**08 – Upstreaming in FOSS**

**Activities**

COMP190 – Tools and Techniques for Software Development

Dickinson College

**Name:**

Top of FormThis week’s topic focused on upstreaming. We saw a typical workflow for how code and documentation contributions can be contributed back to an upstream repository on GitHub. While different projects will use different workflows, the one we saw in class and the one you will use through these activities is based on GitHub Flow and is quite common.

* See <https://guides.github.com/introduction/flow/> if you are interested in GitHub’s description of the GitHub Flow workflow.

These activities will building on A07 by having you apply the workflow to fix the issue that you claimed and then upstream the change that you make. We will complete the first parts of this activity in class. The activity then asks you to repeat the process by claiming another issue, fixing it and making a pull request (PR) to upstream your changes. We’ll use those PR’s in the next class to see what happens when multiple developers working asynchronously make conflicting changes.

**Current State:**

Figure 1 shows where you should be at the end of A07. You will have forked the upstream FarmData2 repository into your own GitHub account. You will then have cloned your fork into a local repository on your computer. Your local files at this point are a copy that reflects the current state of the upstream main branch. Let’s explore this state just a little before making the changes that address the issue that you have claimed.

Figure 1 - Fork and Clone

1. The git log command provides a way to see information about the history of the project. It will display information about the most recent commits that have been made to the repository. By default, git log displays information about the 10 most recent commits. You can append a -1 or -5 to show only the most recent or most recent 5 commits.

a. Use the git log command to display the information about the two most recent commits. Give the command that you used and the output that it produced.

b. A line that begins with \* commit appears at the start of the information about each commit. That line also contains a long string of numbers and letters. That string is called the SHA (secure hashing algorithm) hash. It is a hexadecimal number that is computed from the changes contained in the commit and is the unique identifier for the commit. Following the SHA hash is information about the author, the date the commit was made and the commit messages that were used to describe the changes.

Use the output from part a to answer the following questions about the project history:

i. What is the SHA for the most recent commit.

ii. When was the most recent commit made?

iii. Who made the second most recent commit?

iv. What was the stated purpose of the second most recent commit?

**Create and Switch to Feature Branch:**

When you set out to make changes to the project you will do so by working on a feature branch. Figure 2 illustrates the process of creating and switching to a feature branch. The activities in this section will walk you through the process of creating a feature branch on which you will fix the issue that you claimed.

Figure 2 - Create and Switch to Feature Branch

2. *Explain git status, run it, report the output, ask a question about which branch is active. Ask what they think it means when it says “Y﻿our branch is up to date with 'origin/main'.”? Ask what they think it means that “﻿nothing to commit, working tree clean”? maybe Hint: recall that commits describe changes that have been made.*

3. *Explain git branch <name>, ask for a command that makes a feature branch for fixing their issue. Suggest using a short but descriptive name (not Issue #123). Ask for the output of git status. Ask which branch are they on? How can they tell.*

4. *Explain that git branch doesn’t change branches. Explain git switch and ask for a command that changes to their feature branch. Ask for output of git status. Ask which branch they are on and how they can tell.*

**Edit Local Files:**

You have created a feature branch and switched to that branch and you are now ready to fix the issue that you claimed. This state is illustrated in Figure 3. The local files started as a copy of the last commit on the main branch, but have been modified as indicated by being drawn in blue (as opposed to green as they were in main).

Figure 3 - Editing Local Files on a Feature Branch

5. *What is the number and title of the issue that you claimed? In which file will you be making changes to address that issue? Hint: Look at the text of the issue in the issue tracker.*

6. Using a text editor of your choice (nano, vi, gedit, etc) modify the file you identified in #5 so that the issue you claimed has been fixed. If your issue asks you to correct a typo, simply change the indicated text and save the file. If your issue asks you to add a link, the syntax for that is as follows:

[Link text in square brackets](URL in parenthesis)

For example:

[Dickinson Home Page](<https://www.dickinson.edu>)

No Answer Required: Making the change to the file is all that is necessary for this question.

7. *Use the git status command to answer the following questions: What output does git status produce now? Ask something about the output that has them identify what has been changed, that is a modification and that it has not yet been staged.*

**Stage Changes and Commit:**

As you saw in question #7 you now have modifications to your local files that have not been staged or committed to your local repository. This process of staging (i.e. adding the files to the stage) and then committing them to your local repository is illustrated in Figure 4. The activities in this section will have to first stage your changes and then commit them to your local repository.

Figure 4 - Staging Changes and Committing

8. *Explain the git add <file> command. Ask for a command that stages their changes.*

9. *Ask for the output of git status and ask a question about the difference from what was seen in question #7 and what still has to be done.*

10. As you saw earlier when looking at the output of git log, each commit has a *commit message* that briefly describes the changes that are contained in the commit. These messages should be concise but meaningful without referring back to the issue. That is, some future reader of the git log should be able to get an idea of the changes you have made by reading your commit messages.

*Make up two issues and give three commit messages for each (best, worst, okay). Ask them to rank the three. This should be pretty easy, the idea is just to make them do this so that when they write their own they think a little bit. Maybe something like“fixes issue #123” (worst) or “Fixed typo” (okay) or “Made the word bug plural (i.e. bugs)” (best).*

11. *Explain the git commit -m “<message>” command. Ask them to give a command that will commit their changes to the local repository. Ask for the output that is generated and*

12. *After running your command from #11, give the output of git status. Ask something about the output that gets them to think about what has changed from the output in #9 and how it connects to the bigger picture.*

13. *Give the output of a git log command that shows the three most recent commits. Ask a question(s) about how and why it has changed from what they saw in #1. Maybe ask an extra question that connects the SHA to the snippet of the SHA shown in the output of 12?*

14. *Have them switch to main and look at git log again. Have them look in the file for their change. Ask a question here where they need to explain that their changes have been committed to the feature branch and not to the main branch.*

15. *Have them switch back to their feature branch and confirm with git status.*

**Push Branch to Origin:**

You have now made the changes necessary to address an issue in the project and are ready to upstream those changes. In order to do that you need to get the changes you have made into GitHub so that the upstream maintainers can see and review them. As shown in Figure 5, this is done by *pushing* your feature branch to GitHub.

Figure 5 - Push Feature Branch to Origin

16. *Explain the git branch -l (ell not one) command. Have them list the branches that appear in their local repository.*

17. *Compare that list of branches to those that appear in GitHub and explain the difference.*

*Getting a Personal Access Token:*

Before you will be able to push your feature branch to GitHub you will have to get a Personal Access Token (PAT) from GitHub. A PAT is like a password but has some security benefits. In particular you can have multiple different PATs. Each one can have different limits on what can be done with it and each one can be revoked without affecting the others. It is not required, but if you want to learn more about why GitHub uses PATs you can read this page:

* Need a Link here…

18. *Create a PAT. Need to give directions or a link to directions. Also implore them to copy and save it somewhere secure.*

19. When interacting with GitHub using the git CLI you will be asked for your PAT anytime GitHub required authentication (e.g. when pushing a branch). Clearly you don’t want to try to type your PAT every time. Fortunately, you can configure git so that it will remember your PAT and automatically provide it to GitHub when required.

Make sure that you are in the directory for your FarmData2 repository and issue the following command:

git config --global credential.helper store

Give the output of the following command to confirm that your added configuration was successful:

git config --global --list

*Pushing your Feature Branch:*

20. *Explain git push origin main. Note that they need to paste their PAT as the password. Ask for output of the command and ask a question about it.*

21. *Ask for the output of git status and ask a question about how it has changed from the output in #12 and why.*

22. *Connect back to #17 somehow here to have them see that the push put the branch onto their GitHub.*

**Make Pull Request:**

A pull request is the mechanism by which you ask the maintainers to consider merging your changes into the upstream main branch. As shown in Figure 6, a pull request is made from your origin repository, which contains your feature branch, to the upstream repository. With your feature branch successfully pushed to your origin repository you are now ready to make a pull request.

Figure 6 - Making a Pull Request

23. There are a number of different ways to make a pull request in GitHub. In general, they are all equivalent and it won’t matter which one you use. The following steps will walk you through one of those ways.

a. When you first push a feature branch, GitHub assumes that you are likely to make a pull request for it. To make this easy it will automatically display a big green “Compare & pull request” button at the top of the page.



Click that button. You will use the page that appears to create your pull request.

b. A gray bar similar to the one shown below should appear at the top of the page. This bar specifies what you are requesting to be pulled and where you are asking it to be merged.

i. Is your upstream listed as the *head repository* or the *base repository*?

ii. Is this pull request asking that your changes be merged into the head repository or the base repository?

iii. Where does this pull request indicate the branch that contains your changes?

iv. Where does this pull request indicate the branch of the upstream into which you are asking that your changes be merged?

*c. Give your PR a descriptive title – similar to your commit message.*

*d. Replace the <add description> with a longer description of your changes.*

*e. Explain Closes and have them Add “Closes #123” to link to the issue.*

*f. At the bottom of the page there is a “diff” showing the upstream version of the code and the changes you have made. Ask some question about this. What line number did you change? Is your code shown on the left or right? Something more creative?*

*g. Confirm that your PR can be merged automatically.*

24. *Go to the “Pull Requests” tab, find your PR. Give the number, title and URL for your PR.*

**Merged into Upstream:**

If the upstream maintainers decide that your changes are good for the project, they will merge them into the upstream main branch. That way they will become a part of the main project, and everyone can benefit from your work. In class your instructor will play the role of an upstream maintainer and demonstrate how the changes contained in pull requests are merged.

Figure 7 – Changes Merged into Upstream

Figure 7 illustrates the state after the commit from your feature branch (blue) has been merged into upstream main.

25. Notice that the upstream main branch also contains another commit (pink). This wasn’t part of your feature branch. Briefly explain how that commit likely got added to upstream main.

26. You can see this using git as well.

a. Switch to the main branch of your local repository. What command did you use?

b. Use the git status command. What output is generated?

c. *Status output will say something about your branch being some number of commits behind the upstream. Ask something about what they think that means and how it happened.*

**Synch with Upstream:**

As changes from pull requests (yours and others) are merged into upstream main, your main branches will get *out of synch with the upstream*. Recall that the upstream main should be the starting point for all new work. So, before you can work on something new you’ll need to make your main branches look like the upstream main (i.e. you will synchronize them with the upstream). This is a two-step process. You will pull the upstream main into your local repository and then you will push it to your origin. Figure 8 shows how pulling from the upstream and pushing to your origin synchs your main branches.

Figure 8 - Synch with Upstream

*Setting Upstream Remotes:*

In order to pull changes from the upstream to your local repository you will need to tell your local repository where the upstream is. In all of the earlier figures dotted orange lines represented that your local repository knew about the origin and the origin knew about the upstream. But there was no dotted line from your local repository to the upstream. That’s because by default, your local repository isn’t aware of the upstream.

27. Notice new orange dotted line from your local repository to the upstream that appears in Figure 8. You need to create that link to be able to pull from upstream.

*a. Give the output of git remote -v*

*b. Which orange dotted line are represent by the two lines of output.*

*c. Set the upstream remote using the command…*

*git remote add upstream <Upstream Repository URL>*

*d. Give the output of a git remote -v*

Note that you will only need to set the upstream remote once for a local repository. Once it is set you will be able to pull from the upstream as often as is necessary.

Pulling from Upstream:

28. *Make sure on the main branch.*

*a. Give the output of git status*

*b. Do a git pull upstream main. Give the output.*

*c. Ask a question about the output.*

*d. git status give the output*

*e. ask a question about the output.*

*f. Confirm that your changes are now contained in the main branch. Note that there will be others as well.*

Push to Origin:

*29. Now let’s push the updated main branch to your origin.*

*a. git push origin. Give the output.*

*b. Ask a question about the output.*

*c. Check to see the changes in the main branch on your origin.*

As with most things related to git and GitHub, there are a number of different ways to synchronize. For example, you can also fetch the changes from the upstream main to your origin and then pull them from there to your local repository. If you are curious, you can check out this GitHub: link:

* <https://docs.github.com/en/github/collaborating-with-pull-requests/working-with-forks/syncing-a-fork>

**Delete Feature Branch:**

Once your changes have been merged into the upstream main there is no need to retain your feature branch. There is no harm in keeping it. But most developers will delete them to avoid having their repos become cluttered with old feature branches. Figure 9 shows the state that would result after deleting the feature branch.

Figure 9 - Deleting Featrue Branches

30.

*a. Give the output of git branch -l*

*b. use git branch -D <branch>*

*c. Give the output of git branch -l*

31. In the previous question you deleted your feature branch from your local repository.

a. Did this also delete that branch from your origin repo on GitHub? How did you check?

b. You can delete your feature branch from your origin repo using the GitHub user interface in the browser. Or you can deleted it using the git CLI. Use the command git push -d origin <branch>

c. Confirm that your feature branch is deleted from GitHub.

**Command Summary:**

It will take a while for this process and all of the commands to become a natural part of the way you work. Until that happens it is often useful to have a short concise cheat sheet of the command for accomplishing each task.

32. Complete the table below by filling in the right-hand column with the commands that accomplish the task listed on the left. Use the <…> notation appropriately to indicate parameters that need to customized for each use. Note that the tasks listed are in approximately the same order as they appear in this activity.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Task to Complete** | **Git Commands** |  |
|  | Display recent commits to the active branch. |  |  |
|  | Create a feature branch. |  |  |
|  | Change branches (i.e. to a feature branch or to main) |  |  |
|  | Check the current state of your local repo. |  |  |
|  | Stage changed files to be committed to the repo. |  |  |
|  | Commit staged files with a message. |  |  |
|  | Push a feature branch to your origin. |  |  |
|  | Delete a feature branch from your local repo. |  |  |
|  | Delete a feature branch from your origin. |  |  |
|  |  |  |  |

**Do it All Again:**

*33. Pick a 190-round2 issue. Only a few so multiple of you will work on each one. Give the Issue # and title*

*34. Give them a list of the steps they need to do.*

*Maybe just ask for the commands used at a few points (e.g. branch, add, commit, push)*

*No need to ask a lot of questions here.*

*Last one is Make PR and confirm that it can be merged automatically*

*35. Ask something about what will happen if someone else was working on the same issue as you, changed the same line as you and then the maintainers merge their solution into main?*

**Optional:** To help us improve and scope these activities for future semesters please consider providing the following feedback.

a. Approximately how much time did you spend on this activity outside of class time?

b. Please comment on any particular challenges you faced in completing this activity.

**Acknowledgements:**

Some materials, questions and resources have been adapted from activities posted on foss2serve.org:

* <http://foss2serve.org/index.php/Git:_Cloning>
* <http://foss2serve.org/index.php/Git:_Git_Intro_Activity>
* <http://foss2serve.org/index.php/Git:_GitHub_Issues_and_Pull_Requests>
* <http://foss2serve.org/index.php/Git:_GitHub_Workflow_Activity>
* <http://foss2serve.org/index.php/Intro_to_GitHub_(Activity)>
* <http://foss2serve.org/index.php/Version_Control_(Activity)>
* <http://foss2serve.org/index.php/Work_Locally_with_Git_from_the_Command_Line_(Activity)>