Lecture Notes: GitHub Additionals

1.1 Contributing To A Project

Have you ever thought about how developers from all around the world contribute to open source projects? Open source projects are basically the projects that are free to be used, modified and distributed under public license. Making contributions to open source projects always plays a major role in learning more about social coding on GitHub.

The open source community provides a great opportunity to beginners and aspiring programmers to improve their skills and learn more by contributing to open source projects. There are various advantages of contributing to open source projects, some of which are as follows:

- Their code writing skills are enhanced.
- Their chances of being noticed as potential developers are increased.
- The bugs in the code easily get fixed as the code is open to all developers. Therefore, developers can easily raise a pull request.

So, from the above discussion, we can say open source contributions not only increase the efficiency but also the coordination and the collaboration among the developers. There are two major terms involved in open source contributions, which are as follows:

1. Forking
2. Pull request

Let's understand each of these terms in detail.

Forking

Let's understand forking using the following points:

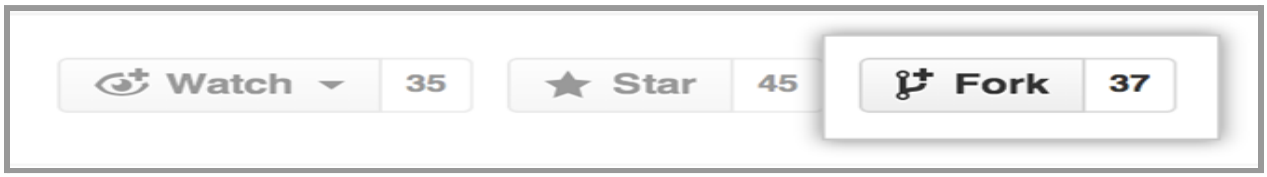
- It is the first step whenever you want to contribute to any project
- It is used to create a personal copy of someone else's project.
- It basically acts as a bridge between the original copy and teh personal copy.
- The changes made inside the forked repository are not reflected in the original one.

How to fork a repository

Whenever you want to fork a repository, you need to perform the following steps:

1. Sign in into your github account.

2. Search for the repository that you would like to fork.
3. After finding the desired repository, click on the fork button on the top right corner, as shown below.



Now that you have successfully forked a repository in your account, let's see how to work locally on the forked repository.

Working locally on a forked repository

1. The repository forked by you exists only over GitHub.
2. You need to clone the project to locally work on it.
3. Go ahead and make some changes using any code editor.
4. Then, commit the changes locally over your system.

So far, you have seen how to fork a repository and how to make the changes locally. Now let's see how to push these changes over to GitHub and create a pull request.

Pull requests

Till now, the changes you have made exist only in your repository. In order to reflect these changes into the remote repository, pull requests are raised. So, let's understand pull requests using the following points:

- A pull request is the last step of adding the proposed changes.
- Always remember the following points before making the pull request:
  - ❏ The contribution must be of good quality.
  - ❏ It can be small but should add value to the project
  - ❏ The acceptance of the proposed change is never a guarantee.
- After pushing the code, you will see a banner that indicates that you have recently pushed a new branch.
- Click on 'Compare & pull request'.

1.2 Tagging

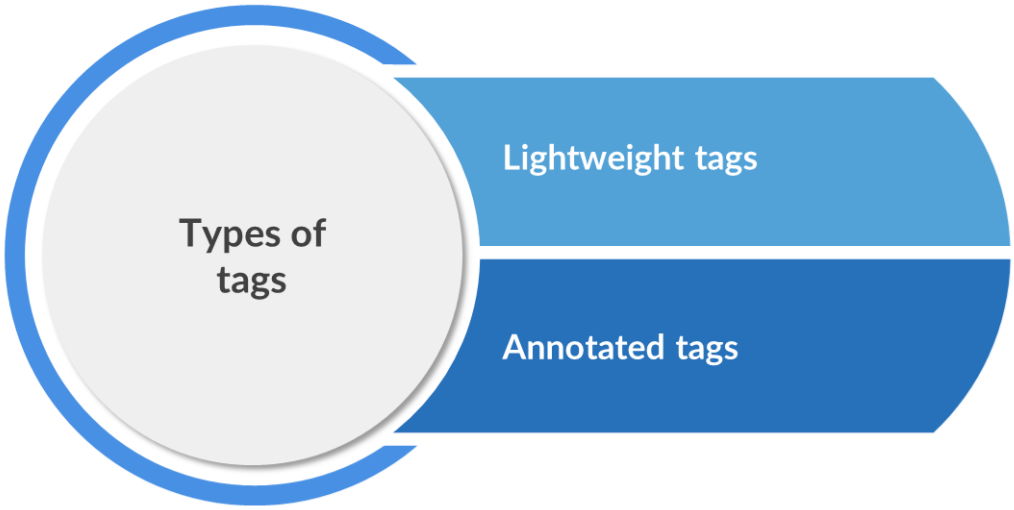
Generally, developers working on a project make a lot of commits. But, how are they able to mark any specific commit as a release version or any specific commit as a mark that some bug was fixed here? The answers to all these questions is 'tagging'. So let's understand tagging using the following points:

- A tag acts as a reference to a specific point in commit history.
- A tag is used to capture a point in the commit history that can be used as a release version.
- Different tags can be created for major and minor releases.
- Tags are also used to flag the merge points of the release and the master branch.
- Tags help keep a tab on the 'last known good-state' of the code in the case of a bug or a failure.

Types of tags

Basically, tags are of the following two types:

1. Lightweight tags
2. Annotated tags



Lightweight tags

Let's understand lightweight tags using the following points.

- One of the simplest methods to create tags is applying lightweight tags.
- A lightweight tag is just a pointer to a specific commit.
- It does not contain any extra information, such as tagger name, email or any message.

- The general syntax to create a lightweight tag is **'git tag tag_name'**.

Annotated tags

- Annotated tags are stored as a full object in the Git database.
- They contain complete information regarding the commit, such as tagger name, email and the message.
- Therefore, they are preferred over lightweight tags .
- The general syntax to create an annotated tag is **'git tag -a tag_name'**.

Note: Whenever you need to deal with a public repository, create annotated tags as they store extra metadata regarding the commit.

When to use tags

- Ideally, a tag should be applied at each merge.
- A tag is applied at a point where a particular branch merges with master.
- Complex projects can include various production and non-production tags.
- Tag comes in handy at release points.

Different Tagging Commands

Let's see some of the important tagging commands, as shown below:

- **git tag <tag_name>**: This command is used to create lightweight tags. For example, **git tag v1.0** will create a tag named 'v1.0'.

- **git tag <tag_name> <commit_id>:** The above command creates a tag for the top-most commit. However, if you need to tag any older commit, then the following command can be used.

- **git tag -a <tag_name>**: This command is used to create annotated tags in Git. However, running this command opens the text editor to give the message associated with the tag. Therefore, it is a good practice to use the -m flag. For example, **git tag -a v2.0 -m "Released Version 2.0"**.

- **git push origin <tag_name>**: This command is used to push a specific tag to the remote repository. For example, if you have created two tags, say, v1.0 and

v1.2, then, to push v1.2 over the remote repository, the following command will be used. For example, **git push origin v1.2**.

- **git push --tags**: This command is used when you want to push all the tags at once, but not step by step. For example, if you have created two tags, say, v1.0 and v1.2., then, to push all the tags at once over the remote repository, this command will be used. For example, **git push --tags**.

- **git tag -d <tag_name>**: This command is used to delete any specified tag in the repository once it has been pushed to the remote repository. For example, if you have a tag v1.0, then, to delete that tag, you can run the command **git tag -d v1.0**.

1.3 Interactive Staging

Interactive Staging is another variant of staging. Here, instead of staging all the files at once, we organise them into specific commits. For example, if you are working on a file named 'user.txt' and make changes to 12 lines of the file, then, when you stage it using **git add user.txt**, all the changes in the files are staged at once. However, if you want to stage only the first six lines for commit, then interactive staging can be used. Let's see the advantages of interactive staging as follows:

- It helps us to organise our changes into specific commits.
- Commits are logically separated from each other.
- It is useful when you want to stage only a certain part of the file.

Steps involved in interactive staging

1. Run the command 'git add -i' or 'git add --interactive'. After running this command, switch to the interactive mode.

```
Lenovo@Ashok_Pandey MINGW64 ~/Desktop/git (master)
$ git add --interactive

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

2. In the interactive mode, you will be provided with a list of commands. Now, in order to run any command, press the associated number or the first letter.

3. Once your required task is done, quit by just entering 7 and pressing 'enter' key.

Different options in interactive staging

- **status:** Status shows the state of a working directory, that is, the number of lines added or removed.
- **update:** Whenever you want to stage a complete file into the staging area, an update option can be used.
- **revert:** Revert command allows to reverse back the changes, that is, unstage the changes back to the modified section.
- **add untracked:** As the name suggests, this command is used to add untracked files in the staging area.
- **patch:** Patch command is used to deal with individual hunks, that is, it helps decide whether to commit their changes or not.
- **diff:** This command is used to see the difference between the two sections, head and index.
- **quit** and **help**, which are self-explanatory.