Lecture Notes: Stashing and Undoing Previous Commits

4.1 Introduction to Stashing

Stashing

Often, we find ourselves in situations wherein we are working on one project and a need to work on some other code or feature suddenly arises. In such cases, we cannot commit our unfinished work, which still needs working upon. Instead, it could be useful if we had a temporary buffer where we could keep our changes and the in-progress work so that we can go back to it whenever we are ready to resume. Stashing is generally used in such situations. Let us try to understand stashing through these points:

- Stashing is used to store unfinished work and continue with some other functionality code.
- It basically deals with the dirty state of our working directory.
- You can stash both untracked and staged files, as well as .gitignore files.
- These files can be later unstashed to resume working from the point where you left off.

Therefore, stashing plays quite an important part for any developer to deal with work in progress. Now, we will see some specific use cases when storing our work as a stash comes in quite handy and is advantageous. Here are they:

- When you do not want to lose your progress in case of bug fixing
    - For example, if you are working on a feature and there is a need to fix some urgent bug in a previously released version of your code
- When you do not want to lose the local changes in your working tree while performing a hard reset, which basically resets the index along with the working directory. You can, thus, store your changes temporarily in the stash and then perform the reset over the repository to get back to a clean state to start the work from the beginning.
- When changes do not need to be part of the commit history

.gitignore File

Whenever you work on a project, you basically deal with two types of files. One is the project files, which need to be tracked so you can have a record of the changes made in the project and can also obtain the different versions of files. You can easily return to a previous snapshot of your file later if you need to . Similarly, there are

other files that you do not want to track, such as log files. These files are just random files, which are generated at run time. Nevertheless, they do not need to be tracked as you don't need to maintain versions for them. So, how do you tell Git to ignore these files? In such cases, .gitignore files are used.

.gitignore is a simple plain text file that contains the names of private and secret files. The files mentioned in the .gitignore file always remain untracked by Git. .gitignore is generally kept in the root folder of a directory. You can write simple names, for example, file1, or specify patterns, such as *.log, as well.

Note: The files mentioned in the .gitignore file remain untracked by Git. However, .gitignore can itself be tracked by Git. Therefore, if you don't want to commit the .gitignore files in the repository, then you can stash them using the **git stash -a** command.

You have now learnt what a .gitignore file is. Next, you will see which files should be ignored by Git. Here is a list of these files:

- Build artefacts and system-generated files should be ignored because they can be derived from the source code.
- Files containing credentials or any kind on secret information should be kept in the .gitignore files as they hold confidential data.
- Files generated during runtime, such as log files, dist folders, etc. should be ignored by Git.
- Operating system files should also be kept in the .gitignore files.
- Compiled code, for example, the .class file, and so on should also be ignored by Git.

These were some examples of files that should be ignored by Git. A point to remember here is that a file tracked already by Git cannot be ignored by it. You first need to delete that file from the tracked region and then add it in the .gitignore file.

4.2 Different Stashing Commands

Stashing is used to save your unfinished work whenever you need to work on some other important task, for example, bug fixing. To deal with these stashes, there are different commands. You will take a look at each of them now:

- **git stash:** This command is basically used to save or stash your changes present in the staging and modified sections. Nevertheless, it does not work on untracked and ignored files. The general syntax to apply this command is **git stash**.

- **git stash -u:** This command, by default, works only on the files that are either present in the staging area or have been modified and are being tracked already. To stash files that are untracked, you should use the -u option with the git stash command. This command instructs Git to stash the untracked files along with the modified and staged files. The general syntax to apply this command is **git stash -u**.

- **git stash -a:** The -a flag instructs Git to stash not only untracked and modified files, but also the ignored files. Therefore, all the files present in the staged and untracked sections, as well as the .gitignore files, are saved and stored to work upon later. The general syntax to apply this command is **git stash -a**.

- **git stash list:** This command is used to see a list of the stashes that have been stored to work upon later. The general syntax to apply this command is **git stash list**.

- **git stash show:** This command is used to see the changes stored in the stash. It shows which files have been changed in the particular stash that is specified. The general syntax to apply this command is
git stash show stash@{stash_id}.

- **git stash pop:** This command is used to apply back the stored stash and then remove it. By default, git stash pop will apply back the latest stash stored and then remove it. To apply any arbitrary stash, the general syntax is
git stash pop @stash{stash_id}.

- **git stash apply:** This command also works the same as **git stash pop**. The only difference is that **git stash apply** does not remove the stash from a stash list. To remove a stash from the list, you need to use the **git stash drop** command. To apply git stash apply on an arbitrary stash, the general syntax is
git stash apply @stash{stash_id}.

- **git stash branch:** This command creates a new branch and applies back the selected stashed changes over there. These changes can later be staged and committed over that branch. The general syntax to apply the git stash branch command is
git stash branch <branchname> stash@{stash_id}.

- **git stash drop:** This command is used to delete a stash that you no longer need. By default, git stash drop would delete the most recent stash if stash id is not specified. The general syntax to delete any arbitrary stash is **git stash drop stash@{stash_id}**.

- **git stash clear:** This command is used to remove all the stashes at once. All the stashes are deleted at once after the execution of this command. The general syntax to apply this command is **git stash clear**.

4.3 Different Undoing Commands

You often encounter situations wherein you are working on a project and have reached a disordered state. In such situations, you need to go back to the previous state when the condition of your repository was stable to again start from a fresh point. Let's take a look at some cases wherein you need to revert to a previous state:

- You began experimenting with Git but now want to restore the previous version.
- There are several bugs in the project and so, you want to restore to the previous snapshot of your project.
- You have staged your changes, but later realise that they need to be reverted.

There are different commands available to undo your work. We will now take a look at each of them and see how they work. Here are the commands:

- **git reset:** This command is used to reset the state of your working directory to some other state as specified. The general syntax to apply the git reset command is **git reset <mode>**. There are three modes that can be applied with the git reset command. These include the following:
    - ❏ **--soft:** This mode moves the head to the state of the specified commit such that the files are shown as present in the staging area.
    - ❏ **-- mixed:** This mode moves the head to point to the state of the specified commit such that the files are shown as present in the untracked region just before the commit took place.
    - ❏ **--hard:** This mode resets everything, including the working directory. Data loss also occurs in the case of a hard reset.
- **git revert:** This command works not by rewriting commits. It creates a new commit such that the content of the history is reversed. The advantage of using this command is that the history is never lost. The general syntax to apply this command is **git revert <commit_id>**.

Note: git reset should never be applied on public branches. This is because if you perform a reset on the commits that you have already pushed to the public repository, then it becomes difficult for other developers to sync their progress with the public repository as they are still working on the older version. Therefore, git revert should be used in these cases, as it undoes tasks by creating new commits. Therefore, this commit can be pushed to remote from where other team members can pull it.

- **git reflog:** This command keeps a record of the tasks that you perform in Git. It keeps track of the commits that the head has pointed earlier and checks out to another branch.

- **git rm:** This command is used to delete files from the Git repository. It is another way of telling Git that it no longer needs to track a particular file. The general syntax to execute this command is **git rm <file_name>**.

- **git mv:** This command is used to rename a file that is already being tracked by Git. The general syntax to apply this command is **git mv <old_name> <new_name>**.