Git diff

- git diff filename -

Forking

- A GitHub fork is a copy of your repository. Forking a repository allows you to freely experiment with changes without affecting the original project. Forking copies all files at their current version, along with all commits up to that point. Forking does not affect the original repo, this is just your own copy.

Git Log

- Shows all commits that have been made into the branch.
- Use git log –oneline to show sha and first line of commit message.
- Use –decorate to display references (branches, tags, etc) that point to each commit. Use

  git log –oneline –decorate for example to show on one line.

-

Cloning

- A repository on GitHub is called a remote. You can clone your remote to your local computer, and then sync the two up. After cloning, you can add/edit files, and push and pull updates. This is a one time process, like git init. Git makes no distinction between working and central repo.

Commit and File Details

- git show shaID – Shows you the commit message, with the diff itself. Also see changes made in the files. This is essentially a combination of git log and git diff, however you can show a singular commit.
- git show HEAD – Shows the most recent commit.
- git show HEAD~3 – Show 3 commits before head commit. Just replace the number.
- git diff HEAD~1 HEAD~4 is a valid command. Don't need sha's anymore.
- git annotate – Shows git log but for a particular file. You can see changes line by line in the file.

Undoing Changes and Reverting Commits

- **git checkout --filename** - Undo's the changes in a file if not staged. If the file is staged, unstage the changes with **git reset HEAD filename**. Then execute the git checkout command and you will undo the changes.

- **git checkout commitHash filename** - This will take you to another commit. Say you committed and pushed changes to a remote, but the code doesn't work. You just checkout a prior commit, and the changes in the most recent commit can be corrected.

Best Practices

- You should always have a working version of your project.
- You have Master, Test, and Dev branch normally.
- The dev branch merges into the test branch, and the test into the master. Features will be in the dev branch.
- When a branch is created, the new branch contains all the commit history of the original branch up to that point.
- GitHub calls merges "pull requests". They're used for repositories and branches.

Branches

- You can create a branch by the command: **git branch branchName**
- **git branch** will list all branches that exist locally.
- Switch branches, you do **git checkout branchName**
- You can create and switch to the new branch with: **git checkout -b branchName**

gitIgnore

- If you have a file that you don't want to be public, use gitignore. For example **/filename.extension** will not be included by git to be tracked.
- To ignore a folder, put forward slash first: **/folderName**
- To ignore files with a certain extension in a folder, say txt files, say **folderName/*.txt**

Sync Branches

- If a branch exists locally, and not remotely, just push it up, and the remote will now have that branch.
- Say you create a branch called "test" in remote. You can create a new branch called test locally, and sync the local and remote branches with: **git checkout --track origin/test**
- You can also **git fetch** and then **git checkout test** if test is an existing branch in remote. Either works.

Graph Branches in Terminal

- **git log --graph** will create a graphical representation of the branch you're in

Merging Branches

- In order to merge branches, use: **git merge branch-to-be-merged destination-branch** Make sure you get the order correct.
- When branch-to-be-merged has been merged, you can delete it with: **git branch -d branch-name**

Merge Conflicts/Merges

- When merges don't happen automatically. For example, if same file has different code on the same line in two different branches is merged.
- Fast forward merges and recursive merges. Merge conflicts only happen with recursive merges.

  You can execute : **git merge --abort** to abort a merge.

- Remove the headers that git adds when merge conflict occurs, and then decide which code stays, and which goes, in the file(s) that have the conflict. Then add the file(s) and commit to finalize the merge.
- Pull and push commands in git are actually merges.
- On GitHub, Pull request can result in conflicts. Usually the contributor, not the owner of the repo, will resolve the conflict.

Tags

- **git tag -a <tag name, like v1.0> - m "message"** - This will create a tag for you, at a specific point in time. The tag is assigned to the most recent commit**.**
- **git show <tag name>** - This will show you information about the specified tag, like who tagged it, date, message, commit sha, etc.
- **git log --pretty=oneline** - Shows all commits in one line.
- You can tag a commit prior to the most recent commit. Use git-log to get the sha of a commit, and use **git tag -a <tag name> <commit sha> -m "message"**.
- You push the tags to the remote using **git push origin --tags**.
- In GitHub, releases correspond to tags.

GitHub Collaboration

- If you are not owner of a repo in GitHub, you can't just push to a repo. You will have to utilize Pull Requests, which are apporved by the owners. However, the owners can add you as Collaborators in the Settings tab, which will allow you to push.

Protecting Branch

- In Settings, go to Branches. Normally, you want to keep master as your default branch. In Branch Protection Rules, you can specify rules about branches, such as how many reviews required before merging.
- In the “Files Changed” tab, is where you review changes. This is the official review when a branch is protected. You can request a certain number of reviewers review a change, before a merge can actually happen.

Issues

- You can add issues in the issues tab, and assign assignees and labels.

README

- Just add a header, and a description of what your project is about in the repository. Add instructions in README as well. Have a header, description, installation guide, usage, and how to contribute is a good template.
- Add topics tags so that people can search for your repo.

Important Templates

- Contributing files are usually put in a docs folder.
- Create a CONTRIBUTING.md file.
- Have a header, thank users for contributing, add a code of conduct, How can you contribute, Using other endpoints, new features, report bugs, and how to submit a good bug report.
- In GitHub, in the settings, go down to Issues. You can create templates from there.
- After the owner creates these templates, GitHub gives users the templates to use.
- For Pull Requests, create PULL_REQUEST_TEMPLATE.md
- For this, as to reference a related issue, description of changes, and mentions.

Git Log

- Use **git shortlog** to see all contributors for the repo.
- **git log --author=”name”** - Filter on commits on specific author.
- **git log --grep=#2** - Filter on commits with given parameter.
- **git log --grep=#2 --author=Malam620** - Now add author as additional condition
- **git log --grep=#2 --author=Malam620 --since=1.hour**

Contribution Flow

- Make sure to update your local repository with the master repository. You can use Compare to do this. This is because maybe another contributor made changes to the original repository, so you want your forked repository to be up do date.

Naming of Feature Branches

- Don't commit to master.
- There are naming conventions for how to name feature branches, bug branches, etc. For example, if you have a tag for a bug called bug #1345, create a branch called: bug-1345.

Commit Messages

- Make them descriptive and concise.
- Limit message to 50 characters. Capitalize the message. Don't end the message with a period. Use body to add details. In the body, explain what/why not how.

Markdown

- You can write issues, pull requests, comments, templates, README, CONTRIBUTING files with markdown.
- You can format text in different ways, such as add headers, add images, add tasks using markdown.

Format Text With Markdown

- Use ** to bold, or __ to bold text. These are two underscores and two asterisks. For example: **This text is bold** or __This text is bold__
- Use single underscore and single asterisks for italics.
- > "This test shows up as quotes"
- ~~ or the two tildas will cross out text.
- Use # for headers. Add one # for header one, ## for header 2, and so on. Goes up to 6 headers. Also adds a link to navigate to the specific area of the file with a header.

Creating Lists with Markdown

- You can create a list like this:
  - this is the first element
  - this is the second element
    this will still be part of this list above

- In order to “get out of the list”, just hit enter twice, instead of one. On the second line, if you start typing, you are out of the previous list.
- You can also use + instead of - to create a list.
- Instead of two spaces, you can break lists by mixing and matching - and +
- These lists are unordered.
- You can create ordered lists as:
    1. First element

        1. This creates a sublist. You need to add three space for sublist.

    2. Second element
    3. Third Element
- Now, you can add task lists as:
    - [ ] task 1
    - [ ] task 2

- You can also use + and * above for the lists.

Tables in Markdown

Android | iOS | Windows

--- | --- | ---

Medium App for Android | Medium App for iOS | Medium App for Windows

- For the - added, you only need 3, but you can add more if you want.
- In order to align text a certain way, add a colon before --- for left, after for right, and on both sides for center alignment.

:--- | :---: |---:

- In order to add links, do []()
- Inside the [] add the text you want displayed, and inside the () add the actual link. Don’t have any spaces between the brackets and parentheses.

Using Images

- For images, very similar syntax as links:
- You can use links from the internet, or just upload your own images in your GitHub and use those.
- In order to add a video: [](url of image)

Writing CodeBlocks

- Use ` to write code. You can specify any language you want. Open and close with three ` and close with three ` as well.

```` Java
public static void main(String [] args){


   System.out.println("Hello World!");
}
```