Final Project

Introduction

Here, in the final project, I have tried to cover the major parts of the course especially those topics that are challenging and related to version control issues that are faced by developers and alike in a real-world situation. This will enable you to turn yourself into a true Git Pro.

Note: In the Final Project, you are expected to have understood all the concepts and implementation knowledge that have been imparted throughout the course. It is advisable to refer to the training if you get stuck somewhere.

Problem Statement

The entire project has been organized under a 10 Step Challenge.

These challenges will give you hands-on experience in building a Distributed Version Control System using Git & GitHub for source control management of a Software Application.

Doing this project, you will also get a good understanding of collaboration within a simulated team environment & also understand how to perform an integration test of the entire source control management system.

The following are the steps that are needed to be taken for this project.

- Step-1: **Setting up a local repository:**
    - For Windows users, you have to set up a local repository using the Bash terminal. For Mac and Linux users, you can use any of the terminals that your operating system offers preferably with a Bash shell.

- Step-2: **Setting up the remote repo and establishing a connection with the local repo:**
    - We will be using GitHub to host the remote repo. You will be working with a single GitHub account.
      Note: Only for Step-8, you would be needing two GitHub accounts for teamwork simulation.

- Step-3: **Establishing SSH communication between local and remote repo:**

    - Both SSH and HTTPS can be used to carry out all the steps but we preferred to use SSH in this project. SSH is more secure due to its passwordless nature, strong authentication, encryption, etc.

- Step-4: **Tackling Commit history:**
    - Commit history is one of the most important aspects of Git. Keep in mind that building a clean commit history goes to a great length in successfully managing your projects

- Step-5: **Diving deep into Git Branches (Part 1) - resolving merge conflicts:**
    - In this step, you need to create a separate branch and simulate a conflict situation with the master branch. You need to set up your merge tool beforehand to carry out this task.

- Step-6: **Diving deep into Git Branches (Part 2) - resolving to rebase conflicts:**
    - Rebase is sometimes misunderstood. This step will instill in you a solid knowledge about this important Git process. Like a merge conflict, you need to have a separate branch in order to simulate the conflicting requirements.

- Step-7: **Pull from GitHub with rebase option:**
    - This step takes local rebase to another level where you need to work with the upstream repo. It's assumed that you already populated the remote repo with all the artifacts.

- Step-8: **Team Collaboration in GitHub using a Pull Request:**
    - This is a truly interesting step. In the real world, nobody works in isolation, be it in the capacity of a developer in a corporate environment or as a freelancer. You need to have two GitHub accounts. The problem of a pull request can be solved in 2 ways where a team member may be a collaborator approved by the repo owner or a non-collaborator. In this project, we have chosen the latter option.

- Step-9: **Working with Git Stash:**
    - This step requires you to set the Git Stash list so that you effectively apply the stash-related sub-commands such as - list, show, apply, drop, pop, clear, etc. Here too, you need to create a separate branch to achieve a four-step process.

- Step-10: **Delving into Git Tags:**
    - Git tags help a version control engineer to highly organize his commits. You need to set up both lightweight and annotated tags. You need to create a .gitconfig file if you don't already have one in order to automate the pushing of annotated tags as default and automatic behavior.

Note:

- Information about the above steps has been spelled out in great detail below that makes every item actionable with a fresh experience of challenge and brainstorming.
- The yellow highlighted words which you will find in the steps below indicate the current git branch you are on.

Assumptions

- The student has already (if you haven't done the following, please do so)
  - installed a visual text editor as the default text editor for Git (mandatory)
  - installed p4merge as the default diff and merge tool for Git (mandatory)
- The terms “folder” and “directory” may be used interchangeably
- The terms “repo” and “repository” may be used interchangeably

Approach: Follow the steps and complete the assignment

1. **Step-1: Setting up a local repository:**
   a. Go to https://github.com/bibroy/downloads
   b. Click on the file link initializr-verekia-4.0.zip
   c. Click on the Download button to download the file.
   d. Once the file is downloaded, open your Git Bash terminal (if you are a Windows user) or any terminal where you can Bash commands
   e. CD (change directory) to the directory where you want to create your Git Repository for the Final Project. Let’s call this directory “git-fast” (your chosen directory can have a different name; we chose this name just for the purpose of explaining the project)
   f. (git-fast) Copy the downloaded file initializr-verekia-4.0.zip to “git-fast” directory
   g. (git-fast) Unzip the file initializr-verekia-4.0.zip
   h. (git-fast) Unzipping will create a directory called “initializr”
   i. (git-fast) Rename the directory “initializr” to “insh-final-project”
   j. (git-fast) CD (change directory) to the directory “insh-final-project”
   k. (insh-final-project) Initialize the repository with “git init” command
   l. (master) Now you are on the master branch of the repository. List all the files/directories including hidden files. Can you see the “.git” hidden directory? If yes, then follow the steps below:
   m. (master) Add all the files/directories to the staging area
   n. (master) Commit all the artifacts (files/directories). The commit message is “initial commit”. Let’s say, the hash is commit-1 (actually the commit has is an

alphanumeric number but we are using terms such as “commit-1” for the sake of simplicity)

o. (master) Run the git status command to see if the working directory is clean. If it’s clean, then follow the next step below:

2. **Step-2**: **Setting up the remote repo and establishing a connection with the local repo:**
   a. Go to github.com. Create a remote repo on GitHub. The remote repo name should be the same as the local repo i.e, “insh-final-project”. You may add README file, LICENSE, and .gitignore files
   b. Go to your Bash Terminal
   c. (master) Add a remote named “origin” for the remote repo
   d. (master) Now cross-check the creation of the remote added in the above step by running the appropriate git command
   e. (master) Push all the artifacts from the local repo to the remote repo
   f. (master) Now cross-check the creation of remote reference branch “origin” by running the appropriate git command
   g. Go to github.com and check if all of your artifacts got pushed from the local repo

3. **Step-3**: **Establishing SSH communication between local and remote repo:**
   a. Open your bash terminal. You are within the local repo
   b. (master) Generate the SSH keys using the appropriate bash command. Use the following specifications
      i. Key type: *RSA*
      ii. Key length: *2048 bits*
      iii. Avoid passphrase for the keys for the purpose of this project
   c. (master) Check if the keys have been generated successfully. Let’s assume the name of the private key is “**id_rsa**” and the public key is “**id_rsa.pub**”
   d. (master) Copy the public SSH key to the clipboard
   e. Go to github.com and create the SSH key
   f. (master) Go to your bash terminal and test the SSH connection.
      i. Are you behind a firewall? If yes run the appropriate command
      ii. If you are not behind a firewall, run the appropriate command
   g. If the test succeeds then proceed to the next step.
   h. (master) Now the default connection between remote and local repo is via HTTPS. Hence you need to switch from HTTPS to SSH. Run the appropriate git command to make the switch from HTTPS to SSH
   i. Run the appropriate git command to test if the protocol switch (HTTPS to SSH) worked correctly.
   j. (master) Make some changes in the humans.txt file and push to the remote repo to make a final test if all with SSH connection works good or not

4. **Step-4: Commit history in-depth**
   a. Let’s build up the commit history since we have very few commits now. You can choose your own commit messages.
      i. (master) Make two changes in robots.txt and commit them separately to have two additional commits.
      ii. (master) Make two changes in humans.txt and commit them separately to have two additional commits.

iii. (master) Create a branch called feature-branch-1 and checkout to the branch
iv. (feature-branch-1) Make two changes in browserconfig.xml and commit them separately to have two additional commits in this branch

b. (feature-branch-1) checkout to the master branch
c. (master) Run git log command with necessary options that do the following
   i. Decorates commit history with branch names
   ii. Prints a graph of branches and
   iii. Includes history of all branches
   iv. History is displayed in oneline per commit
d. (master) Run git log command with the right option that does the following - gives information about which files were altered as well as the number of lines added/deleted from each file (Hint: type "q" to quit if the information displayed is more than that can be accommodated in a single screen)
e. (master) Run git log command with the right option that provides patch information such as diff (Hint: type "q" to quit)
f. (master) Run git log command to display commit history for the directory "css" only
g. (master) Run git log command with the right option which allows searching to commit history that contains a particular query string. As an example, the solution to this project takes "initial" as the query string to search the commit-hash for the initial commit.

5. **Step-5: Diving deep into Git Branches (Part 1) - resolving *merge* conflicts**
   a. (master) Create a branch feature-branch-2 and checkout to the same
   b. (feature-branch-2) Make three simple changes in of the three lines of index.html
   c. (feature-branch-2) commit your changes
   d. (feature-branch-2) checkout to the master branch
   e. (master) Make 3 different simple changes in the same 3 lines of index.html as was done in the feature-branch-2 (Hint: we need to create conflicting changes for the purposes of this project)
   f. (master) commit your changes
   g. (master) Run a diff to visually inspect the differences & potential conflicts (before merging) with p4merge
   h. (master) Merge the feature-branch-2 with master. Merge conflict is expected to occur
   i. (master) Run visual merge tool p4merge. Visually, resolve the merge conflicts. Retain the changes which you want to keep and dismiss others. Save and exit p4merge tool.
   j. (master) Commit changes to conclude merge
   k. (master) Confirm that conflicts have been resolved (Hint: git status)
   l. If you see any generated file with **.orig** extension, then proceed to complete the following items in *Step-5*
   m. (master) Create .gitignore if it doesn't exist. Add appropriate entry within the .gitignore file such that .orig files are ignored.
   n. (master) Add .gitignore file to the staging
   o. (master) Commit changes
   p. (master) Check commit history to see if everything is OK
   q. (master) Delete branch feature-branch-2 now that you are done with conflict

resolution

r. (master) Merge feature-branch-1 with master

6. **Step-6: Diving deep into Git Branches (Part 2) - resolving to *rebase* conflicts**
    a. (master) Create a branch feature-branch-3 and checkout to the same
    b. (feature-branch-2) Make three simple changes in of the three lines of 404.html
    c. (feature-branch-2) commit your changes
    d. (feature-branch-2) checkout to the master branch
    e. (master) Make 3 different simple changes in the same 3 lines of 404.html as was done in the feature-branch-3 (Hint: we need to create conflicting changes for the purposes of creating rebase conflict)
    f. (master) Commit your changes
    g. (master) Checkout to feature-branch-3
    h. (feature-branch-2) Rebase feature-branch-3 with the master. Rebase conflict is expected to occur
    i. (feature-branch-2) Run visual merge tool p4merge. Visually, resolve the rebase conflicts. Retain the changes which you want to keep and dismiss others. Save and exit p4merge tool
    j. (feature-branch-2) After resolving conflicts visually, continue with rebase by running the “git rebase” command with the right option
    k. (feature-branch-2) Check 404.html to see if conflict resolution is rightly reflected in the file content.
    l. (feature-branch-2) Check the commit history to ensure that all looks good from the rebasing perspective
    m. (feature-branch-2) Checkout master branch
    n. (master) Merge feature-branch-3 with master
    o. (master) Check the commit history again to ensure that all looks good.
    p. (master) Delete branch feature-branch-3 now that you are done with rebase conflict resolution
    q. (master) Check the commit history again to see that all branch labels related to feature-branch-1 have been removed from the history

7. **Step-7: Pull from GitHub with rebase option**
    a. (master) Let’s run the git status command
    b. (master) When you run the “git status” command, you might notice that Git comes with a statement something similar to - Your branch is ahead of 'origin/master' by x commits, where the value of x depends on the number of pending commits to be pushed.
    c. (master) So, let’s push all the changes in our local repo to our upstream (remote) repo at GitHub
    d. (master) Once the push is done, let’s make 2 separate changes (in our local repo) to the robots.txt file and commit each time and thus generating 2 separate commits
    e. Now go to the remote repo on GitHub and make 2 separate changes to the humans.txt file and commit each time and thus generating 2 separate commits. (Note: normally we don’t make changes directly on the remote repo. Usually what happens is that some other team members push their changes on to the remote and we pull the upstream changes in our local repo. But here, we are making changes directly on the remote repo in order to simulate a collaborative

situation)

f. (master) Once the changes are done in the local and remote repo, do a git pull with rebase option in your local repo.
g. (master) Run a git log command. Can you see your local unpushed commits (to robots.txt) sitting on top (rebased) of the remote changes you pulled in? If yes, then you are good.

8. **Step-8: Team Collaboration in GitHub using a Pull request**
    a. Setup
        i. You need to have two GitHub accounts. Assuming you already have one GitHub account, please create another GitHub account.
        ii. Let's refer to the username for one account as "bob" and the username for another account as "alice" for the sake of explaining (Your actual GitHub usernames can be different).
        iii. Let's assume you are Bob and Bob is the owner of the GitHub repository and Alice is the collaborator
        iv. Let's name the repository as "pull-request-demo" of which Bob is the owner as (The actual name of your repository can be different).
        v. Since in reality both Bob & Alice are you, use 2 different browsers for each of them for accessing GitHub for simulating a collaborative environment.

    b. Assignment
        i. Now above is the setup. This setup demonstrates how Alice and make a Pull Request to Bob and the latter can approve it for merging.

9. **Step-9: Working with Git Stash**
    a. Working on the Master branch
        i. Edit index.html. Add some simple changes
        ii. Now stash away the changes without any comment
        iii. Now display the list of all stashes
        iv. Edit 404.html. Add some simple changes
        v. Now stash away the above changes with a suitable comment that will help you to identify the stash
        vi. Now display the list of all stashes
        vii. Edit browserconfig.xml. Add some simple changes
        viii. Add the above change to the staging area
        ix. Now stash away the above changes with a suitable comment that will help you to identify the stash
        x. Now display the list of all stashes
        xi. Create a new file test-file.txt
        xii. Now stash away the changes without any comment (Hint: test-file.txt is an untracked file)
        xiii. Now display the list of all stashes
        xiv. Now show the details of stash@{3}
        xv. Apply stash@{3}
        xvi. Commit the uncommitted changes after applying the above stash. Add a suitable comment to identify the commit
        xvii. Drop stash@{3}
        xviii. Now display the list of all stashes

xix. Now pop stash@{0} (Hint: apply and drop in a single command)
xx. Now add the changes in the working tree to the staging area
xxi. Commit the uncommitted changes with a suitable comment to identify the commit
xxii. Now display the list of all stashes
xxiii. Now pop stash@{0} (Hint: apply and drop in a single command)
xxiv. Again, pop stash@{0} (Hint: apply and drop in a single command)
xxv. Now stash away the above changes with a suitable comment that will help you to identify the stash

b. Working on a new Branch
   i. Now display the list of all stashes
   ii. Stash away all the changes to a new branch called “stashedbranch”.
   iii. The above step will apply all of the stash and hence you need to rebuild the stash list. Stash the changes without comment.
   iv. Now display the list of all stashes (Hint” there should be a single entry in the list)
   v. We now decide to scrap all of our stash and delete the branch. So clear the stash
   vi. Now display the list of all stashes to ensure that there are no entries in the stash
   vii. Checkout to the master branch
   viii. Delete the branch named “stashedbranch”

10. **Step-10: Delving into Git Tags**
    a. Create a lightweight tag named "v2.5-lightweight-tag"; tag it to the latest commit
    b. Edit index.html and make a simple change
    c. Commit your changes
    d. Create an annotated tag named “v2.9-annotated-tag” with a tag message; tag it to the latest commit
    e. Edit 404.html and make a simple change
    f. Commit your changes to 404.html
    g. Create another annotated tag named “v3.9-annotated-tag” with a tag message; tag it to the latest commit. This time, add the tag message using your default text editor for Git
    h. Display list of available tags in the repo
    i. Display tag details for the tag “v3.9-annotated-tag”
    j. Confirm that tag “v3.9-annotated-tag” is an annotated tag
    k. List all tags whose name contains the string “annotated”
    l. Do a “git diff” between v2.9-annotated-tag and v3.9-annotated-tag
    m. Edit robots.txt and make a simple change
    n. Commit your changes to “robots.txt”
    o. Create an annotated tag named “v4.9-simple-annotated-tag” with a tag message; tag it to the latest commit
    p. Display list of available tags in the repo
    q. Delete tag “v4.9-simple-annotated-tag”
    r. Confirm tag has been deleted by displaying [1] list of tags [2] commit history
    s. Modify git configuration to turn on pushing (to remote repo) of annotated tags by default

t. Confirm whether git configuration is updated properly
u. Now push all annotated tags (not lightweight tag) along with all local commits. Confirm from GitHub repo that all annotated tags have been pushed but no lightweight tags were pushed
v. Delete tag “v2.9-annotated-tag” from remote repo; confirm from GitHub repo if the tag was deleted

Submission

After completing the assignment, upload the text solution file. This file should contain the following:

- All git commands run for each of the steps of the assignment/project. Each command should start with “$” + single space
- Each command should be preceded by a line (or lines) of comment starting with a “#” + single space. The comment should describe the purpose of execution of the command.
- If the assignment/project involves working with GitHub, describe the steps taken in a step by step manner. The steps can be explained using comments (beginning with a “#” + single space)
- Each step mentioned in the assignment should be separated from other steps by few blank i.e new lines. You need to mention the step number, optionally with a description mentioning the purpose..

Solution

You can download the solution to the assignment from the progress report once you have uploaded the solution. You may compare your solution with the provided one.