**2 – Working Locally and Upstreaming**

**Activities**

**Name:**

Top of FormThis topic focused on working in your local repository using branches and commits and then upstreaming your changes using a pull request. In class we saw a typical workflow for how branches are used to make changes to code and documentation, how those changes are committed to the branch and then how those contributions can be contributed back to the 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 a basic Forking Workflow that is quite commonly used (or adapted) by FOSS projects.

* See <https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow> if you are interested in Atlassian’s description of the forking workflow.

These activities will build those from the previous topic by having you apply the workflow to fix the issue that you claimed and then upstream your changes by making a pull request.

**Getting Started Again:**

1. Like the last activity, you will be working within the *KitClient* for this activity.

a. What is the docker command used to start the KitClient? Note: Not the one to create it, but the one to start it.

b. Use the command from a and connect to the KitClient using the TigerVNC viewer (or noVNC via your browser).

Nothing is required here. But be sure you can connect to the KitClient as you’ll be using it throughout this entire activity.

2. What is the docker command used to stop the Kit Client?

**Current State:**

Figure 1 shows where you should be at the end of the previous set of activities. 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 - Fork and Clone

3. 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 -2 (or -5) to show only the most recent 2 (or 5) commits.

a. Ensure that you are in your cloned repo (i.e. the GitKit-FarmData2 directory is your working directory). Then use the git log command to display the information about the 3 most recent commits.

Give a screenshot of 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 provides a unique identifier for the commit. Following the SHA hash is information about the author, the date the commit was made and the commit message that was used to describe the changes.

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

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Information** | **Value** |  |
|  | SHA of most recent commit. |  |  |
|  | Date of most recent commit. |  |  |
|  | Author of 3rd most recent commit. |  |  |
|  | Purpose of 3rd 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 - Create and Switch to Feature Branch

4. The git status command provides the current status of your local repo. Use the

git status command to see the current status of your local GitKit-FarmData2 repository.

a. Give a screenshot showing the command you used and the full output that is generated.

b. What part of the output tells you which branch currently active in your repo?

c. Your output in part b should contain the text “Y﻿our branch is up to date with 'origin/main'.” What do you think it means that your branch is “up to date”?

d. You output in part b should contain the text “﻿nothing to commit, working tree clean.” What do you think it means that you have “nothing to commit.” Hint: why would you have something to commit? This is the opposite of that!

5. The git branch <name> command creates a new branch with the given name. When creating a branch, you should give it a short but descriptive name (e.g. “FixTypoInReadme”). Note: <name> may not have spaces, so you can use - (dashes) or CammelCaseText to divide words in your branch names.

a. Create a new feature branch with a descriptive name that you will use to fix the issue that you claimed. Remember not to include the < > when writing your own command.

Give a screenshot of the command you used and its output.

b. Use the git status command again now.

Give a screenshot of the command you used and its output.

c. Examine the output in part b. Does creating a new branch with the git branch command change you to that branch? How can you tell?

6. As you have just seen, the git branch command creates a branch (among other uses), but it does not change your active branch. The git switch <name> command switches (i.e. changes) the active branch.

a. Give a command that will change your active branch to the new feature branch that you just created.

b. Use the git status command again now.

Give a screenshot of your command and its output. Highlight the part of the output that confirms that your feature branch is now the active branch.

**Edit Local Files:**

In the last section you *created a feature branch* and *switched to that branch*. In this section will *edit your Local Files* to fix the issue that you claimed in the issue tracker.

7. What is the number and title of the issue that you claimed in the issue tracker?

Figure 3, now illustrates the situation when the Local Files have been edited. Back in Figure 2, the Local Files in your feature branch were drawn in green. This reflected that they were, at that time, an exact copy of the last commit on the main branch, which was also drawn in green. Here Figure 3, Local Files are drawn in blue, to indicate that they have been changed as compared to the final commit in the main branch, which is still green.

Figure - Editing Local Files on a Feature Branch

8. Now it is time to make the changes to your local files in order to fix the issue that you claimed in the issue tracker.

a. In which file will you be making changes to address your issue? Hint: Look at the text of your issue in the issue tracker.

b. Using a text editor modify the file you identified in part a so that the issue you claimed has been fixed. Be sure to save your changes. Note: You can find a number of different text editors in the KitClient. The launcher panel at the bottom of the desktop contains icons for the Mousepad and VSCodium editors. Other editors such as vi, nano, gedit and emacs can be launched from a Terminal window.

Nothing is required here. Making and saving the change to the file is all that is required for this question.

9. The git status command can now be used to check that your changes have been made.

a. Run the git status command now.

Give a screenshot of the command and its output.

b. What *two things* does the output in part a tell you about the file that you edited? If you see the message “﻿nothing to commit, working tree clean,” it means that your edits in #8b were not saved. Try question #8b again.

10. The git diff command provides another way to examine or confirm the changes you have been making. Note *diff* is just short for *difference*.

a. Use the command git diff. Give a screenshot of the command and its output here.

b. How is the change that you made indicated in the output from git diff?

**Stage Changes and Commit:**

Figure - Staging Changes and Committing

As you saw in question #9 you now have modifications to your local files that have not been staged or committed to your local repository. The process of *staging changes* (i.e. adding the files to the *stage*) and then *committing staged changes* to your local repository is illustrated in Figure 4. Recall from Figure 3 that the blue dot in the Local Files represents the changes you made. In Figure 4, those changes are staged and then made into a commit that is added to your feature branch.

The activities in this section will walk you through staging the changes you made to your Local Files and then committing them to your local repository.

11. The git stage <file> command adds the indicated file to the stage, preparing it to become a part of the next commit.

a. Use the git stage command to stage the file that you edited.

Give a screenshot of the command you used and its output.

b. Note that git also has a git add command that is equivalent to git stage. So, you can add files to the stage using either git stage or git add. These activities will use git stage because it seems more descriptive of what is happening. However, you are likely to see git add used in other resources, so it is worth knowing that they are equivalent.

Give the git add command that would be equivalent to your answer in part a.

12. Now use the git status command again. What two changes have occurred in the output of the git status command as compared to #9b to reflect that your changes have been staged?

13. 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 requiring the reader to refer to the ticket in the issue tracker.* That is, some future reader of the git log should be able to obtain an idea of the changes you have made and why you made them by reading your commit messages.

For each of the following issues, rank the given commit messages 1, 2, 3 with 1 indicating the best message and 3 indicating the worst.

a. Issue: The documentation says “bug” instead of “bugs” where plural is needed.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Commit Message** | **Rank** |  |
|  | Fix issue #123 |  |  |
|  | Pluralize bug (i.e. bugs) for clarity |  |  |
|  | Fix typo |  |  |
|  |  |  |  |

b. Issue: The harvesting log should be able to track insect presence.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Commit Message** | **Rank** |  |
|  | Extend harvest logs |  |  |
|  | Insect tracking added |  |  |
|  | Add tracking for insects in harvesting logs |  |  |
|  |  |  |  |

14. The git commit -m "<message>" commits all of the staged files to the currently active branch with the specified commit message. Use the git commit command to commit your staged changes to your local repository *with a meaningful commit message*.

Give a screenshot showing your command and its output.

15. Now use the git status command again. What change has occurred in the output of the git status command as compared to #12 to reflect that your changes have been committed?

16. Use the git log command to show the 3 most recent commits to your branch. How is this output different from what you observed in Question #1? Briefly explain why?

17. Switch to your main branch.

a. What command did you use?

b. Does the output of a git log command now show the information about the commit you made? Briefly explain why or why not?

**Push Branch to Origin:**



You have made the changes necessary to address the issue you were assigned, and you are now ready to upstream those changes. In order to do that, the first step is to get the changes you have made into GitHub so that the upstream maintainers can be told about them. As shown in Figure 5, this is done by *pushing your feature branch* to your origin repository. More generally, pushing a branch, copied it from a local repo to a remote repo.

Figure - Push Feature Branch to Origin

18. The git branch -l (lower case L, not one) command lists all of the branches that exist in your local repository. What branches are there in your local repository?

19. Use your browser to look at your origin repository on GitHub. What branches appear in your origin repository on GitHub? Why is there a difference between the branches in your origin repo and your local repo?

*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.

When interacting with GitHub using the git CLI you will need to use a PAT, instead of your password, anytime GitHub required authentication (e.g. when pushing a branch).

20. Follow GitHub's instructions for creating a (classic) personal access token: (<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token#creating-a-personal-access-token-classic>).

When doing so be sure to:

* Set an expiration date that is after the end of the semester.
* Choose “repo” for the “scopes and permissions.”
* Be sure to copy and paste your PAT somewhere safe – maybe e-mail it to yourself. You will not be able to retrieve it again after you leave the page.

Nothing is required here. You just need to be sure to have created a PAT.

Note: Typing or even copy and pasting your PAT every time it is needed can be a hassle. To help, the git CLI provides what is known as the *credential helper*. The git CLI in the KitClient has configured the credential helper so that your PAT will be stored when you enter it the first time. It will then be used automatically for future commands. If you are curious, you can learn more about the credential helper here:

* <https://techexpertise.medium.com/storing-git-credentials-with-git-credential-helper-33d22a6b5ce7>

*Pushing your Feature Branch:*

21.The git push <remote repo name> <branch> command will push the specified branch of your local repo to the specified remote repo (i.e. on GitHub).

a. Use a command that will push *your feature branch* from your local repo to your *origin* repo on GitHub. Notes: You can use the name of the remote repo, so you do not need to use the full URL. You will also need to paste your PAT as the password when it is requested.

Give a screenshot of your command the output it generates.

b. The output from part a should not contain any error messages. The last few lines should also give you some indication that that your push was successful. Visit your origin repository on GitHub and check that the branch was pushed.

Give a screenshot showing that your branch is now in your origin repository.

**Make Pull Request:**

A *pull request* is the mechanism by which you ask the maintainers of a project 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 must contain your feature branch, to the upstream repository. In the last section, you pushed your feature branch to your origin repository, so you are now ready to make a pull request.

Figure - Making a Pull Request

22. 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.

Nothing is required here. You just need to click the “Compare & Pull Request” button.

b. A gray bar similar to the one shown below should appear at the top of the page.



This bar specifies the repositories and branches that are involved in the pull request. That is, they indicate which branch you are requesting to be pulled from which repository and to which branch in which repository you would like it to be merged. Note that the specific information that you see in your browser will be different than what is shown above.

The questions in this section will help you to better understand what this information means.

i. Copy the URLs from the gray bar into the appropriate rows in the “URL” column of the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | |  |  |
|  |  | **URL** | |  |
|  | Base Repository |  | |  |
|  | Head Repository |  | |  |
|  |  | |  |  |

ii. The terms “*Base Repository*” and “*Head Repository*” in part i are generic labels that GitHub uses for the two repositories that are involved in the pull request. Often, and in our case, these two repos will be your origin repository and the project’s upstream repository. Examine the URL’s from part i and fill in the right hand column below with either *head repository* or *base repository* to indicate which term refers to your origin and which refers to the upstream.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | |  |  |
|  | Your origin repository |  | |  |
|  | The upstream repository |  | |  |
|  |  | |  |  |

iii. The information in the gray bar also indicates the branches involved in the pull request. It uses the generic labels of *base* and *compare* for these branches. Complete the table below by filling in the rows of the “Branch Name” column with the names of the base and compare branches. Then fill in the “Changes” column with “pulled from” or “merged onto” as appropriate.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | |  |  |  |
|  |  | **Branch Name** | | **Changes** |  |
|  | Base branch |  | |  |  |
|  | Compare branch |  | |  |  |
|  |  | |  |  |  |

23. To complete your pull request (PR) you will need to add some additional details. Complete the following steps in your browser to create your PR. You do not need to give individual answers for each step here. The details that you provide will become a part of your pull request and your instructor will see them on GitHub.

a. Give your PR a descriptive title – a similar idea to using meaningful commit messages.

b. Replace the <add description> in the body of the PR with a longer description of your changes.

c. If a PR fixes an issue, which yours does, then you can include a line in the body of the PR that says **exactly** “Fixes #123” **or** **exactly** “Closes #123” **or exactly** “Resolves #123” (where 123 would be replaced by your assigned issue number). When a PR with one of these lines is merged into the project the listed ticket (e.g. 123 in this example) will be closed automatically. Including these lines is helpful to the project maintainers because then they don’t have to remember to go close the associated ticket each time they merge a pull request.

Add a line to the body of the PR that will close the issue that you fixed.

The **#123** part of the line should automatically become a link to the ticket. If it does not, check to be sure you have entered the information **exactly** as shown above.

d. Check that status of your PR and confirm that it can be merged automatically. It should look similar to the following image:



e. Click the green button to create the Pull Request.

Graphical user interface, text, application, chat or text message

Description automatically generated

24. Go to the upstream repository on GitHub and open the “Pull Requests” tab:



Find your Pull Request. Give the number, title and URL for your pull request here.

25. On your pull request page there will be “Files Changed” tab:



Open that tab. At the bottom of that page will be a diff that shows the changes that you have made. This diff shows both the upstream version of the code and your version of the code. It also indicates the changes you have made. The project maintainers will often use this diff to see the details of what you have changed.

Briefly describe how the diff indicates the changes that you made?

**Automated testing of your submitted Pull Request:**

As you have just learned, in GitHub, pull requests are used to propose code changes. Many times, these pull requests can be associated with *Continuous Integration/Continuous Deployment (CI/CD)* tests, which automate the building, testing, and deployment of code. When a contributor submits a pull request, many repositories implement CI/CD systems that can trigger automated tests to ensure that the proposed changes meet certain basic quality standards. The test results are displayed in the pull request, allowing reviewers to assess their impact. If the tests pass, the changes can then be further considered for being merged into the main codebase. Integrating CI/CD tests in pull requests can help to maintain code quality and stability.

In the FarmData2 repository, there have been some automated tests implemented for any submitted pull request that do a few basic checks to ensure that the pull request meets a minimum set or requirements. In the case of FarmData2, each pull request gets checked for the

following:

* The pull request description contains a valid issue link.
* The mentioned issue exists in the list of issues.
* The pull request submitter has been assigned to the referenced issue.

If all of these requirements are met and the tests pass, then this provides the repository maintainer advance information that the pull request meets minimum quality standards. However, if any CI/CD tests do not pass, then it is incumbent upon the contributor to fix any problems that may be preventing the tests from passing. Many times, a maintainer will not even review the pull request until it has passed all initial CI/CD tests.

26. Go to the upstream repository on GitHub and open the “Pull Requests” tab:



27. Find your submitted Pull Request. Does it have a green checkmark or a red x next to it?

Examples:  or 

If your pull request shows a red x, you’ll need to investigate the issue further to determine the cause of the failed test. Remember, the tests check for the criteria listed above in the bulleted list. In FarmData2, a failed test will generate a comment in the pull request attempting to provide additional information as to why the test failed. A failed test comment may look like the following:

A screenshot of a computer

Description automatically generated

Take the steps to fix the issue based on the comment left by the test. Consult with your instructor if necessary.

Finally, it's important to note that not all repositories in GitHub necessarily have CI/CD tests associated with them. The use of CI/CD tests in pull requests depends on the development practices and preferences of the project maintainers. While CI/CD tests provide valuable benefits such as automated testing and quality assurance, their implementation is not mandatory for every repository. Some projects may rely on alternative testing approaches or may not have dedicated resources to set up and maintain CI/CD systems. Therefore, the presence or absence of pull request tests can vary across different repositories in GitHub based on the specific requirements and practices of each project.

**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.

28. Complete the table below by filling in the right-hand column with the commands that accomplish the task listed on the left. Use < > to indicate parameters that need to be 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 made to the active branch. |  |  |
|  | Create a new feature branch. |  |  |
|  | Change the active branch |  |  |
|  | Check the current state of your local repo. |  |  |
|  | Display the changes that you made to a file. |  |  |
|  | Stage changed files to be committed to the repo. |  |  |
|  | Another way to stage changed files for a commit. |  |  |
|  | Commit staged files with a message. |  |  |
|  | Push a feature branch to your origin. |  |  |
|  |  |  |  |

**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:_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)>