Git Lab Exercises

As a precursor to your future work with git we have provided a couple of exercises that will allow you to play with git and get to know more of the commands. These exercises simulate a collaborative development environment with more than one developer. As you go through the cohort you will find yourself pair programming and as such will be working with another developer. Feel free to use these exercises to brush up and review the commands anytime you run into issues.

Exercise 1: Simple Rebase

This exercise will be similar to the exercise found in the pre-work where you simulate two developers working on the same code base and performed a merge. Only this time we will introduce rebase and how it can keep the history a little cleaner. This exercise can be done solo by cloning two copies of the repository or with a parter where each of you take on one of the roles.

Also note that the exercise servers as guidelines for investigating these concepts. You can feel free to expand on this and keep working with it. You can and should also run git status and git log frequently to see what is happening with the repository. This is the best way to get familiar with the commands, what they do and how this all works.

Now, for the sake of clarity, Wise and Ward are at it again. I will use their names to distinguish which user should be doing what. If working on your own, feel free to name the directories as such. If working in pairs, one person can be Wise and one can take the actions of Ward.

Steps

1. Create a repository on github.

for clarity the one who creates the repository will be Ward.

2. Ward can start things off by cloning the repository using the git clone command. Remember the URL can be found on github and copied directly from there. Also remember the optional parameter on the command to specify the directory name to create.

```
$ git clone <URL>
```

or if you are simulating this on your own:

```
$ git clone <URL> <DIR>
```

where <dir> can be Ward.

3. Ward will now add a file to the repository, commit the change and push it up to github.

Example:
Create a text file (file1.txt) and add the text “Ward: This is a text file”
Save the file in the repository directory
Use the git add command to add the file

```
$ git add *
```

Use the git commit to save the change to the repository

```
$ git commit -m “initial commit”
```

Then git push to send those changes to github.

```
$ git push origin master
```

4. Now with that change pushed to github we can have Wise clone the repository and start where Ward left off.

```
$ git clone <URL>
```

or if you are simulating this on your own:

```
$ git clone <URL> <DIR>
```

and <DIR> can equal Wise in this case.

5. Wise can open the file and make a modification.

Example Text to Add: “Wise: This is definitely a text file”

remember to run git add, git commit and git push.

6. Ward can now add a line to the text file but notice he does not see Wise’s change… continue anyway…

Example Text to Add: “Ward: another line of text”

run git add, git commit and WAIT there…

Now Ward wants to push to git hub, but Wise made a change to the file and we don't have that change. What can we do?

7. Run git fetch. This command will actually check the remote repository (github) for any changes. A git status at this point will show us we have an issue.

```
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)

nothing to commit, working directory clean
```

We’ve diverged… OH NO!

No, this is ok… just means we have to merge… or what about rebase…

In the pre-work we used merge. It worked and it can work again here… but when we used merge it left that fork in the history and it just didn't look as clean as we hoped. Isn't there a way to clean this up??? That is where rebase comes in…

8. Before Ward can push we can proactively resolve the conflict we will have with the push and then send a clean result to github. To do this let's look at rebase…

```
$ git rebase origin/master
```

Now at first glance you may think rebase failed, it didn't… it just found the conflict we anticipated.

```
First, rewinding head to replay your work on top of it...
Applying: ward - new line
Using index info to reconstruct a base tree...
M       file1.txt
Falling back to patching base and 3-way merge...
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
Failed to merge in the changes.
Patch failed at 0001 ward - new line
The copy of the patch that failed is found in:
   c:/_repos/ward/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort"
```

9. To resolve the conflict, open the text file and fix the file way you and then save. Next you can run git add. Note you ONLY need to run git add, no need to commit as the rebase will take care of this.

10. With the file in good shape and staged (git add) you can go ahead and run the command to continue to rebase.

```
$ git rebase —continue
```

TADA… You just rebased a change…

11. Now go ahead and push the changes to github

Now you can continue by making a change on Wise. You can also make several changes and then see how those rebase. The idea here is not to push anything until you check your issues ahead of time. This way you can resolve the conflicts if any exist before you encounter them in a push.

As many articles and books will discuss the difference between merge and rebase. Some will even engage in heated debates. My opinion is rebase is great when you aren't dealing with anything that is public. In this context, public means any changes that have been pushed to github. If you have not yet pushed, you can rebase and keep a nice clean linear history. If you push something and someone pulled that down and then you try and rebase you can cause further issues as the person who had the original codebase will not have to deal with your rebase when they try and push their changes. This can create extra work and headaches to resolve.

Exercise 2: Branching

Another piece of git that we want to introduce is the topic of branching. Branching is essentially changing the pointer for commits without affecting the master branch. This means if I want to experiment with something, or test something out I can create my own branch. On that branch I can do my development and see how things work. At the same time everyone else can work with the master branch making updates. When I am done I can determine if I want to save my changes and merge\rebase them back into the master branch or if I want to delete them I can delete the branch and all the commits associate to that branch and not master will not be visible anymore.

To perform our git branching we will use a new repository and perform a few modifications concluding with a rebase back to master.

Steps:

1. Create a new repository on github

2. Clone the repository locally

3. Now you can add a file to the directory to initialize the repository.

In this case you can just add a new text file. Once you add the text file use git add and git commit to save a snapshot.

4. Now you can create a branch to do your work.

```
$ git branch <branchname>
$ git checkout <branchname>
```

or you can do this in a single step using the git checkout like the following:

```
$ git checkout -b <branchname>
```

5. Now you can make modifications to the file on your branch without affecting Master.

Make changes to text file. Run git add, git commit.

Next, run check the master branch out and review the file.

```
git checkout master
```

In reviewing the file you will see your changes are gone. Go back to your branch and they reappear… Magic, huh?!? not really, this is just demonstrating how your branch really does isolate the development. When you are done comparing the file on master and your branch make sure you return to your branch.

```
git checkout <branchname>
```

6. At this point you can repeat Step #5 a few times or move on to merging your changes.

7. Since we have a commit and it’s on our dev branch we created and no one else has worked on this repository we can merge this into master pretty easy.

```
git checkout master
git merge <branchname>
```

That’s it! no conflicts, no problems. We don’t even have an additional merge commit…

No merge commit… but…

This is a case where rebase and merge would have the exact same result. You see, since no one else is working on this project and no one modified master, master was in the same state we left it when we created the branch. Therefore, no conflicts and our commit just comes over to master and master fast-forwards to the latest commit and now both branches are at the same spot.

8. Notice what happens if we not push the repository to github.

```
git push origin master
```

Github only sees the master branch. We didn't push the development branch we created and therefore github doesn't know about it.

9. Now we can delete our branch if we are done developing on it.

```
git branch -d <branchname>
```

Now you may get into situations where you don’t delete branches. This would be where github is also tracking the branch you are working on and it needs to be available to others. In cases where you just create the branch locally and you didn't push it up, you can clean the branch up when you are done and delete it. Just remember to merge\rebase your changes if you want to keep them.