# Introduction to Git and GitHub

Arbel Tepper

These are some of my takeaways from Google's Introduction to Git and GitHub course on Coursera.

## 1 Useful Commands

### 1.1 Git Workflow

- `git init` - Initializes a new repository in the current folder.
- `git config user.name "[name]"` - Associate commits with a person.
  - `git config user.name "[name]"` - Add the email of the person.
  - `git config --global credential.helper cache` - Saves the GitHub login details for 15 minutes, so that you wouldn't need to repeat them at every `push` and `pull` commands.
- `git add [file]` - adds the file to the staging area to be tracked.
- `git commit` - Takes a snapshot of the files at the staging area and saves them as a `commit`.
  - `git commit -m '[message]'` - Adds the message to the commit.
  - `git commit -a` - Skips the addition to the staging area step.
- `git diff` - Helps to see the differences between files.
- `git rm` - Removes files and stages the changes to commit.
- `git mv` - Changes a file's name or moves a file to another directory.
- `git log` - Shows the latest commits.
  - `git log -p` - Shows the changes made in the committed files.
  - `git log -5` - Shows the last 5 commits.
  - `git log --graph --oneline` - Graphically shows the commit history as a line.
  - `git log --graph --oneline --all` - Does the same but for all branches.

## 1.2 Files

- `.gitignore` - A file that instructs Git to ignore certain files in the directory.
  - `*.o` - ignore all compiled files.
  - `*.log` - ignore all log files.
  - `.Ds_Store` - ignore operation system files.

## 1.3 Undoing Changes

- `git checkout --[file]` - Discard changes to a file in the working directory.
- `git commit --amend` - This replaces the last commit by a new commit of both the changes in the old commit and the changes currently in the staging area.
- `git revert [commit_id]` - Takes a specified commit and creates a new commit which inverses the specified commit. It does **not** change the commit history.

## 1.4 Branches

- `git branch` - Shows all of the existing branches.[1]
  - `git branch [name]` - Creates a new branch by that name.
- `git checkout [branch]` - Switches to the specified branch.[2]
  - `git checkout -b [branch]` - Creates a new branch by that name and switches to it.
  - `git checkout -d [branch]` - Deletes the specified branch.
- `git merge [branch]` - Merges the specified branch into the current branch and creates a commit.
  - `git merge --abort` - Resets the merged files in the working tree to how they were before.
    This command can only be used after a merge conflict.[3]
- `git rebase [branch]` - Merges the specified branch into the current branch by copying every commit as if it was made on the current branch. It keeps the commit history linear which helps with debugging.

[1] A BRANCH is a pointer to a certain commit.
[2] The `checkout [branch]` command moves the HEAD POINTER to that branch.
[3] A MERGE CONFLICT happens when there is an overlap between 2 changes made in different branches which are set up to be merged. For example; editing the same line of code in different branches. The conflict has to be addressed and fixed manually by editing the relevant files.

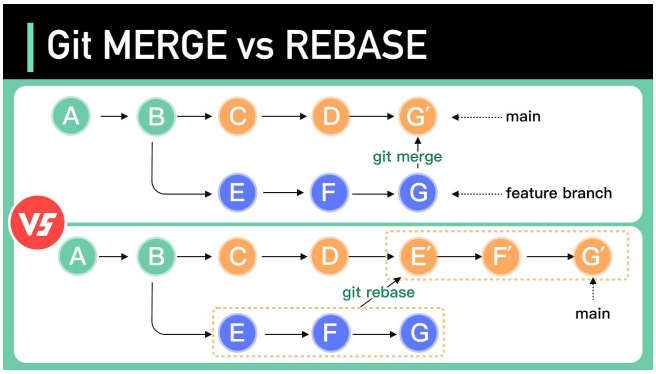


Figure 1: Merge vs. Rebase

- Commits squashing
  - `git rebase -i [branch]` - Opens the Rebase File with the branch commits. This file shows the command, commit hash and commit message.
  - `pick -> squash` - Melds the `squash` labeled commits into the `pick` labeled commit, and creates a new commit for the meld.
  - Use `merge` or `rebase` to add the changes to the relevant branch.

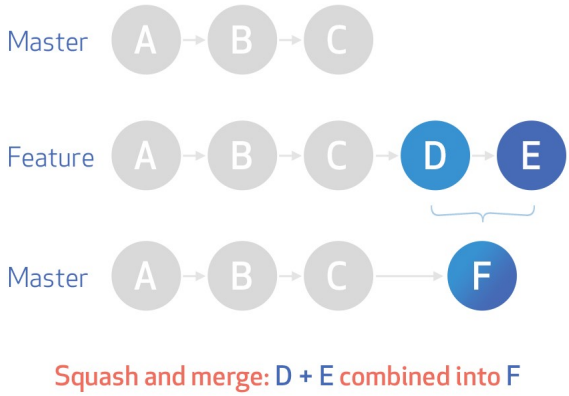


Figure 2: Squash

## 1.5 Working With Remote Repositories

- `git clone [repo_url]` - Creates a local copy of a remote repository.[4] [5].
- `git push` - Updates the remote repository will all the changes made in the local copy.
  - `git push -f` - Forces the push, in cases warning are raised.[6]

[4] Other remote repository hosting websites include BitBucket and GitLab.
[5] Cloning a private repository from GitHub cannot be done with a password anymore, instead a Personal Access Token is used.
[6] A forced push is fine for pull requests branches - because nobody else should have cloned the repository, but not for public repositories that someone might be working on.

  - `git push -u origin [branch]` - Creates the corresponding remote branch and pushes the branch to the public repository. The flag `-u` is needed for the first time only.

- `git pull` - Downloads changes from the remote GitHub repository to the local copy and merges them.
- `git fetch` - Downloads changes from the remote GitHub repository to the local copy but does **not** merge them. This allows us to take a closer look at the added branches or the commit history of the updated remote repository. The changes can be merged later on.
- `git remote show origin` - Shows information about the remote repository, including if the local copy is out of date.
- `git remote -v` - See the remote repositories you can interact with.
- `git remote add [repo_name][repo_url]` - Add a remote repository to interact with.[7]

# 2 Best Practices for Collaboration

- Always add meaningful commit messages.
  - It gives your collaborators more context on why you made the change and can help them understand how to solve conflicts when necessary.
- Always synchronize your branches with the remote repository before startientailng any work on your local machine.
  - This minimizes the chance for a merge conflict or for the need to re-base.
- Avoid committing changes directly to the master branch.
- Avoid heaving very large changes that modify a lot of different things. Split unrelated changes to different commits.
  - For example; renaming a variable in a file should not be in the same commit as adding a function.
- Regularly merge changes made on the master branch back onto the feature branch.
  - This way we end end up fewer merge conflicts when the time of the final merge comes around.
- When working on a big change or addition, it makes sense to create a separate feature branch for it.
- When maintaining a few versions of the same project have the latest version of the project in the master branch and the stable version on a separate branch.

[7] Adding the origin repository (commonly as "upstream") is useful for example when working on a fork of the original repository. The fork is not updated when a commit is made to the original repository, but we might want to work on the most recent version of the project. This allows us to fetch and pull the latest commits from the origin.

# 3 Creating a Pull Request

1. Fork the repository that you want to suggest a pull request to.
2. Clone the fork to the local machine.
3. Create a new branch and address the issue.
4. Commit the changes and push the branch (without merging) to the forked repository
   (a) If the pull request corrects a GitHub issue the commit message should include "`closes #[issue_number]`"
5. Go to your forked repository in GitHub (choosing the relevant branch) and click on "open pull request".
6. Write a detailed message explaining why a change is necessary and how your code achieves it.