Final Project Solution

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.

Solution statement with steps

```
#################### STEP-1 (Setting up local repository)

# It's assumed that you have implemented all the steps till Step-1 (f)

# Unzip the file initializr-verekia-4.0.zip
$ unzip initializr-verekia-4.0.zip

# Rename the directory "initializr" to "insh-final-project"
$  mv initializr insh-final-project

# CD (change directory) to the directory "insh-final-project"
$ cd insh-final-project/

#  Initialize the repository with "git init" command
$ git init

# List all the files/directories including hidden files.
$ ls -al

# Add all the files/directories to the staging area
$ git add .

# Commit all the artifacts (files/directories). Commit message is "initial commit".
$ git commit -m "initial commit"

# Run the git status command to see if the working directory is clean
$ git status
```

#################### **STEP-2 (Setting up remote repo)**

[1] Create a remote repo on GitHub
[2] The remote repo name should be same as local repo i.e, “insh-final-project”

Add a remote named “origin” for the remote repo
$ **git remote add origin** https://github.com/bibroy/insh-final-project.git

Now cross-check the creation of the remote added in the above step.
If remote creation went fine, proceed to next step
$ **git remote -v**

Push all the artifacts from local repo to remote repo
$ **git push -u origin master**

Now cross-check the creation of remote reference branch “origin”
If remote reference branch creation went fine, proceed to next step
$ **git branch -a**

Go to github.com and check if all of your artifacts got pushed from local repo
To check if git push went fine, simple refresh remote repo home page or click on the repo link

#################### **STEP-3 (Establishing SSH communication)**

Generate the SSH keys using the appropriate bash command
$ **ssh-keygen -t rsa -b 2048 -C "SSH keys for final project"**

Check if the keys have been generated successfully
$ **ls -l ~/.ssh**

Copy the public SSH Key to clipboard
$ **clip < ~/.ssh/id_rsa.pub**

Go to github.com and create a new SSH key (Hint: You need to use the key you copied in previous step)

Test the SSH connection - Are you behind a firewall? If yes run the appropriate command
$ **ssh -T -p 443** git@ssh.github.com

Test the SSH connection - Or, if you are not behind a firewall, run the appropriate command
$ **ssh -T** git@github.com

Run the appropriate git command to make the switch from HTTPS to SSH
Here replace “bibroy” (github username) with your github username
$ **git remote set-url origin** git@github.com**:bibroy/insh-final-project.git**

Run the appropriate git command to test if the protocol switch (HTTPS to SSH) worked correctly
$ **git remote -v**

Make some changes in humans.txt file; push to remote repo to make a final test for SSH connection
$ **vi human.txt**
$ **git commit -am "humans.txt - ssh push test"**
$ **git push**

########## **STEP-4 (Commit history in-depth)**

Building commit history
$ **vi robots.txt**
$ **git commit -am "building commit history with robots.txt - edit 1"**
$ **$ vi robots.txt**
$ **git commit -am "building commit history with robots.txt - edit 2"**
$ **vi humans.txt**
$ **git commit -am "building commit history with humans.txt - edit 1"**
$ **vi humans.txt**
$ **git commit -am "building commit history with humans.txt - edit 2"**
$ **git checkout -b feature-branch-1**
$ **vi browserconfig.xml**
$ **git commit -am "building commit history with xml file - edit 1"**
$ **vi browserconfig.xml**
$ **git commit -am "building commit history with xml file - edit 2"**

checkout to master branch
$ **git checkout master**

Run git log command with necessary options that does the following
[1] Decorates commit history with branch names
[2] Prints a graph of branches and
[3] Includes history of all branches
[4] History is displayed in oneline per commit
$ **git log --oneline --decorate --graph --all**

Run git log command with the right option that gives information about
[1] which files were altered as well as
[2] the number of lines added/deleted from each file
$ **git log --oneline --stat**

Run git log command with the right option that provides patch information such as diff
$ **git log -p**

Run git log command to display commit history for the directory “css” only
$ **git log css**

Run git log command with the option for searching commit history containing the string “initial”
Hint: you can use any query string that you want to search
$ **git log --oneline --grep="initial"**

########## **STEP-5 (resolving *merge* conflicts)**

Create a branch feature-branch-2 and checkout to the same
$ **git checkout -b feature-branch-2**

Make three simple changes in of the three lines of index.html file
$ **vi index.html**

commit your changes
$ **git commit -am "index.html - 3 changes in feature-branch-2"**

checkout to the master branch
$ **git checkout master**

Make 3 different simple changes in the same 3 lines of index.html as was done in the feature-branch-2
$ **vi index.html**

commit your changes
$ **git commit -am "index.html - 3 changes in master branch"**

Run a diff to visually inspect the differences & potential conflicts (before merging) with p4merge
$ **git difftool master feature-branch-2**

Merge the feature-branch-2 with master
$ **git merge feature-branch-2**

Merge conflict is expected to occur
Visually, resolve the conflict with p4merge
Retain the changes which you want to keep and dismiss others
$ **git mergetool**

Commit changes after resolving conflicts
This will open your visual text editor. Click Save and Exit.
$ **git commit**

Confirm that conflicts have been resolved
$ **git status**

If you see any generated file with .orig extension, then proceed as below
create .gitignore if it doesn't exist
add *.orig to .gitignore file since you do not want to track the .orig files
$ **vi.gitignore**

add .gitignore file to staging
$ **git add .gitignore**

commit changes
$ **git commit -m "updating .gitignore...."**

check commit history to see if everything is OK
$ **git log --oneline --decorate --graph --all**

Delete branch feature-branch-2 now that you are done with conflict resolution
$ **git branch -d feature-branch-2**

Also, merge feature-branch-1 with master
$ **git merge feature-branch-1**

Delete branch feature-branch-1 now that you are done with conflict resolution
$ **git branch -d feature-branch-1**

#################### **STEP-6 (resolving *rebase* conflicts)**

Create a branch feature-branch-3 and checkout to the same
$ **git checkout -b feature-branch-3**

Make three simple changes in of the three lines of 404.html
$ **vi 404.html**

commit your changes
$ **git commit -am "404.html: feature-branch-3"**

Checkout to master
$ **git checkout master**
# Make three simple conflicting (with master) changes in of the three lines of 404.htm
$ **vi 404.html**

commit your changes
$ **git commit -am "404.html: master"**

Checkout to feature-branch-3
$ git checkout feature-branch-3

Rebase feature-branch-3 with master. Rebase conflict is expected to occur
$ **git rebase master**

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
$ **git mergetool**

After conflict resolution, continue with rebase by running the “git rebase” command with right option
$ **git rebase --continue**

Check 404.html to see if conflict resolution is rightly reflected in the file content.
$ **vi 404.html**

Check the commit history to ensure that all looks good from rebase perspective
$ **git log --oneline --decorate --graph --all**

Checkout master branch
$ **git checkout master**

Merge feature-branch-3 with master
$ **git merge feature-branch-3**

# Check the commit history again to ensure that all looks good
$ g**it log --oneline --decorate --graph --all**

Delete branch feature-branch-3 now that you are done with rebase conflict resolution
$ **git branch -d feature-branch-3**

Check the commit history again
You should notice that all branch labels related to feature-branch-3 has been removed from the history
$ **git log --oneline --decorate --graph --all**

########## **STEP-7 (Git pull with rebase)**

check git status
$ **git status**

sync up local and remote
we can do a push without a pull here since we have not done any changes on remote
$ **git push**

[1] make 2 separate changes (in our local repo) to robots.txt file and commit each time (2 commits)
[2] make 2 separate changes (in our local repo) to humans.txt file and commit each time (2 commits)

Do a git pull with rebase option in your local repo
$ **git pull --rebase origin master**

Confirm if 'pull with rebase' worked by examining the commit history
$ **git log --oneline --decorate --graph --all**

########## **STEP-8 (Pull Request)**

1. Bob creates the repository "pull-request-demo" on GitHub. Bob downloads the file "initializr-verekia-4.0.zip" from https://github.com/bibroy/downloads for the purpose of populating the repository.
2. Bob populates "pull-request-demo" with the content of "initializr-verekia-4.0.zip" after unzipping the file.
3. Bob clones the repo "pull-request-demo" in his local computer.
4. Alice forks the repo "pull-request-demo"
5. Alice clones the forked copy (not original repo belonging to Bob) of the repo "pull-request-demo" in her local computer.

6. Alice makes some changes in the file 404.html in her local repo, commits it and pushes to the remote GitHub repo (forked copy of “pull-request-demo”)
7. Alice goes to https://github.com/alice/pull-request-demo (notice this is a hypothetical URL. You need to replace “alice” with whatever username you are using)
8. Alice clicks on the tab “Pull Requests”
9. Next, Alice clicks on the “New Pull Request” button
10. Next, Alice clicks on the “Create Pull Request” button
11. This opens the “Open a pull request” page
12. Next Alice adds a “Title” to the Pull Request along with a Comment
13. Next Alice clicks the “Create Pull Request” button
14. This will fire the pull request email notification to Bob. This email id is the same email with which Bob created his GitHub account
15. Bob clicks the pull request invitation link in the email which takes him to the GitHub page where he gets the link “Pull request from Alice”. Clicking on the link takes him to the page where he gets the “Merge Pull request” button.
16. Bob clicks the “Merge Pull request” button and adds a approval comment as well
17. Next Bob clicks on the “Confirm Merge” button which completes the approval process of Bob
18. Alice immediately gets notified that the pull request has been approved and changes have been merged by Bob.
19. This closes the pull request process fully

########## **STEP-9 (Git Stash)**

Edit index.html. Add some simple changes
$ **vi index.html**

Stash away the changes without any comment
$ **git stash**

display the list of all stashes
$ **git stash list**

Edit 404.html. Add some simple changes
$ **vim 404.html**

Stash away the above changes with a suitable comment that will help you to identify the stash
$ **git stash save "404.html"**

display the list of all stashes
$ **git stash list**

Edit browserconfig.xml. Add some simple change
$ **vim browserconfig.xml**

Add the above change to the staging area
$ **git add browserconfig.xml**

Stash away the above changes with a suitable comment that will help you to identify the stash
$ **git stash save "browserconfig.html**"

display the list of all stashes
$ **git stash list**

Create a new file test-file.txt
$ **vi test-file.txt**

stash away the changes without any comment (Hint: test-file.txt is an untracked file)
$ **git stash -u**

Now display the list of all stashes
$ **git stash list**

show the details of stash@{3}
$ **git stash show stash@{3}**

Apply stash@{3}
$ **git stash apply stash@{3}**

Commit the uncommitted changes after applying above stash.
Add a suitable comment to identify the commit
$ **git commit -am "index.html"**

Drop stash@{3}
$ **git stash drop stash@{3}**

Now display the list of all stashes
$ **git stash list**

Now pop stash@{0} (Hint: apply and drop in a single command)
$ **git stash pop stash@{0}**

Add the changes in working tree to staging area
$ **git add test-file.txt**

Commit the uncommitted changes with a suitable comment to identify the commit
$ **git commit -am "test-file.txt"**

display the list of all stashes
$ **git stash list**

Now pop stash@{0} (Hint: apply and drop in a single command)
$ **git stash pop stash@{0}**

Again, pop stash@{0} (Hint: apply and drop in a single command)

$ **git stash pop stash@{0}**

Now stash away the above changes with a suitable comment that will help you to identify the stash
$ **git stash save “404.html, browserconfig.xml”**

display the list of all stashes
$ **git stash list**

Stash away all the changes to a new branch called “stashedbranch”.
$ **git stash branch stashedbranch**

Stash the changes without comment
$ **git stash**

Now display the list of all stashes (Hint” there should be a single entry in the list)
$ **git stash list**

clear the stash
$ **git stash clear**

display the list of all stashes to ensure that there are no entries in the stash
$ **git stash list**

Checkout to master branch
$ **git checkout master**

Delete the branch named “stashedbranch”
$ **git branch -d stashedbranch**

#################### **STEP-10 (Git Tags)**

create a lightweight tag named "v2.5-lightweight-tag"; tag it to latest commit
$ **git tag v2.5-lightweight-tag**

Edit index.html and make a simple change
$ **vi index.html**

commit your changes
$ **git commit -am “index.html”**

create an annotated tag named “v2.9-annotated-tag” with tag message; tag it to latest commit
$ **git tag -a v2.9-annotated-tag -m "release version 2.9"**

Edit 404.html and make a simple change
$ **vi 404.html**

commit your changes
$ **git commit -am “404.html”**

create another annotated tag named “v3.9-annotated-tag” with tag message; tag it to latest commit
this time, add the tag message using your default text editor for Git
$ **git tag -a v3.9-annotated-tag**

Display list of available tags in the repo
$ **git tag --list**

display tag details for the tag “v3.9-annotated-tag”
$ **git show v3.9-annotated-tag**

confirm that tag “v3.9-annotated-tag” is an annotated tag
$ **git cat-file -t v3.9-annotated-tag**

List all tags whose name contains the string “annotated”
$ **git tag -l "*annotated*"**

Do a “git diff” between v2.9-annotated-tag and v3.9-annotated-tag
$ **git diff v3.9-annotated-tag v2.9-annotated-tag**

Edit robots.txt and make a simple change
$ **vi robots.txt**

commit your changes
$ **git commit -am “robots.txt”**

create an annotated tag named “v4.9-simple-annotated-tag” with tag message; tag it to latest commit
$ **git tag -a v4.9-simple-annotated-tag -m "release version 4.9"**

Display list of available tags in the repo
$ **git tag**

Delete tag “v4.9-simple-annotated-tag”
$ **git tag v4.9-simple-annotated-tag --delete**

Confirm tag has been deleted by displaying [1] list of tags [2] commit history
$ **git tag --list**
$ **git log --oneline --decorate --graph**

Modify git configuration to turn on pushing (to remote repo) of annotated tags by default
$ **git config --global push.followTags true**

confirm whether git configuration is updated properly
$ **git config --global --list**

Now push all annotated tags (not lightweight tag) along with all local commits
confirm from GitHub repo that all annotated tags has been pushed but no lightweight tags were pushed
$ **git push origin master**

delete tag “v2.9-annotated-tag” from remote repo; confirm from GitHub repo if tag was deleted
$ **git push origin :v2.9-annotated-tag**