|  |  |  |
| --- | --- | --- |
| **Qatar University**  **College of Engineering**  **Dept. of Computer Science & Eng.** | **quLogo94x89NT** | **Software Engineering**  **CMPS 411**  **Spring 2015** |

**Lab 8: Configuration Management with GitHub**

**Student Name: Student ID: \_\_\_\_\_\_\_\_\_\_\_\_**

***Task***: Practice with a configuration management software, git Hub using both GUI and command line terminal

***Lab topic(s****):* Configuration management with git Hub

***Lab objective(s****):* Learning the following basic features of git Hub using GUI and command line terminal (git Hub Shell)

* Configure git
* Understanding working directory, staging area, committed, and .git repository
* Finding differences between versions
* Understanding branching and merging
* Working with remote repository.

***Requirements***:

* For this lab session, you are required to bring you laptop installed with software git Hub.

***Lab activities*:**

|  |  |  |
| --- | --- | --- |
| **Activity** | **Resources and notes** | **Estimated time** |
| * Students write their name and SID (See first page on where to write these) | * This lab document | 5 minutes. |
| * Lab Instructor briefly explain the main aspects of software configuration management (SCM) | * This lab document | 5 minutes. |
| **Task 1:** **Practicing git Hub using GUI**   * Open your git Hub software * Create a repository called “**FirstProject**”. Notice that the repository will be created under the GitHub directory (default is in “My Documents”) * Create two java files with the following file names: **FirstJava.java; SecondJava.java;** with the codes given below, and save them in the directory **FirstProject** that you’ve just created * Click on the “Uncommitted changes” to view all source code files that are not yet saved for tracking (versioning) with git Hub. * Select the files to be committed. You can also expand the files to observe its changes. * Enter a commit summary, and then click the button “Commit to master” * To publish your respiratory, click the top right button “Publish Respiratory”. This will save a copy of your work in the git Hub master server for either public/private publication. The work will be published with the same local name that was created. | * This lab document * Git Hub Software | 20 minutes |
| **Task 2:** **Configure git, and setting for the lab**.   * Open your terminal, type the following commands to set your name and your email. You do it once:   $ git config --global user.name “your name”  $ git config --global user.email <your email>   * You now check if your email and name have been correctly configured with the system:   $ git config –-list   * Try also:   $ git config user.name   * Three ways you can get the manual page or help for any of the Git commands, <verb> stands for a command:   $ git help <verb>  $ git <verb> --help  $ Now create a directory (folder) in your computer, and go into that directory:  $ mkdir lab8\_CMPS411  $ cd lab8\_CMPS411   * Different operating systems handle line endings differently. Use the following command first to ensures git handles line endings correctly if you are using Mac OS X, otherwise you may get like, "*warning: CRLF will be replaced by LF in file*."   $ git config --global core.autocrlf input   * Now you are ready to go. Create three java files with the following file names: **FirstJava.java; SecondJava.java; ThirdJava.java**; with the codes given below using any editor, and save them in the directory **lab8\_CMPS411** that you’ve just created: | * This lab document * Any editor * Git Hub Shell | 15 minutes. |
| **<FirstJava.java>**  import java.lang.\*;  public class FirstJava {  public static void main(String[] args)  { System.out.println("This is my first Java program.");  StdOut.print("Type the first integer: ");  } }  ---------------------------------------------------------------------------  **<SecondJava.java>**  import java.lang.\*;  public class SecondJava {  public static void main(String[] args)  { System.out.println("This is my second Java program.");  StdOut.print("Type the second integer: ");  } }  -----------------------------------------------------------------------------------  **<ThirdJava.java>**  import java.lang.\*;  public class ThirdJava {  public static void main(String[] args)  { System.out.println("This is my third Java program.");  StdOut.print("Type the third integer: ");  } } | | |
| * **Task 3: Understanding /.git repository, staged and committed.** * Now your working directory contains three files. Most operations in Git only need local files and resources to operate – generally no information is needed from another computer on your network at this stage. * Git has *three main states* that your files can reside in: *Working directory, Staged (index), Committed(HEAD).* * In the **working directory**, you can modify the file as you wish, not staged yet for the git repository (HEAD).   Working directory  FistJava.java  SecondJava.java  ThirdJava.java  Committed (HEAD)  /.git directory  Staging area (index)   * **Staged** (index) means that you have marked a modified file in its current version residing in the working directory to go into your next commit snapshot. * **Committed** means that the file is safely stored in your local /.git repository (HEAD). * We will talk about the **remote repository** later of this lab. * Git thinks about its files more like a **stream of snapshots**. In order to create/initialize empty git repository (HEAD) in your current directory like: /lab8\_CMPS411/.git/, use the following command:   $ git init | * This lab document * Eclipse * Java perspective   Remote repository  https://…… | 20 minutes |

|  |  |  |
| --- | --- | --- |
| * Now you have a .git repository. Add the file FirstJava.java to the staging area (index), and check the status after adding the file. How? Use these:   $ git add FirstJava.java  $ git status   * It shows FirstJava.java is a new file in the staging area, that means, this file has been marked as staged. It also shows other two files as untracked files in your current working directory, meaning, these are not staged yet. Snapshot of your /.git directory conceptually looks like the following:   Committed (HEAD)  /.git  Working directory  SecondJava.java  ThirdJava.java  Staging area (index)  FistJava.java   * Notice the snapshot of the above. However, you can now *unstage* your file FirstJava.java using the command. Do not do this now, we will practice this later.   $ git rm --cached FirstJava.java.   * The above command takes the file back to the working directory. * git add can take options such as   git add –all (take all files to the staging area)  git add . (same as above)  git add \*.txt  git add <a list of file>   * Try these later on your own. You can now commit your staged file FirstJava.java to the /.git repository (HEAD). Do the following command.   $ git commit –m “My first commit to FirstJava.java”   * A commit must be associated with a meaningful comment as above. By writing clear commit messages, you can make it easier for other people to follow along and provide feedback. **Note, the commit records the snapshot you have set up in your staging area.** Anything you didn’t stage is still sitting in the working directory. To see what happened after your commit:   $ git status (give this command often)  $ git log (give this command often)   * What do you see now? git log shows repository history, to look back to see what has happened. git log lists the commits made in that repository. You can use this with various options such as   –p  -p –2  --pretty=oneline  --author=<author name>  --graph --oneline --decorate  --graph --oneline --decorate –-all   * Try these with git log. The snapshot of your .git directory has now changed after the commit:   Committed  (HEAD)  /.git  FistJava.java  Staging area (index)  Working directory  SecondJava.java  ThirdJava.java   * Every time you commit, or save the state of your project in git, it basically takes a **picture (snapshot**) of what all your files look like at that moment, and stores a reference to that snapshot. The code that you see after commit is the SHA-1 encoding of the file. * You can also unstage a file. Try these.   $ git add SecondJava.java  $ git status  $ git reset -- SecondJava.java (this will unstage SecondJava.java to the working directory)  $ git add .   * This will stage all files from your working directory to the staging area)   $ git status (to see the status of files)  $ git reset (all staged files are to be unstaged)   * In this case, SecondJava.java and ThirdJava.java will be unstaged)   $ git status   * Basically, $ git reset is used to undo a $ git add command.   $ git add SecondJava.java (file is staged)  $ git checkout -- SecondJava.java   * It copies SecondJava.java from the stage to the working directory. Use this command to throw away local changes). All these commands are summarized in the diagram:   $ git reset  $ git commit  $ git add  $ git checkout  Working directory  ThirdJava.java  Committed  (HEAD)  /.git  FistJava.java  Staging area (index)  SecondJava.java  **Q. 8.1: Lab questions:**   1. What is the main difference between $git reset and $git checkout? 2. Give the command $ls. What do you see? All three files are in your working directory although one has been staged and one has been committed. Why all three files are still in your working directory then? |  |  |
| * **Task 4: Finding differences between versions** * Go to your editor, load FirstJava.java, and add the following line after theStdOut.printstatement:   System.out.println("Type second integer.");   * Save the file and write the following command in the console   $ git add FirstJava.java  $ git status   * What do you see? To know exactly what you have changed, not just which files were changed, try this command:   $ git diff  In order to know:   1. What have you changed but not yet staged? 2. And what have you staged that you are about to commit?  * Although git status answers these questions very generally by listing the file names, git diff shows you the exact lines added and removed. It compares what is in your working directory between what is in your staging area. The result tells you the changes you’ve made but you haven’t yet staged. You can use this command with options such as   git diff –staged  or  git diff –cached   * Now do the followings and understand what you see:   $ git commit –m “My second commit to FirstJava.java”  $ git log --pretty=oneline   * The above command lists all committed files in order with their full hash number]   $ git log --graph --oneline --decorate --all  or,  $ git log --graph --oneline --decorate   * The above command shows the history of your commits, displaying where your branch pointers are and how your history has diverged. It shows similar to the following information:   <a hash number> (HEAD, master) My second commit to FirstJava.java  <a hash number> My first commit to FirstJava.java |  | 10 minutes |
| * **Task 5: Working with branches** | * This document | 25 minutes |
| * **master:** A branch in Git is a pointer to one of the commits. The default branch name in Git is **master**. Changes you make on a branch don't affect the master branch. As you start making commits, **you’re given a master branch that points to the last commit you made**. Every time you commit, **master** moves forward automatically to the latest commit. In the previous command, therefore, master is pointing to your last commit, that is: My second commit to FirstJava.java * **HEAD**: Git keeps a special pointer called HEAD. This is a pointer to the branch you’re currently on. In the last command, it shows you’re still on master. That means, master and HEAD are pointing to the last commit. You can move HEAD to any branch, but not **master**. * Let’s create a branch at the last commit.   $ git branch one-branch  $ git --graph --oneline –decorate  Or,  $ git log --graph --oneline --decorate –-all   * It shows that one-branch has been created at the last commit. HEAD, master, and one-branch all are pointing to the last commit. HEAD did not switch to the newly created branch. Conceptually it looks like the following diagram   *one-branch*  *HEAD*  *Master branch*  *(trunk)*  Second commit  First commit   * However the following command moves HEAD to the branch:   $ git checkout one-branch  $ git log –graph --oneline –decorate   * What do you see? * Go to your editor, load FirstJava.java, and modify the **last** statement such as:   System.out.println("Type 3rd integer.");   * Save the file and write the following command in the console   $ git add FirstJava.java  $ git commit –m “My third commit to FirstJava.java”  $ git log --oneline --decorate   * What do you see now? It shows that master is still pointing to the previous commit (second commit), but HEAD and one-branch have moved forward and are pointing to the latest commit (third commit) as shown below in the diagram (*green colour signifies master branch, and yellow for one-branch*): * **Why master did not move forward to the last commit?**   *one-branch*  Third commit  *HEAD*  *Master branch*  *(trunk)*  Second commit  First commit   * To switch back HEAD to the master branch, give the following command:   $ git checkout master  $ git log --graph --oneline --decorate   * What do you see? The first command has moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to.   Third commit  *HEAD*  *Master branch*  *(trunk)*  Second commit  First commit   * Note the equivalency of commands: git branch <a branch name> and git checkout <a branch name> can be achieved by a single command: git checkout –b <a branch name>   *one-branch*   * Go to your editor, load FirstJava.java, and modify the **last** statement such as:   System.out.println("Type 4th integer.");   * Save the file and write the following command in the console   $ git add FirstJava.java  $ git commit –m “My fourth commit to FirstJava.java”  $ git log --graph --oneline --decorate  **Q.8.2: Lab questions:**   1. What do you see now after the above command? 2. How to make HEAD point to the branch one-branch? 3. Change FirstJava.java by adding one more statement like this:   System.out.println("Type 5th integer.");   * 1. Commit the file with the comment “My fifth commit to FirstJava.java” to the branch one-branch. Conceptually it should look like the following diagram:   *HEAD*  *Master branch*  *(trunk)*  Second commit  First commit  Third commit   1. Go to your editor, load FirstJava.java, and modify the last statement such as: System.out.println("Type 6th integer.");   Fifth commit  Fourth commit   * 1. Commit it at the master branch with a comment “My sixth commit”   2. Modify the file FirstJava.java by changing the statement such as: System.out.println("Type 7th integer.");      + Commit it at the master branch with the comment “My seventh commit”  1. Create another branch called two-branch at one-branch 2. Modify FirstJava.java again with the following in the last statement such as: System.out.println("Type 8th integer.");    * + Commit it at one-branch with the message “My eighth commit” 3. Modify FirstJava.java again with the following in the last statement such as: System.out.println("Type 9th integer.");    * + Commit it at two-branch with the message “My ninth commit” 4. Check your /.git directory to see all commits and the branches that you have created. Conceptually it should look like the following diagram. The diagram has three branches:   *one-branch*  Eighth commit  Seventh commit  sixth commit  Fourth commit  Third commit  Second commit  First commit  Fifth commit  *one-branch*  *HEAD*  *Master branch*  *(trunk)*  *two-branch*  Ninth commit | | |
| * **Task 6: Merging branches** * You can merge one branch to another such as one-branch to the master branch. See the previous diagram again. Before you merge, you can check the differences between two branches using $ git diff with the following options:   $ git diff --stat master one-branch   * The above command shows the common commit of the two commits, and finds the differences. To compare two branches, you can also use the following with three dots.   $ git diff --stat master…one-branch  $ git diff HEAD   * The above shows what has changed since the last commit   $ git diff HEAD^   * It shows what has changed before the last commit.   $ git diff –cached   * Shows what has been added to the staging area via $ git add, but not committed.   $ git diff   * Shows what has changed but has not been added to the staging area yet. * Now it is time to merge two branches.   $ git checkout master  $ git merge one-branch  The above two commands could be achieved with a single command:  $ git merge master one-branch   * After the merge operation above, the conceptual scenario of the .git repository looks like this: |  | 15 minutes |
| Eighth commit  First commit  Seventh commit  sixth commit  Fourth commit  Second commit  Third commit  *Merge commit automatically done by Git*  *one-branch*  Ninth commit  *two-branch*  *HEAD*  *Master branch*  *(trunk)*  Eighth commit  Fifth commit  *Common ancestor* | | |
| * You can now delete one-branch because you do not need this pointer after the merge:   $ git branch –d one-branch   * Note that the commit on the branch you’re on isn’t a direct ancestor of the branch you’re merging in. In this case, Git does a simple three-way merge, using the two snapshots (Seventh commit pointed by HEAD, and Eight commit) pointed to by the branch tips and the common ancestor of the two. * Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a **merge commit**, and is special in that it has more than one parent. * The following command will delete the branch, two-branch if you do not need this, or you can merge this with master.   $ git branch –d two-branch (to delete)  or  $ git checkout master  $ git merge two-branch  or, (using a single command without checkout command)  $ git merge master two-branch   * Occasionally, this process doesn’t go smoothly. If you changed the same part of the same file differently in the two branches you’re merging together, Git won’t be able to merge them cleanly. * Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts.   $ git branch   * You can see a \* prefix the master branch. It indicates the branch that you currently have checked out (i.e., the branch that HEAD points to). This means that if you commit at this point, the master branch will be moved forward with your new work. To see the last commit on each branch, you can run .   $ git branch –v   * The useful --merged and --no-merged options can filter this list to branches that you have or have not yet merged into the branch you’re currently on.   What have you achieved with all these in this lab so far? The every change you have made to FirstJava.java have been saved in a sequential order. No changes have been lost. |  |  |
| * **Task 6: Working with remote repository** |  | **20 minutes** |
| * Ultimately, you need to work with other team members in a project, therefore, you will use remote repository. Working with remote repository includes knowing how to add remote repositories, how to get files from remote, how to upload your changes to the remote directory, etc. * When you put something on your GitHub account, that copy is stored on one of the GitHub’s servers. This makes it a remote repository because it is not on your computer. By pushing your local changes to it, you keep it up to date. * Other people can get the latest from your project by puling your changes down from the remote repository.   **Creating a remote repository:**   * Go to github.com, log in you git Hub account, and click on the ‘+’ on the top left corner of the git GUI window to create a new repository called lab8\_CMPS411, matching your local repository name located in your machine in order to sync. * Make it public * Click on create repository button.   *Note: You can create a GitHub repo via the command line using the GitHub API. You can try this on your own.*  **Connecting your local repository to your remote repository:**   * Now you have a remote empty repository called lab8\_CMPS411 located on GitHub.com * Go back to your command line terminal * You now need to tell Git the location of your remote repository on GitHub’s server * Your remote repository is commonly named origin * Give the following command if you are using Mac machine:   $ git remote add origin <the URL of your remote repository from Git Hub>   * The URL address of your remote repository at git should look like the following:   https://github.com/your git login name/lab8\_CMPS411.git   * You can find the correct URL of your git account from the Git Hub site. * For MS Windows users, give this command:   $ git remote set-url origin <the URL of your remote repository on Git Hub>   * Your local repository now knows where your remote repository ‘origin’ is located on GitHub’s servers. * Check now if you have correctly configured with your remote repository. Give the following commands:   $ git remote  $ git remote –v   * What do you see? Is the information correct?   **Send everything you have done locally in this lab so far to your remote repository:**  $ git push origin master   * The above will **send** all versions of FirstJava.java saved in the master of your local machine to your remote repository on GitHub * You can also send a branch to your remote repository by the following command:   $ git push origin <branch name>   * The push command shares your files with other too. The command for this is simple:   $ git push <other remote-name> <branch-name>   * You can see the URLs of the remote server.   $ git remote show origin   * This shows more information about a particular remote. It lists the URL for the remote repository as well as the tracking branch information. |  |  |
| * Now we will practise a bit more * Create a new director in your local machine   $ mkdir my-remote  $ cd my-remote  $ ls   * Your my-remote directory is empty. Give this command:   $ git clone https://github.com/your gitHub login/lab8\_CMPS411.git my-git-files   * The above $ git clone gets a copy of your existing Git repository to your local directory my-git-files under my-remote. You can, for example, copy a project from a remote directory you’d like to contribute to – the command you need is:   $ git clone <URL>   * This command creates a directory named “my-git-files” in your computer under the folder my-remote, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. * Now do the following commands:   $ ls   * It must show the directory: my-git-files   $ cd my-git-files  $ git log --graph --oneline --decorate   * You can see all revisions your created (committed) in there, ready to be worked on or used.   **Fetch/pull commands:**   * git fetch origin command fetches (pulls) all the information that origin has but only those that you don’t yet have in your repository. Fetch the latest history from the server and point your local master branch at it. * git fetch command pulls the data to your local repository – it doesn’t automatically merge it with any of your work or modify what you’re currently working on. * git fetch <remote-name> command goes out to that remote project and pulls down all the data from that remote project that you don’t have yet.   $ git pull   * The above command directly pulls the update from the remote repository to your current working directory.   **Q.8.3: Lab Question:**   1. What is the main difference between $ git fetch <remote-name> and $ git clone <remote-name> ? 2. What is the single command that creates a branch, and moves HEAD to the newly created branch? |  |  |
| * The above basic exercises in this lab would help you to explore Git Hub further on your own. It has a lot of useful features. You can discover them as you practice newer commands. |  |  |
| * Save your tasks and submit them to your lab instructor |  |  |
| * Leave the lab clean and tidy. |  |  |

**More learning resources for Git Hub:**

<https://guides.github.com/activities/hello-world/>

<https://guides.github.com/introduction/flow/index.html>

<http://jlord.us/git-it/challenges/branches_arent_just_for_birds.html>

<https://try.github.io/levels/1/challenges/1>

<http://rogerdudler.github.io/git-guide/>

<https://www.codeschool.com/courses/try-git>

<https://training.github.com/kit/downloads/github-git-cheat-sheet.pdf>

<https://training.github.com/kit/>

<https://help.github.com/articles/good-resources-for-learning-git-and-github/>