Prerequisites

- Install an IDE (e.g. VSCode) on your computer if you have not done so already.
- Install you command line tools (e.g. Python, NodeJS, etc) if you have not done so already. (Use commands like "`python --version`" or "`python3 --version`" to determine if you have these tools installed.)
- You will also need a terminal application for issuing command line instructions. Windows cmd and MacOS terminal com pre-installed. You also might like iTerm2 or Powershell. Pick one and make sure it is installed on your computer.

GIT

1. In a command prompt, type "`git --version`" (That's two dashes.) If the git version is displayed, skip step 2!
2. Install git
    - macOS users should have git installed, but if you haven't run git before, you may be prompted to install the Xcode command line tools. You will need these tools installed on your Mac.
    - Windows users have two options to install git:
        - Git for windows includes a bash environment.
        - The official Git release for windows will allow you to use git from command prompt or Powershell.
        - If you have trouble with your local Windows machine, you could optionally try the Anyware VMs.

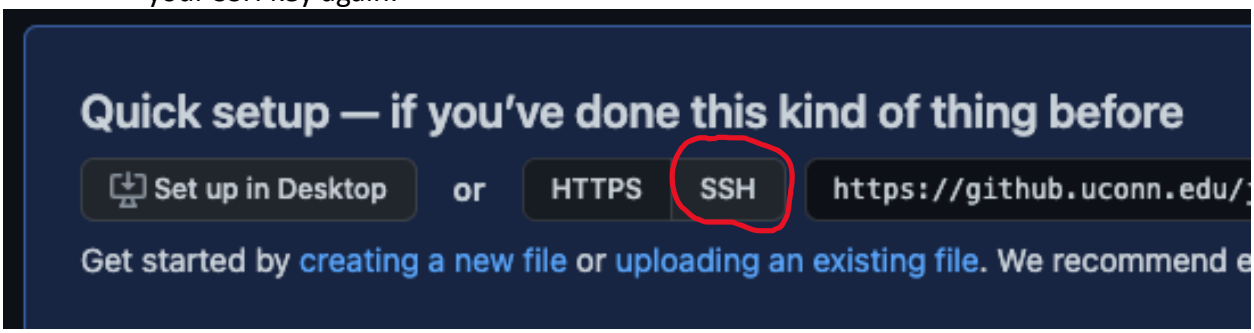
GitHub requires you to authenticate to your repository by using either SSH or HTTPS. The SSH method requires a one-time setup of credentials and is slightly more complicated. The HTTPS method requires you to enter a user/password repeatedly. We recommend the SSH approach.

3. Create a SSH key
    - macOS users follow these instructions to generate a SSH key
    - For windows, there are three options:
        - For Windows 10 or newer, openssh utilities are now included. Please refer to microsoft's Key-based authentication in OpenSSH for Windows page.
        - Openssh utilities are included in Git bash from "Git for windows." From there you can use bash git as described in the github guide linked above.
        - If neither of these are an option (such as on a Anyware VM), then use the https url to connect to GitHub (instead of ssh) and an authentication window will pop up when needed.

4. Add your new SSH key to your github account.

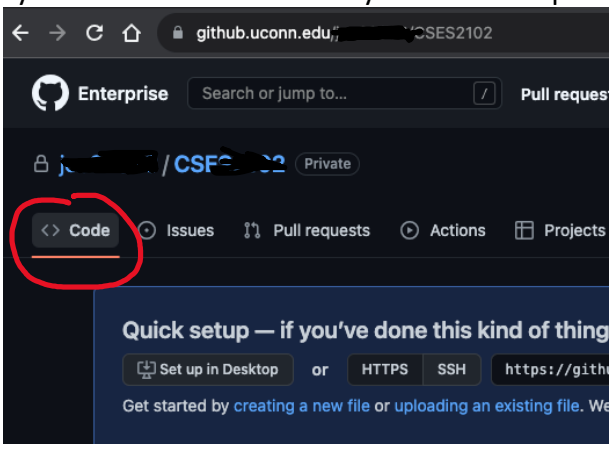
5. Create a new git repo on the github server
    - Navigate to **github.uconn.com**
    - Click the "+" in the top right and select "New Repository"
    - Select the owner to be the name of your course. (e.g. "**CSE4095-Spr24**").

- It is useful to keep all of your work for a given class in a single repo. Specify a name of the repo using a naming convention like**: firstname-lastname-netid**. For instance, if your name is LeBron James, your repo would be named "lebron-james-lej24071". This will help your instructors and TAs to quickly differentiate the work of various students.
- Chose "**Private**". (NOTE: All of your work for your course must be placed in this private repo. The instructors and TAs will have access to view all private repos in the given organization.)
- Keep the rest of the defaults and click the "Create Repository" button.
- Github will display a results page with the commands you need to enter in step 6 below. Notice how the instructions contain a button to select either the SSH or HTTPS approach. You must click the "SSH" button. Keep this page open! Once you close it, you cannot retrieve your SSH key again.

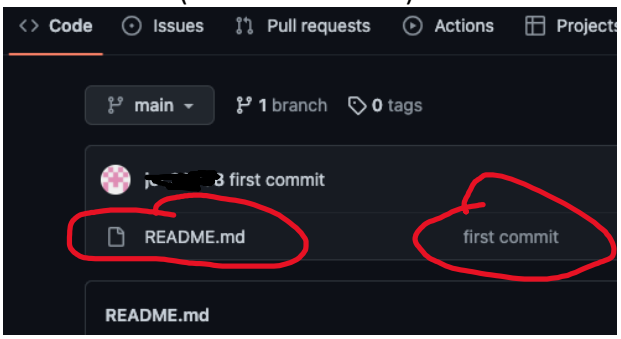


6. Create a local repo on your machine
   - Open a command prompt (e.g. Cmd, Terminal, Powershell, or iTerm2)
   - Important: Create an empty folder on your local computer and navigate into it. For instance:
     ```
     mkdir cse4095
     cd cse4095
     ```
   - Copy/paste the commands from step 5 above. The commands should look something like this:
     ```
     echo "# lebron-james-24071" >> README.md
     git init
     git add README.md
     git commit -m "first commit"
     git branch -M main
     git remote add origin git@github.uconn.edu:CSE4095-Spr24/lebron-james-24071.git
     git push -u origin main
     ```

7. Verify the connection between your local computer and the GitHub server



- Click on the "Code" menu (as shown above) and you should see the list of your files. If you did the previous step, you should see a "README.md" file with a commit comment of "first commit". (As shown below.)



8. Create a new branch called "develop" in your local command-line tool
   - The syntax to create a new local branch that does not exist on the remote server is: `git checkout -b <new branch name>`
   - After you create a new local branch, you must "push" this branch to the remote server before you can push other files and changes. Note, git will give you the exact command if you forget this step:
     `git push --set-upstream origin <branch name>`
   - Create a new folder in your repo for each assignment to keep your work separated. For instance, "hw01". You could do this in your IDE (e.g. VSCode) or via the command line.

9. Create your code.
   - Use these git commands in your development. Commit often. Practice pushing to the remote server.

Git Command	Notes
git diff <filename>	Shows the differences between your local file and the copy of that same file in the remote repository. Very useful for remembering what you changed to make sure you don't have extra code, and for creating a meaningful message in the commit step below.
git status	Displays the modifation status of each local file and the current branch.
git add <filename>	Stages a file for the next commit. Use a * for the <filename> to stage all changed files. Hint: use "git status" to know which files you've changed. Then do a "git add" for each changed file. Note: you can list multiple files in a single git add command. E.g. "git add file1.py file2.py file3.py".
git commit -m "Your message here"	Creates a snapshot of all staged files. If you omit the "-m" switch, you will be placed into the vi editor where you can enter a commit message. Make sure you add meaningful message to every commit. Your instructors and TAs will be looking for these!
git push	Sends the snapshot to the remote server.

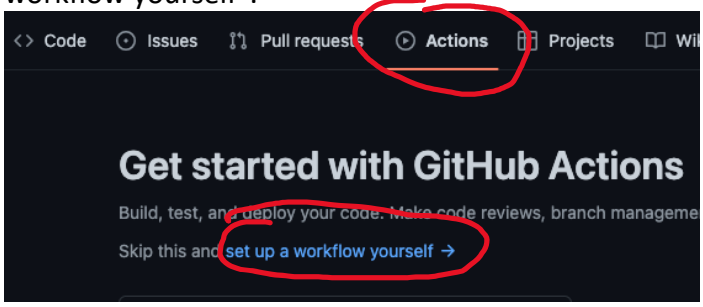
git push --set-upstream origin <branch name>	Initial push command when creating a new local branch
git pull	Pulls down any changes from the remote server for your current branch. This would be necessary, for instance, if any developer made changes in the repo or if you made changes directly in the remote server via the git browser interface.
git pull <branch name>	Pulls changes from another remote branch and merges them into your current local branch. Be careful trying to do this until you are very comfortable with git commands.
git log	Displays the git history for the current branch.
git ls-remote origin 'pull/*/head'	Displays the list of all pull requests.
git fetch	Downloads commits and updates branch list locally.

10. Don't forget to create unit tests to verify that your application can handle any user input without crashing and will produce the correct output for various edge cases. Place your tests in a separate file from the source code of the application. For instance, if you are creating a Python application, you could use pytest or unittest packages for your test case generation.

GitHub Actions

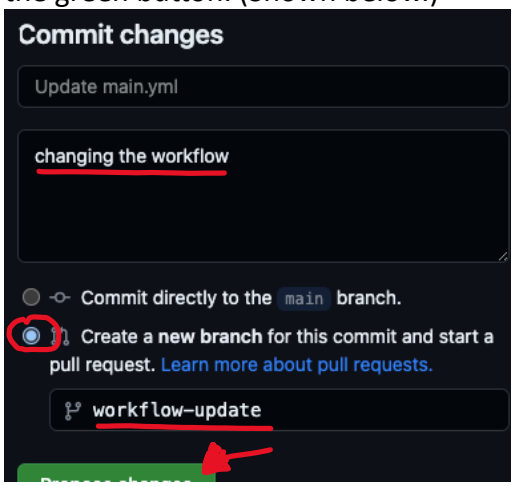
GitHub Actions is tool used for Continuous Integration and Continuous Deployment (CI/CD). Continuous Integration (CI) is about integrating changes from multiple developers and running automated tests to insure the code still works after being merge into the repo. Continuous Deployment (CD) automatically deploys merged code to a server where it can be tested manually or automatically. The CICD automation reduces the changes for human error and creates a predictable result. CICD is very useful for migrating code between successive environments (E.g. Dev, Test, Prod, etc.)

11. Create a CICD Pipeline using Github Actions
    - In the GitHub repo browser interface, click on "Actions" in the menu. Then click on "set up a workflow yourself".

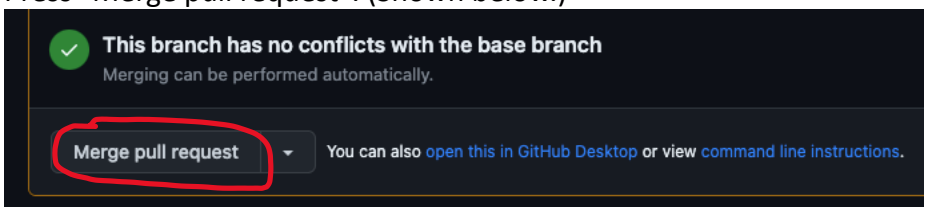


    - You are adding a new file to your repo directly on the remote server (rather than doing it locally and pushing to the remote server). You can find examples of yaml files online or in this repo from Prof. Jonathan Clark. Copy / paste one of the templates you find into your new GitHub Actions workflow. Modify the code to match your repo.

    - Note: when using github.uconn.edu, it is import that the "**runs-on**" value in your yaml code is "**self-hosted**". The code you copy/pasted contains this line already. (Your professor also needs to have the UConn GitHub admins to create Runners in their GitHub Organization.)
- After you complete your yaml file edits, click the green button "Start Commit" in the top right of the page. Click "Commit new file."
- A popup window will appear. Add a description (e.g. "Changing the workflow"), select the "Create a new branch..." option, provide a branch name (e.g. workflow-update), and press the green button. (Shown below.)



- Press "Create pull request"
- Press "Merge pull request". (Shown below.)



- Press "Confirm Merge" to complete the merge. (Afterwards, you may optionally press "Delete Branch" to keep your repo clean.)
- Now go back to your IDE (e.g. VSCode) and make a small change to a file. Use the command-line instructions mentioned at the beginning of this tutorial to Add, Commit, and Push to your Develop branch.
- In your Browser, go to the "Actions" tab and you should see github running your new pipeline.
    - A green checkmark indicates that the workflow passed! A red X means that your workflow failed. (Note: A failure does not necessarily mean the workflow itself is bad. Your workflow might fail because of a syntax error in your python tests. In the world of CI/CD, failed workflows are very common. The errors mean that a developer messed something up and someone needs to figure out what happened in the underlying code.)

12. Update your local repo
    - Whenever you edited a file directly on the remote server (e.g. when you created the action workflow yaml file), your local computer does not know about it yet. If you don't sync your local computer to the sever, you are going to have problems!
    - In your terminal window, run the following commands to pull changes down from the server:

```
git fetch
```

```
git pull
```

13. But what if we don't want the tests to run every time we push to the "develop" branch? Maybe we only want the automation running when we push (or merge) to the "main" branch. To do that, we need to update the top our workflow yaml to say something like this:

```
on:
  push:
    branches:
      - main
```

    - Remember, if you make changes to your yaml file locally, then you need to add/commit/push to the remote server. If you make these changes to the yaml file directly on the remote server, you need to pull those changes down to your local computer.

14. What is a Linter?
    - In the example python GitHub action pipeline, you will see on line 25: " - name: Lint with ruff" and on lines 26-30 the run commands for the Ruff linter. Linters are useful in finding warnings and style mistakes in your code. Those types of issues can lead to hidden bugs. Your code might run okay, but if it is not linted, you might have bugs you cannot see. Like the lint in a clothes dryer, Ruff will remove unneeded substance in your code.
    - If you want to see what happens when you have lint errors, trying adding an unused import (e.g. "import os") to your code and add/commit/push. That is an import statement that is not needed. Let's see what Ruff thinks about it!
    - Your pipeline should now fail! Click into the error and you should see the "Unused import" error found by Ruff. Notice that the pytest command never runs. Our workflow requires Ruff to be error free before pytest runs.

Let's review what you have done! (And learn about a few things.) You created some Python code and a corresponding unit test file. You then created a CI/CD pipeline (aka. workflow) that runs pytest on all Python code in your repo. (Pytest automatically looks for files with "test*" methods and runs them.) Your new pipeline is triggered whenever code is merged into the "main" branch.

Now that you have the basics down, imagine creating lots of code with lots of tests. In a group project context, developers on your team would create their own branches and commit/push code as often as they wish. BUT, when those branches get merged into the "main" branch, your pipeline will run! In this way, you are continuously testing the integration of code that is merged into the main branch. That is an example of "Continuous Integration."

You can find lots of examples on the internet of yaml configurations to do common CI/CD tasks. You can also select a common "template" when creating a workflow, rather than creating the yaml code from scratch.

"Continuous Deployment" means that our pipeline sends our code to the server where it will run. That is very convenient and saves us having to manually copy files to a server. If you are using the AWS Learner Lab in a cloud development course, be sure to use the AWS Learner Lab yaml examples provided.