Lab - Git Workflow Basics

Lab - Git Workflow Basics

This assignment needs to be done individually! You are responsible for being proficient with git in your teams and this is one step in that direction.

LEARNING OBJECTIVES

- Become familiar with git for software management.
- Understand how to setup git for use with your own private repositories.
- Use the basic git commands to establish an initial workflow for code development.

The first sections of this lab repeat what we went through in class. If you feel really confident with what we managed to do in class so far with git, you can skim over the sections but make sure you create a new repository and work through the ***Try It*** parts. Otherwise, it would be wise for you to spend time reading these sections and repeat some of these steps to make sure you understand them.

NOTE: The Try It sections will have you record output from commands. I strongly suggest you keep that output in a file so you can (1) turn it in and (2) use it in a later part of the lab without having to redo it.

Section: Setting up Git

The first item to take care of is to make sure you have a working version of git on your development machine. We're staying agnostic in this class so if you want to develop on your Windows PC, your Mac, or you Linux box, go for it.

For Windows, you will need to download the git client, which includes a git bash shell.

https://git-scm.com/downloads

If you're on a Mac, make sure you have Xcode installed from the App Store. Depending on your version of OS X, you may be prompted to install the Xcode Command Line Tools. If asked to do this, make sure to click the "Install" button.

And, finally, if you're on Linux, you'll need to make sure git is installed from one of the packages systems. For Ubuntu, this will be

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

Once git is installed on your machine, we will work through setting up git to use SSH keys, which will result in it working well (and securely) in our development environments. Git can use different protocols to transfer repository information to and from distributed servers. One way is through HTTP(S), but the more reliable and secure means is through the SSH protocol. SSH, as you may already know, is a protocol for securing network traffic and is typically used to login to a remote machine (e.g. ssh csdev02.d.umn.edu).

To use SSH with git, you will need to generate a ssh key on the machine you want to develop with. To do this, you will run the ssh-keygen command. This command is available from the command line on Linux, Mac OS X, and from within the git-bash shell that comes with the git install for Windows. Note that if you already know what a SSH key is and you have one, you can skip the steps below to create a key.

To generate the ssh key, enter the following command and press Return or Enter:

```
ssh-keygen
```

You will be prompted for where to save the file. Just accept the defaults. You will also be prompted for a passphrase. You can choose one if you would like, but it is not strictly necessary. If you choose a passphrase the ssh systems on your computers will likely ask you the password before the key is used as you continue on with this course. Note that OS X and Linux will use a ssh-agent to record your passphrase for a time and submit it on your behalf so you may not actually have to type it all that much.

Once your ssh key set is generated, you will have two keys: a private key (.ssh/id_rsa) and a public key (.ssh/id_rsa.pub). Do NOT give out the private key; these are RSA public-key cryptography key pairs so the private key must remain private. You can however, provide your public key to others to ensure secure communication.

In your shell concatenate the file to the terminal window:

```
cat ~/.ssh/id_rsa.pub
```

You'll see something like this:

```
ssh-rsa AAAABJKSDzzxdkjllas....
```

it will go on for a lot longer than that snippet.

Now, login to https://github.umn.edu and click on your Profile (it's your "picture" up in the right hand portion of the web page). In your Profile, select Settings. You will then see a tab on the left of the page titled "SSH keys". Select that and add your new SSH key. All you have to do is give it a name so you know which key (you can have multiple keys) and then paste in the public key you concatenated to the terminal.

Now, you're ready to use git!

Step 1: Creating and Cloning a Repository

The first git exercise will be to clone a repository you create on github.umn.edu under your own account.

First, go back to the https://github.umn.edu site, login and then create a repository. You'll find a "+ New Repository" or "Create New Project" link on your main page once you've logged in.

For now, create a Private repository so that you can play around with git on your own. Give the repository a name and a description that informs you what the repo is for. Also, make sure to initialize the repository with a README.

Finally, before you click the "Create Repository" button, add a .gitignore file. From the selection list, pick a language of your choice for your .gitignore file. Create the repository.

Now, go back to your development machine and clone the repository. You can find the repository URL that you will use to clone your new repository on the right side of the github webpage for your repository. Make sure you select the SSH clone URL. It will look something like this:

git@github.umn.edu:willemsn/test2.git

To clone the repository, use the git clone command:

```
git clone <Repository URL>
```

so, in my case this would be

```
git clone git@github.umn.edu:willemsn/test2.git
```

which will create a directory with the same name as your repository in the current directory that you execute that command from. In this example, it will create a directory called test2 and the contents will be the .gitignore and README.md files that were added when the repository was created on github.

LEARNING OBJECTIVE SUMMARY

At the end of this step, you should know how to create a repository on github and clone it on your development machine. Here are a few additional insights into how git is working: git uses a distributed repository system - when you clone a repository, you are creating a replica or copy of the repository on your own machine. There is no real centralized server notion in git. Any repository that is cloned from another repository becomes a valid repository itself. This is unlike other software revision systems (such as svn). Having stated that, when you clone a repository, you create a working copy for yourself, but your repository is also linked "upstream" to the remote repository it came from and this connection is given a name. By default, the name used for the remote end of a cloned repository (where it came from) is "*origin*".

Step 2: Configuring your Credentials

On your development machine, you will need to configure git (once) to make sure it stores your name and your email. The reason for this is so that when you make commits back to other repositories, your information will be propagated correctly. You will use the git config command to set both your user name and your email. The commands look like this:

```
git config --global user.name <Your Name in Quotes>
git config --global user.email <Your Email in Quote>
```

for instance, mine would like

```
git config --global user.name "Pete Willemsen"
git config --global user.email "willemsn@d.umn.edu"
```

Do this now for your own information. Using the same command but with different options, you can set your preferred commit log editor with git:

```
git config --global core.editor emacs
```

Feel free to pick the editor you would prefer to use. If you're unsure about code editors, I would suggest Atom (https://atom.io/), Sublime (https://www.sublimetext.com/) or even Visual Studio Code (link). Set your editor to what you want now.

Finally, to check the status of the configuration options, use the --list option:

```
git config --list
```

Step 3: Adding and Committing Files

Your repository doesn't contain much yet. You will need to add files often and each time you do this, you will need to add the file to your git repository "database" (i.e. by database, git stores records of what it's doing in a directory called .git). To add a file after you've created it, use the git add command. Note that the git add command is also used to "add" files to the staging area (as we discussed in class -- be sure you understand this):

```
git add <filename or directoryname>
```

You can add files individually with that command or you can add entire directories. When you use that command you help to stage those files and changes for future committing to the repository.

Try It: Add a source code file (Java, C++, whatever you'd like) to your repository now by first creating the file, placing some content into it and then using git to add it.

Once files are added, they are staged for the commit process. Adding a file and making a commit in git creates a snapshot of your code that you can revert back to in the future, exam separately later and basically, just record the process of your code development. When you issue a git commit command, you record these snapshots of your code into your local repository on your development machine. The commit command requires that you enter a log message. If you type

```
git commit
```

you will be prompted with your preferred text editor to type in the reason for the code edits, what you did. My strong suggestion to you and your teammates --- BE DESCRIPTIVE with these log messages. You can add a smaller message on the command line using the -m "log message here" option of the commit command. Sometimes, you will want to commit all of the changes you've made but may not have added them to the current staging area with git add To do this, you will add the -a option to the git commit command which does the add for you while it's committing the files. Here's an example:

```
git commit -a -m "Added new line to output version information in the main function and updat
ed where this function is called in the startup procedures."
```

which would commit any files that had been previously added and modified.

Alternatively, you can commit a single file by specifying it as the last argument on the command line:

```
git commit -m "More changes that fix the issue" file1.cpp
```

Now, practice using the add and commit commands of git.

Try It: Add three (3) files and then edit and commit one of those files at least three (3) times. Make sure all files are committed.

LEARNING OBJECTIVE SUMMARY

By the end of this step, you should feel comfortable adding files and staging those files into snapshots with the git commit command. You'll be using this style of workflow frequently. Again, remember that you maintain your own repository on your development machine. Your commits are still on your machine and will not be contained with the "origin" repository from where your clone was made. Check the github site now to make sure this is correct. Also, git's commit system records the commit with a SHA-1 hash so that you can go back to any commit and inspect it later.

Step 4: Checking git Status

You're going to need to occaisionally see what the status of your repository is at some moment in time. You might want to know if you committed a file yet, or if you've added a file. You can do that with the git status command.

```
git status
```

Try It: What does it report. Edit your README.md file and add a section about git status and what it does. Then, copy/paste the results from git status in there.

Try it: What's different? State the difference in the README.md file you just edited.

Now, commit your changes to the README.md file. Check git status again.

Step 5: Master and Checking Out Commits

git works by utilizing what are known as branches. You should know that when you created your repository on github a main branch of development was created for you. It is called "*master*". In fact, git status will return to you the branch you are working on. Run git status now and find where it tells you which branch you are on.

Let's examine this a bit more now. You are now on the master branch. You've also made a few commits. To view the commits you've made, use the git log command to inspect them:

```
git log
```

This will return you a summary of your commits along with your log messages. Note that with each commit, you will see the SHA-1 hash listed next to the commit, as in the following output example:

```
commit 792383c94c66c726ef0636cf2b1f807cb1699312
```

You can do a read-only check out of any previous commit without harming your current master branch by using the git checkout command. Try it now. Run git log and find a previous (somewhere in the middle of your edits) commit. Check it out and inspect it.

```
git log
```

```
git checkout <commit hash here>
```

Look at your files. Verify that they are now what they were. Run git status. It will now show you that you are detached from the head. In other words you are no longer in the main master branch. When you're done, you can switch back quickly to the master by checking it out:

```
git checkout master
```

Make sure you try these steps out where you checkout a previous snapshot and then go back to master, or even jump between snapshots.

Step 6: Pushing Back to the Origin

So, you've made a lot of changes but they are only in your local repository. Go ahead and check your repository on github.umn.edu now. You should NOT see the changes you've been making to this new repository. To get them up there you need to push to the remote repository. Since this is a private repository there will not be any conflicts with your commits and pushes (since it's only you developing on one machine at the moment), so we can use the git push command very simply to push the local commits you've staged back up to github.umn.edu. You can do this with the git push command.

```
git push
```

Try It: Then, go to https://github.umn.edu and check your repository. See if it worked for you.

Section: Working with Branches

LEARNING OBJECTIVES

- Create, modify and delete branches with git.
- Fetch changes and Merge branches back into the master branch.

Your goal for lab is to create a branch of your own, modify it and merge it back into the master branch. The style we will use in the lab mimics how you should use git in collaborative environments in which multiple commits may be happening to the master branch while new features/ideas are being tested or integrated in branches. Today's branch work will be simpler and we will eventually ramp up into more complex interactions and how to deal with them.

Branches

When you start using git, there is a branch called "master". It is the main thread or line of development. You or your group might tend to think of master as being the primary set of code that you are developing, or the code you wish to be most stable. Branches are a way to create a parallel line of development that does not conflict with the master branch (or any other branches) but relates to it. You should think of branches as ways to integrate a new feature, fix a bug, try out new ideas and test them before placing it in the master branch and affecting other developers. This style of workflow is very beneficial to large group development in which lots of developers interact with source changes.

Using the repository you created for this lab, make sure you (1) have it cloned [you should-- we just worked on it] and (2) you are back in the master branch. Remember, when you first clone a repository, you are placed into the master branch. You can confirm this by running the git status command. It reports the branch that you're currently working with.

Next, create a branch that you can use for a little while to make some changes to the files. To create a branch, you can use the git branch command, but you can also use the git checkout command. The benefit of using git checkout is that it both creates the branch for you AND checks it out for you at the same time. Using git branch requires a second checkout command to work on the branch. Do this now:

```
git checkout -b YourNewBranchName
```

Of course, replace YourNewBranchName with something that identifies your branch. It is always good to be descriptive so others (when they work with your repo might know what that branch is for). For instance, "bugfix_23723" might be a good branch name to identify a bug fix related to issue 23723. In general, branch names should relate to the features or ideas they are testing. The -b option signifies to the checkout command to *create* a branch.

Check git status again. It will show you being on your new branch. Note that so far, no one else will know about your new branch because it has been created locally. Try that now. Look on github.umn.edu and see if your repository shows any branches.

Next, to push your branch back up to the remote you would type:

git push -u origin YourNewBranchName

which signifies that the branch YourNewBranchName will be placed upstream (-u option) on the remote. Go ahead and re-check github.umn.edu to see if your branch is there.

Now, do two things:

Add a file to your branch with some source code in it.

Edit the README.md file and add a summary of the output you've gotten so far from the Try It parts of this lab.

Once you have completed the two items above, make sure to commit them to your branch using the git commit command. Check git status to make sure you have no changes that have not been committed to your branch. Feel free to push your branch changes back up to the remote (this time you will not need the -u since you did that already; git push would work just fine).

Congratulations! You have completed your "feature" and are now ready to merge your branch back into the master branch!

Fetching and Merging

At this point in a more complicated lab, it is quite likely that others would have edited the README.md file or something else. Since you're editing the same file, git is going to have a challenging time understanding how to merge these changes and will rely on the developers to make the merges themselves. We will be a little simple this time and only have you deal with merging your branch back into the master.

The process that is best to use with git when working collaboratively is to use the git fetch and git merge commands to bring your branch back into the master. These commands are preferred over git pull (which can sometimes do this when no conflicts exist), because git merge creates a merge commit which summarizes what you did.

Let's try this now.

First, call switch back to the master branch.

```
git checkout master
```

Then, fetch any remote changes that may have occurred since you last checked:

```
git fetch
```

This will fetch the specific commits that have happened and report these potential changes to your code. It will not, however, modify your local copy! That's great because you can then decide (in a more complicated situation) if you really want to merge or not.

We do want to merge for the lab, so the next step will be to perform the merge with the branch you just created. We will be specific by stating the exact branch we want to merge with master:

```
git merge YourNewBranchName
```

Now, here comes the fun part if we were working in a more complicated scenario. There may or may not be conflicts. If there are, good for you - you get to practice an important piece of using git! In a future class or lab, we will be working through how to deal with merge conflicts.

After you merge, commit and push your changes back to the master.

With any luck, your changes will be present on the master branch now. Go up to the UMN github site and verify! You can also check with git status to make sure all is well too.

Deleting a Branch

Once you're sure your changes are back in the master branch and your branch feature is no longer needed (because its in the master now). You can delete it.

Make sure you are on the master branch (hint: use git status to check). Then issue the local branch delete:

```
git branch -d YourNewBranchName
```

which deletes it locally. You can then push that delete to the remote with the following command:

```
git push origin :YourNewBranchName
```

And with that, your done with git for today.

LEARNING OBJECTIVE SUMMARY

You've reached the end of the lab. By this point, you should feel comfortable using the basic git commands to work with your source code and a private repository. In the next lab and in class, we will work through some of the workflow issues and patterns that will be necessary when working as a team on a shared set of repositories.

Before you end the lab, you must add Jon Rusert and myself as Collaborators on your private repository. You can do this by going to your main repository page, select the Settings link on the right side and then find the Collaborators tab. After that, add

ruse0008

willemsn

We only need read access so that we can see your work and make sure you completed the lab.

Grading summary

Participants	27
Submitted	2
Needs grading	1
Due date	Wednesday, September 7, 2016, 11:55 PM
Time remaining	Assignment is due

View/grade all submissions