Git Lab

Appalachian State University
Mx. Willow Sapphire
CS 3667

This project will help you familiarize yourself with Git. It is split up into several parts. All of the work will be done in this repository. You should read chapters 1-3 of the Pro-Git book before starting this assignment.

Many of the steps will ask you questions, such as “what happened to your repo?” or “do you understand the output from this command?” You should answer these questions for yourself and, if you do not understand, you should consult the book or ask for help. You do not need to write down the answers to these questions or turn them in anywhere.

Your final deliverable for the project will simply be this repository. If you follow all the steps, then I will be able to see everything and you do not need to take any further steps to turn in the assignment.

Table of Contents

Part 0: Understanding the Repo

Take a look at our folders and files. We have a readme file, a src folder, a docs folder, and a bin folder. This will be the general setup for our repos in this class. Let's walk through what each of these is.

- **readme.md**: This file gets displayed on the GitHub Repo main page. It should describe the repository and have any important information about it.
- **src**: This is the folder where our code goes (src stands for source as in source code).
- **src/.gitkeep**: .gitkeep is a file that exists only to keep a folder in the repository. If a folder is empty, it will not be stored in git. Once you add code to the src folder, you can delete this file.
- **bin**: This is the folder where our compiled code goes (.class files). We don't want those clogging up our src directory so we send them here.
- **bin/.gitkeep**: This .gitkeep file functions the same as the one in src, but we do NOT want to delete it. We will keep our .class files out of the repo since they are generated whenever the project is compiled and don't need to be saved. Therefore, this folder will always be empty in the repo. We still need it to be there since

our command to compile code won't work if it doesn't exist

- **docs**: This is the folder where any relevant documentation about our project goes. We could have outlines, diagrams, and other such materials here.
- **docs/Instructions.pdf**: That’s this file! It tells you what to do in this project. It’s in the docs folder since it could be considered documentation about what is in the repo.

Part 1: Individual Repo

Step 1: Branching

We always want to work on a different branch than main. Never code directly on the main branch.

1.1) Make a branch named hello

```
git branch hello
```

1.2) Checkout this branch

```
git checkout hello
```

1.3) Run `git branch -a`. Do you understand the output?

Step 2: Writing Code

It's usually nice to stay in the root folder since commands work best from here. So as we edit and run code, we will not cd into other directories.

1) In the src folder, create a file named Hello.java

```
touch src/Hello.java
```

2) Write a simple hello world program in Hello.java

```
vim src/Hello.java
```

3) Compile your program. The d flag stands for destination and tells the compiler where to put the .class files.

```
javac -d bin src/Hello.java
```

4) Run `ls bin`. You should see that you now have the class file in the bin directory.
5) Run your program. The cp flag stands for class path. It tells java where the files we are using are located. In this case, our compiled files are in the bin directory.
   `java -cp bin Hello`
6) Run `git status`. Do you understand the output?

Step 3: Ignoring Files

Now that we have our working program, we want to add it to our repository. However, we do not want to add the .class file since it is unnecessary. It will get generated when the project is compiled, so it doesn't need to be in the repo. We can ignore other files too. For example, if you work in vscode, it will sometimes add a vscode folder. You wouldn't want to add that to your repository so it would also go in the .gitignore.

1) Make a new file name .gitignore in the repository root
   `touch .gitignore`
2) Add the line bin/**/*.class to the .gitignore file. This tells git to ignore any .class files in the bin folder and any subfolders
   `vim .gitignore`

3) Add the gitignore file to the staging area and commit it so that our repository knows what to ignore. We'll talk more about these two commands in the next step.
   `git add .gitignore`
   `git commit -m "added gitignore file"`
4) Run `git status` again. Do you see what changed? Do you understand why?

Step 4: Adding Files

Now that we are ignoring the files we don’t want to add, let’s go ahead and add the files we do want to add. In this case, we have 1 new file: Hello.java

1) Add Hello.java to the staging area. Note that this does not add it to the repository yet.
   `git add src/Hello.java`
2) Run `git status`. Do you see and understand what changed?
3) Commit your changes to the repository. This puts everything from the staging area into the repository (in this case, just Hello.java). The -m flag is required and is used to add a message to the commit. The message should be descriptive of the changes you are committing.
   `git commit -m "created hello world program"`
4) Run `git status`. Do you see and understand what changed?

Step 5: Merging Branches

We have finished writing our code on the hello branch and we have checked that it works properly. So now we want to add it to the main branch.

1) Switch back to the main branch
   `git checkout main`
2) Run `git status`. Do you understand the output?
3) Look into the src folder with `ls src`. What changed? Do you understand what happened?
4) Run `git log --oneline --decorate`. Do you understand the output?
5) Merge the hello branch into main
   `merge Hello`
6) Run `git status`, `ls src`, and `git log --oneline --decorate` again. What has changed? Do you understand why these changes occurred?

Step 6: Pushing to GitHub

Now that we have updated our local repository with our changes, we want to put those changes to GitHub. There are three main reasons for this.

- It serves as a backup in case something happens to our computer.

- It allows us to immediately start working from a different machine by simply pulling the code from GitHub.
- It allows other developers to see and use our code.

1) Let's push our main branch to GitHub. GitHub already has the main branch and we have already merged our updates into main.
   `git push origin main`
2) Go look at GitHub. You should see that there is a new commit on the main branch. The files you created are all there now!
3) Run `git log --oneline --decorate` and `git status` again. Do you understand why the outputs are what they are?

Part 2: Shared Repo

In Part 1, you learned the basics of working in a repository. However, there were a couple things we did that are usually only appropriate when you are working on a repo by yourself. Often we are working with others and want to do things a little differently. Namely:

- We do not merge to main on our local machine
    - We can still merge other development branches together as we will see a bit later in this lab
- We do not push code directly to main. Instead we make something called a pull request.

Step 1: Doing Some More Work

Edit your Hello.java file to use a Scanner to ask the user for a name, then print out "Hello <name>" instead of "Hello World"

1) Make a new development branch named ask-name
2) Checkout the new branch
3) Make your changes
4) Compile and test your code
5) Add your changes to the staging area
6) Commit your changes to the repository

Step 2: Pushing Our Changes

Now you are done and ready to move your code into main. However, we want to let other developers check our work. Pushing to main on our own can be dangerous since other developers won't know what changes we are making and bugs can more easily slip by if no one is checking our work.

1) Instead of switching to main and merging our branch, we will push our development branch to GitHub. The -u flag tells GitHub that this is a new branch that is not already stored on GitHub. If we are pushing to a branch that is already there, we would omit this flag.

   ```
   git push -u origin ask-name
   ```

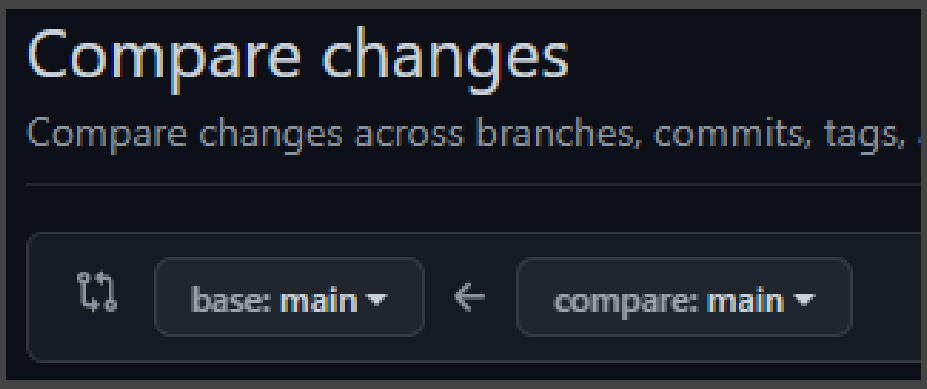
2) Now go to your repository GitHub and, in the upper left corner, you should see a dropdown that currently says main. If you click on it, you should see that your ask-name branch is also there (you may need to refresh the page). Click on it and then look at the Hello.java file on GitHub. You should see that the changes are there.

Step 3: Making a Pull Request

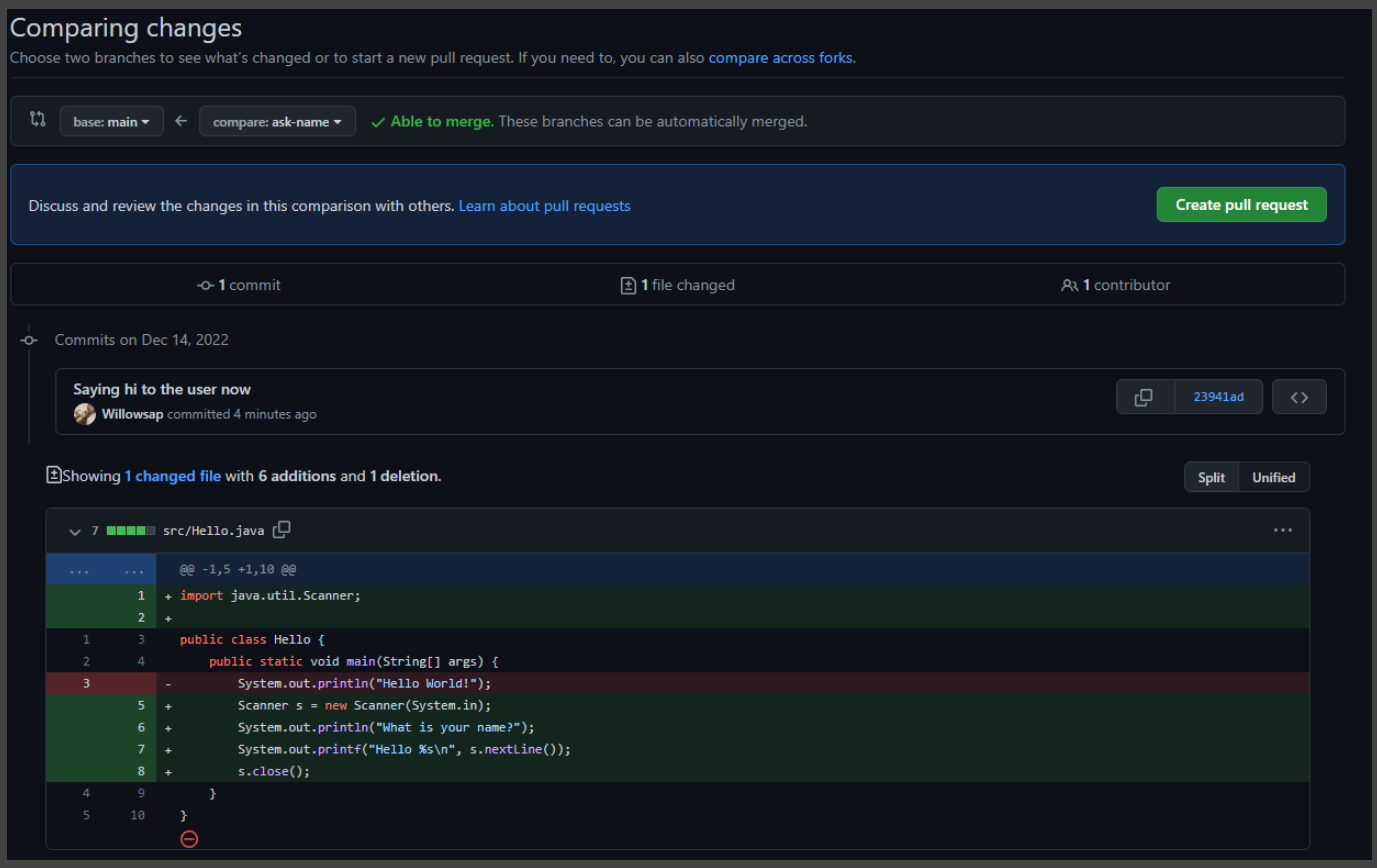
You will likely see a prompt at the top of your GitHub page telling you that you recently pushed a new branch and suggesting that you make a pull request. This is exactly what we are going to do. However, we will not go through

this prompt to do it since we want to learn how to make pull requests at any time, not just when GitHub suggests it.

1) Click on the Pull Requests tab at the top of the page.
2) Click on the New Pull Request button in the upper right corner. You should see something like this in the upper left side of the screen.



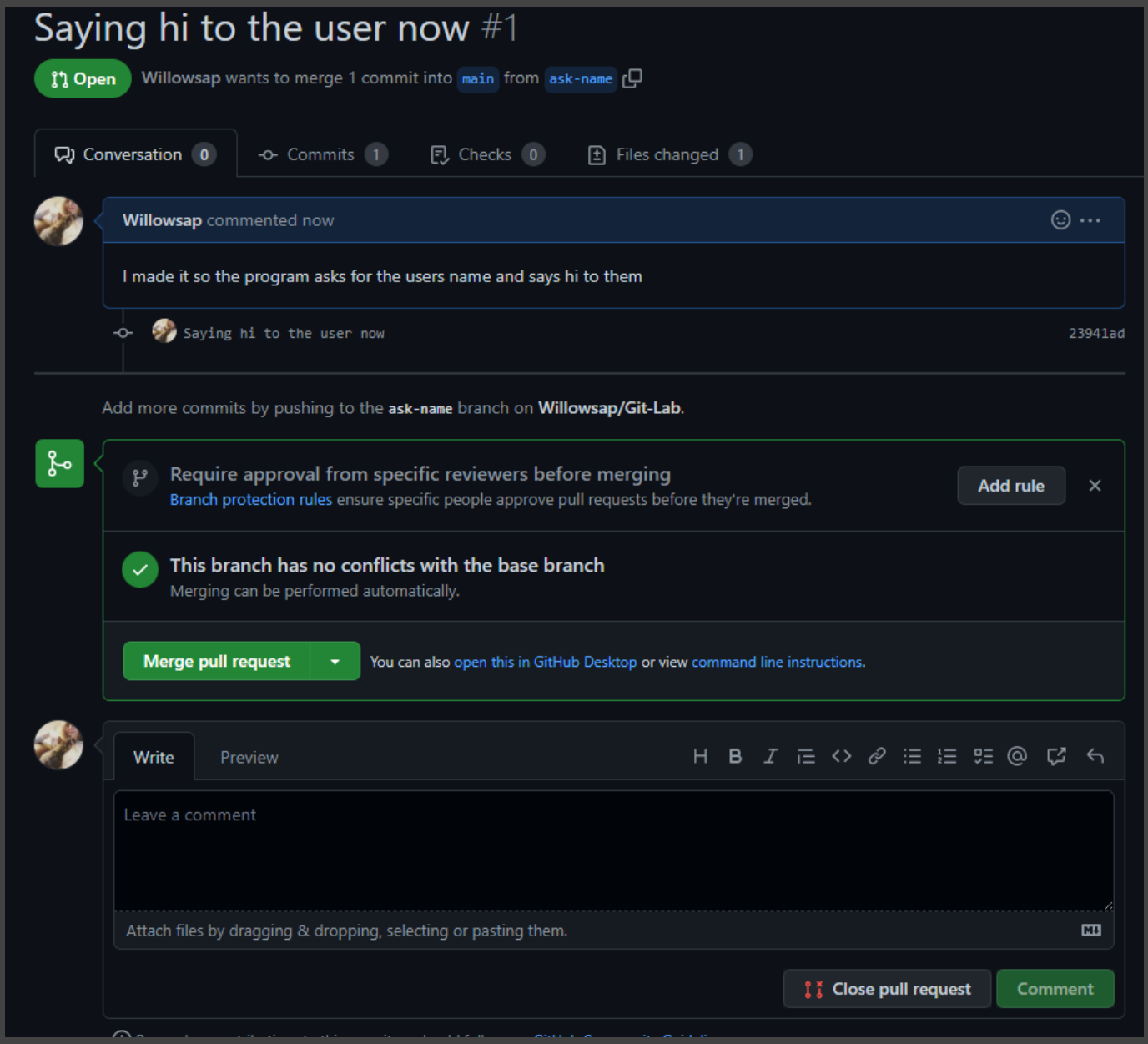
3) Click on the dropdown on the right that says compare: and select ask-name. The screen should then look something like the following image. Of course, depending on how you wrote the program, the changes to the file might look different.



4) Click the Create pull request button
5) You should then see a page that shows a title for the pull request, a text box to leave a comment, and a button that says Create pull request. Write something in the comment box like "I made it so the program asks for the users name and says hi to them"
6) To the right you should see some options: Reviewers, Assignees, Labels, Projects, and Milestones. Click on Reviewers and add me, Willowsap. This notifies me that you created a pull request and would like me to review the changes you are making.

7) Click the Create pull request button

8) You should now see a page that looks something like this:



9) Click around the tabs at the top to see what they all are. Do NOT click the Merge pull request button. As a general rule, we never merge our own pull requests when working in a group. We want other members to merge them to ensure that our code was checked. Sometimes other

members might leave comments and say that it is good to merge, in which case you could merge it. That is what we will do in this scenario.

10) When you make the pull request, I will take a look at it and make a comment. Hopefully my comment will say that you can go ahead and merge it. I'm letting you merge it instead of doing it myself so that you get to do all the steps.

11) Once I give the okay to merge, you can go back to the pull request and click Merge pull request.

12) We can see that there are no conflicts with main, so GitHub can merge the branches on its own. Sometimes there are conflicts that we have to fix, but that is a topic for another time.

Part 4: Conflicts

When multiple people are working on projects, conflicts are inevitable. A conflict is when multiple people edit the same file in the same place and git does not know how to merge the changes.

Step 1: Making More Changes

1) Make a new branch called ideas
2) Make a file named next-steps.txt in the docs folder
3) In that file, write:
   Have the program have dialogue with the user.
   Have it ask more questions and receive answers.
4) stage and commit your new file
5)

Step 2: Evil Twin

In order to practice conflicts, we have to pretend that we have someone else working on our repo. We can simulate this by cloning our repo into a new directory.

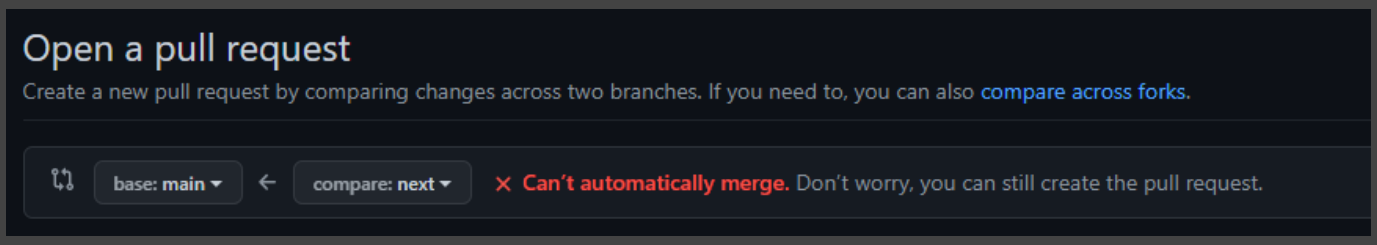
1) On the student machine, go out of your repository directory and make a new folder named evil-twin. Then cd into that directory and clone your repository.

2) For this part we will work directly on main, partly to be quicker but also because this is your evil twin and working directly on main is evil.
3) In this new version of our repository, make the next-steps.txt file (it won't be there since we never pushed it.).
4) Int this version of the file, write:
   Ask for more information about the user
   Sell info to advertisers
5) stage and commit this file.
6) push the changes to GitHub (you are pushing directly to main)

Step 3: Back to You

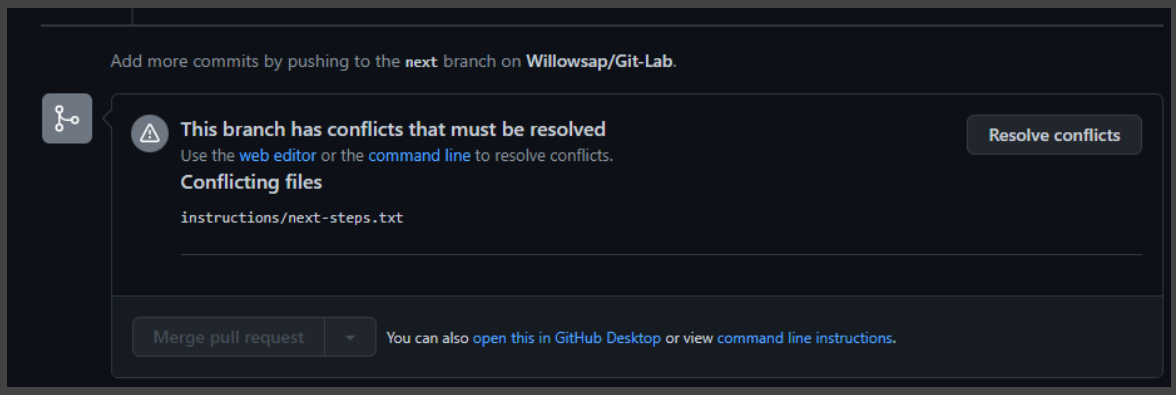
Now you are ready to merge your changes. You are going to do it the right way though.

1) push your local development branch (next-steps) to GitHub.
2) Go to GitHub and start to open a pull request to merge this branch into main.
3) Uh oh! You should see something like this:

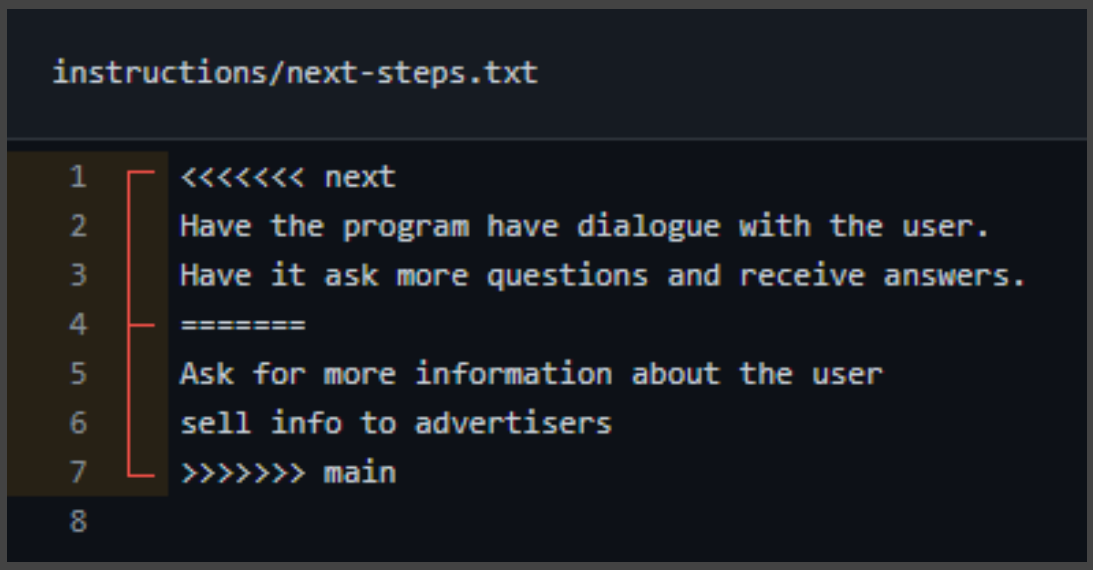


4) We can see already that git has detected a conflict. Let's go ahead and open the pull request anyway. We can resolve conflicts there.

5) On the pull request page, you should see a box that looks like this:



6) Click the Resolve conflicts button. It will open up a page that has something looking like this:



7) This is showing you where the conflicts are. At the top we can see the name of the file that contains the conflicts (note that yours will be docs/next-steps.txt, not instructions/next-steps.txt).

The incoming conflicting code is denoted by

<<<<<<<<<<< incoming-branch-name

The end of the incoming conflicting code and start of the conflicting base code is denoted by

==========

The end of the conflicting base code is denoted by

>>>>>>>>>>>>> base-branch-name

All of this text is actually in the next-steps.txt file! You can resolve conflicts on your local machine as well and you would see this same thing inside of the file with the conflicts.

8) To resolve the conflicts, we need to edit the file to keep what we want and remove what we don't. Of course, we will always want to remove the lines that were added by git. If this was a java file, those lines would make our code not compile.

9) Edit the file to keep only the text you want. Delete the lines added by git. You can decide which other lines you want to keep. You could also add more lines here if you wanted to.

10) Once you have fixed the file, click the Mark as resolved button and then the commit button.
11) You will now be back at the pull request page and should see that the conflicts are gone.
12) Add me as a reviewer to the pull request if you didn't already and do not merge it

Wrapping Up

So we have learned a lot here. We can branch, write files, stage files, commit files, push files, push new branches, merge branches, create pull requests, resolve conflicts, and merge pull requests. These are the basic building blocks of git that you need to know. However, they are only scratching the surface of what we can do. As we work in git throughout the semester, you will likely encounter new problems that we haven't gone over. Use resources like DangItGit, the Pro-Git book, and stack overflow to see if you can solve these problems. Most developers aren't experts at git, but they are experts and looking up how to solve problems in git when they encounter them.

Finally, you do not need to do anything else to turn in this assignment. Your repository is your product.