Chapter 5 Introduction to GitHub

Introduction to GitHub

GitHub is an online repository for source-code. Developers can use GitHub to easily organize projects as participants have one common version of a project to work with. Developers maintain this code on their local computer and use the Git version control system to maintain the online repository (repos). Git was developed by Linus Torvalds due to dissatisfaction with the previous *proprietary* system, BitKeeper.

Creating GitHub account and an repo

To store your program source code on GitHub you first need an account. This is easy, just surf to https://github.com/, click the green button 'Sign up' and fill in the necessary details. But do keep a note of your login name, email address and password. After you have an account you need to create a new repo. This is made easy in the main screen of your account, just click the green button '+ New Repository'. Then choose a name, such as *hackbook* and ensure the repo is public. Now click 'create repository'. After the repo was made you are shown a selection of options for adding local files on your computer to your online repo. This guide will continue to show you how it's done.

Installing and configuring Git

The first step you need to perform is to install Git version control system on your Linux machine. Type the following command into your Terminal.

```
sudo apt-get install git
```

After you have completed the installation use the following two commands to configure your local repository. Use the username and email address provided for the GitHub account.

```
git config --global user.name "username"
git config --global user.email "email address"
```

Now you need to create a directory on you Linux machine in which you can work on your program and upload it to GitHub. Create a directory called *hackbook* and navigate into it. Then use the init function to initialize it as a Git repo.

```
mkdir hackbook
cd hackbook
git init
```

The output will read as follows

```
Initialized empty Git repository in /home/user_name/project_name/.git/
```

The part in Italics depends on your own Linux machine and folder name. We are now set to create a file in the local repository and attempt to place it on the remote GitHub repo. First we need a file. Just create a small text file or even a Mark Down (.MD) and write something in it.

Updating local repository

Before files are send to GitHub they are first placed in a staging area locally on your computer. With the command *add* you add files to the staging area (index). Use a dot '.' to upload the entire folder or use Tab to autocomplete if you just want to upload a chosen file.

```
Git add .
```

With the *commit* command you can decide that after you have no alterations planned anymore to send the files to your official local repository. This split in local repo and working files is handy to prevent errors or in case you would change your mind. This comes at the cost of complexity.

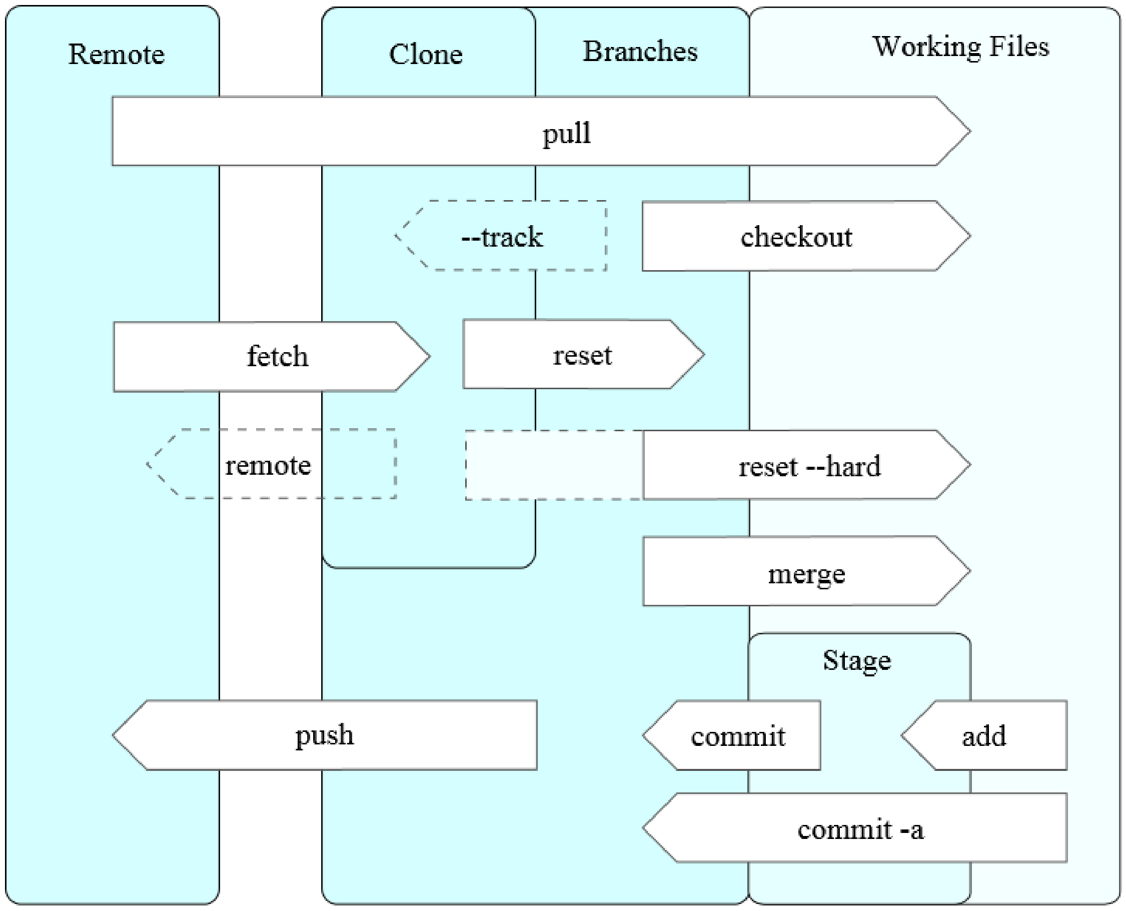

Figure 1. Git and GitHub model

Astute readers will note a problem. Sending out files to GitHub this way is easy if you are the only one maintaining the project. However, with more than just a few contributors the situation quickly becomes complicated as nobody is sure whose version of the software is current. That will be part of the second half of this tutorial. First we still need to send the files to the remote repo on GitHub.

Before you commit the files from you directory you can use the command *status* to see if you need to commit anything. It would be best to use the commit function often so your local repo is always up to date.

```
git status
```

This returns the following comment details

```
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   testHackBook.MD
```

You can see I have placed a Markdown file into the staging area. The comment above even suggests a method to unstage files. Now if you use the commit command and then the status command the situation has changed. Note how we add a small comment at the end of the commit command. This will become visible in GitHub and will make it easier to distinguish between recent updates.

```
git commit -m 'first commit'
git status
```

Now the following comment details are conveyed.

```
On branch master
nothing to commit, working directory clean
```

You have successfully created your own local repository which is distinct from the working folder. Now you still need to connect your local repository with that of the GitHub repo. One thing is certain; typing in the entire URL to the GitHub repo will soon become tiresome, not in the least because it is error prone. Luckily you only need to do it once per project with the following command.

```
git remote add origin https://github.com/username/projectname
```

The final command defined below will place what is in your local repo onto the GitHub repo. This command is final and can’t be withdrawn, so be sure what you upload! In short it states that the local ‘master’ will be pushed out to the remote ‘origin’ defined in the previous command sequence.

```
git push -u origin master
```

Input your GitHub username and password when prompted. If all has gone well your file has made its way to the GitHub website. Check it out! Whenever you have updated your project you can instantly place them on GitHub for everybody else to download.

On a side note, often we only want to upload source code to GitHub and not, for example, compiled code. With Git we can be selective of the files we add and commit to the stages between your working directory and the remote GitHub repo. With a .gitignore files we can exclude files with extensions of our choice. The first step is to create such a .gitignore file.

```
echo ".txt" > .gitignore
```

In this instance we do not commit files with the extension .txt. This is just to illustrate the possibility but perhaps you don't want to place files filled with notes on the development online.

If we were to use git status we would get a message that the file is untracked and has yet to be committed. With git add we can place it in the staging area.

```
git add .gitignore
```

From there on the procedure is similar to any other commit and push command cycle. If we were to create more code in our working directory next time and add files to the staging area those with the .txt extension will be ignored. Be careful, .gitignore, will be pushed to the remote repo as well. If you have forgotten about its existence you can be bug hunting for a while.

Downloading files with Git

Using the GitBash command line tool in Windows or just a Terminal in Linux it is very simple to download files from a repository. To obtain a copy of a Git repo use the following command sequence.

```
git clone /path/to/repository
```

If you're downloading the files from a website you need to add the full path of the website as a prefix to repository path. An example would be to download the hackbook code below would proceed as

```
git clone https://github.com/username/hackbook
```

Now you have downloaded the repository into the present folder you have navigated to. Navigate to that folder to see its content.

```
cd hackbook/
```

However, with clone you are merely making a copy of a file. It works similarly to wget. It does not update your local repo and sync it with the GitHub repo. With pull this can be achieved. In theory, if another developer had updated the GitHub repo your own repo will be changed to reflect that.

```
git pull https://github.com/username/hackbook
```

However, astute readers will notice a problem. What if the GitHub repo has been changed but so has you own local repo, and you don't want your changes to be erased. Well, we can use a two-step command. With fetch we get a copy of the GitHub repo and store it as local branch, not the local repo that you're actively working with.

```
git fetch https://github.com/username/hackbook
git merge
```

After we have reviewed the code we can use the command merge to put the two branches together.

```
git merge localbranch
```

Branching

This final command leads us to the most interesting feature of GitHub for professional users: branching. If you want to include a new feature into your project but keep a copy of the old one (the way it was) then you could use the branch command. This will set up a parallel development branch of your code that you can alter without fear of getting your code mixed up. The following step set up such a branch and we call it *anotherbranch*.

```
git branch anotherbranch
```

To go to this branch and work on the code use the checkout command as follows

```
git checkout anotherbranch
```

Now you are all set to make changes to the second branch. You can create as many branches as you want though I suspect it is usually no more than two: one master branch and one branch for testing code. So let's test out branching. After you have performed the two commands above create a new text file called test.txt and place some dummy sentences in to. Copy the file into your project folder. Now if we perform 'git status' we get the message there has been an untracked file. Next we add it to the staging area 'git add .' and again ask the status with 'git status'. As we are still working in the second branch called *anotherbranch* the file is committed to this branch and not the master branch. In essence the folder into which you copied the textfile can be seen as nothing more than a collection bag. To commit the file to *anotherbranch* use the command 'git commit –m "add textfile"' to commit the branch.

If you now use the checkout to switch to the master branch and use ls

```
git checkout master
ls
```

You will notice that the textfile was not added to the master branch

Merging and deleting branches

We can choose to merge branches if we want to. The method behind it is simple, go to the branch that you want to see over-written. In this case we want to see the master branch over-written by *anotherbranch* as it contains a file we have been working on. Use the following command to merge the two branches

```
git merge anotherbranch
```

If we use ‘ls’ we will see the file is added to the master branch. With ‘git status’ we see that this branch is said to be one commit ahead of origin/master that this found on GitHub. With ‘git push’ we can push this updated master branch to the remote repo if we want.

Finally we can choose to delete a branch. As an example we will delete the branch *anotherbranch* with the following command.

```
git branch –d anotherbranch
```

It is now removed

Conclusion

One thing that Git and GitHub do not do is keeping track of changes within a file. If you have changed code in a particular branch you will have to figure out whether or not there will be dependency issues. Git cannot do that for you. If you are working on a project that uses GitHub for code exchange it can become quickly tedious as you are trying to figure what changes have been made by others. The risk is that as soon as you figured that out and uploaded your new code somebody else will have done the same. Modularity of code and division of labor then become you’re only saving grace. If you find Git and GitHub still to be too daunting then try out a GUI client. For Windows there is GitHub (not to be confused with the website) while git-cola and SmartGit can be sued for Linux (and Windows).