits.

The second command identifies who edited line 1 of the specified file by showing the history of changes affecting that particular line.

After identifying the commit ID of the version you want to revert to, you can type:

```
git checkout 415b318 -- testYQ.txt
```

Besides Git Checkout, you can also use Git Restore.

```
git restore --source=415b318 testYQ.txt
git add .
```

There is a subtle difference: git checkout not only restores the file to a previous state but also stages it (adds it to the index), whereas git restore restores the file in the working directory without automatically staging it.

Do not forget to git commit to update the local respository, with an appropriate message.

```
git status
git commit -m "revert to version 1"
```

Before the end of Activity 1, let us sync our local repository with the cloud by git pushing the changes to Github.

We start with git pull to ensure our local copy is up to date. This is important because collaborators may have pushed changes concurrently. After synchronising, we proceed with git push to upload our changes.

```
git pull origin main
git push origin main
```

Don't worry if you see an error message — this is expected. Since only one person can successfully git push at a time, it's normal to encounter a conflict. Just run the two commands again to retry.

3 ICA2: Branching, Creating & Resolving a Merge Conflict

We will cover the following things:

1. how branch works;
2. how git merge works;
3. how merge conflicts happen and how to resolve them.

3.1 Branching

We first check how many branches do we have right now.

```
git branch
git branch -vv
```

To create and then checkout a new branch, say, featureA, there are three ways.

Let us start with:

```
git branch featureA
git branch -vv
```

We have successfully created a new branch named featureA. Note that the current HEAD is still pointing to the main branch. To switch to the new branch, we need to:

```
git checkout featureA
git branch -vv
```

See how the color highlighted part changes and how * shifts.

Before proceeding, let us first learn how to delete a branch.

```
git checkout main
git branch -d featureA
git branch -vv
```

There is one branch left, main.

The next two lines of code do the same thing, create a new branch and switch to it. Choose one to run.

```
git checkout -b featureA
git branch -vv
```

```
git switch -c featureA
git branch -vv
```

3.2 Git Merge without a conflict

Let us first practice a merge case locally without conflict.

Before continuing, make sure you are currently on the featureA branch.

```
git branch -vv
```

The following activity simulates collaboration between two engineers — one working on the featureA branch and the other on the main branch. Of course, in this simulation, we will perform the work for both roles ourselves.

Imagine that Engineer Jack is assigned to the featureA branch. He begins by creating a new file named featureA.txt.

```
echo code for featureA > featureA.txt
git add .
git commit -m "Craft new code for feature A"
```

```
git log --graph --all
```

What do you observe here?

Now, we switch back to the main branch:

```
git checkout main
```

On the same time, the other engieer, Rose, created another new file named bugs.txt. (You might happen to notice “featureA.txt” is gone!)

She was trying to fix some bugs in the version 2 (current staged version in the main branch), but she chose to give some suggestions in the bugs.txt first.

```
echo suggestions for version 1 > bugs.txt
git add .
git commit -m "suggestions for version 1"
```

Now, let us view the commit history tree.

```
git log --graph --all
```

You could see how the tree branches here.

Git Merge

Suppose Jack is confident that the implementation of Feature A is complete. He now wishes to merge the featureA branch back into the main branch.

Note that this scenario is a over-simplified example intended to illustrate how git merge works. It does not fully reflect how things typically happen in real-world development workflows.

```
git checkout main
git merge featureA -m "Update code feature A to main"
git log --graph --all
```

You can see how the commit history tree displays such a merge without a conflict.

Did anything else catch your eye?

3.3 Git Merge with a conflict

Merge conflicts arise when two members of a team edit the same file on their separate local repositories. The first person to push their changes to the shared remote repository will not have any problems, but the second person will encounter a merge conflict.

Next, we will practice a local Git merge scenario that results in a conflict.

We'll continue with the same setup, where Jack is working on the featureA branch, and Rose is working on the main branch.

To deliberately create a conflict, we will have both Jack and Rose edit the same file — testYQ.txt — in different ways on their respective branches.

When we attempt to merge featureA into main, Git will detect incompatible changes in testYQ.txt, triggering a merge conflict that we will need to resolve manually.

First, Rose edited the code of testYQ.txt, and call it version 3.

```
git checkout main
echo version 3 by Rose > testYQ.txt
git add .
git commit -m "version 3 by Rose"
```

Jack was also working on testYQ.txt, and he didn't know the recent commit by Rose. He did the following:

```
git checkout featureA
echo version 3 by Jack > testYQ.txt
git add .
git commit -m "version 3 by Jack"
```

Now, we switch back to the main branch and try to merge featureA to main:

```
git log --graph --all
git checkout main
git merge featureA -m "update version 3 by Jack"
```

A merge conflict occurs, with Git outputting:

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

Manually Resolve the Conflict

Open `testYQ.txt`, you'll see something like:

```
<<<<<<< HEAD
version 3 by Rose
=======
version 3 by Jack
>>>>>>> featureA
```

You can edit it freely by picking one version or combining both:

```
version 4 combining Jack and Rose's work
```

Save it and then resolve by :

```
git add .
git commit -m "Resolved merge conflict"
```

Now, let us view the commit history tree.

```
git log --graph --all
```

You can see how the commit history tree displays such a merge with a conflict.

Summary of Git Merge

Git merge is a safe and non-destructive operation that integrates changes from different branches by preserving the complete project history. It creates explicit merge commits, allowing collaborators to trace the origin and evolution of changes clearly. Conflict resolution occurs during the merge process, where any overlapping changes are addressed. Importantly, git merge does not rewrite existing commits, ensuring that the original commit history remains intact — a preferred approach for shared or collaborative workflows.

Analogy:

Merge is like combining two roads into a junction — both paths are preserved, and you can trace where each came from. It's useful when teamwork transparency is more important than a clean-looking history.

Clean up before Git Commit

For simplicity, we delete featureA branch after merging.

```
git branch -d featureA
```

```
git rm featureA.txt
git rm bugs.txt
git add .
git commit -m "remove two files"
git push origin
```

3.4 Recommended Student Pair Exercise: Practicing Merge Conflicts

1. **Pair Up and Set Up**
   Form pairs and fork or clone the same Git repository to ensure both students start with identical codebases. Each student should first run `git pull` on the main branch before beginning any work.

2. **Make Conflicting Edits**
   Both students open the same file and deliberately modify the same line differently to create a scenario that will cause a merge conflict.
3. **Commit Changes Separately**
   Each student commits and stages their changes independently.
4. **Push and Resolve Conflicts**
   - Student A git pushes first successfully.
   - When Student B attempts to push afterwards, they will encounter merge errors due to conflicts. Student B then manually resolves these conflicts before completing the push.
   - Repeat the process with Student B pushing first and Student A pushing second, requiring Student A to resolve any resulting conflicts.

Show this in your terminal to visualize history:

```
git log --oneline --graph --all --decorate
```

3.5 Optional Activity: Understanding and Using `.gitignore`

Overview

The `.gitignore` file tells Git which files or directories to intentionally ignore and **not track** in your repository. This is useful for excluding temporary files, logs, build artifacts, or sensitive information.

Key Concepts

- A trailing slash `/` indicates a directory.
- Wildcards like `*` match any string except the directory separator `/`.
- Patterns can be restricted to specific directories by starting with `/`.

Exercise

1. Create a new file called `secret.txt` in your repository folder.
2. Create a `.gitignore` file in the same folder with this content:

secret.txt

3. Check the Git status:

Observe whether secret.txt is staged for commit.

4. Try to git add all files:
5. Check Git status again:

3.6 Optional: Pull Requests

Branches should be created for new features. Use them together with Pull Requests in your project work. Pull requests are a way for you to get your teammates to test out your code and offer suggestions. Pull requests can be initiated through the GitHub web interface.

4 Summary

Essential Commands

- `git add` — Stage changes
- `git commit` — Save snapshots

- `git push` — Upload to remote
- `git branch` — Create branches
- `git merge` — Combine branches

Best Practices

- write clear, descriptive commit messages
- create branches for new features
- Commit small, logical changes frequently
- always pull before pushing

5 References

1. **Official Git Website**
   - This page contains the official reference manual and tutorials. It is the comprehensive reference for Git: https://git-scm.com/.
2. **Cheat Sheet from GitHub**
   - This page contains a cheat sheet (from GitHub) for common commands: https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf.
3. **Visual Cheat Sheet**
   - *Note*: The original document references a visual cheat sheet, but no URL is provided. You may refer to resources like https://www.atlassian.com/git/tutorials/git-cheat-sheet.
4. **Passwordless Connection to GitHub**
   - Instructions for setting up SSH keys: https://docs.github.com/en/authentication/connecting-to-github-with-ssh.
5. https://www.youtube.com/watch?v=Q1kHG842HoI