CI/CD with GitHub Actions

Revision 1.5 – 11/14/24

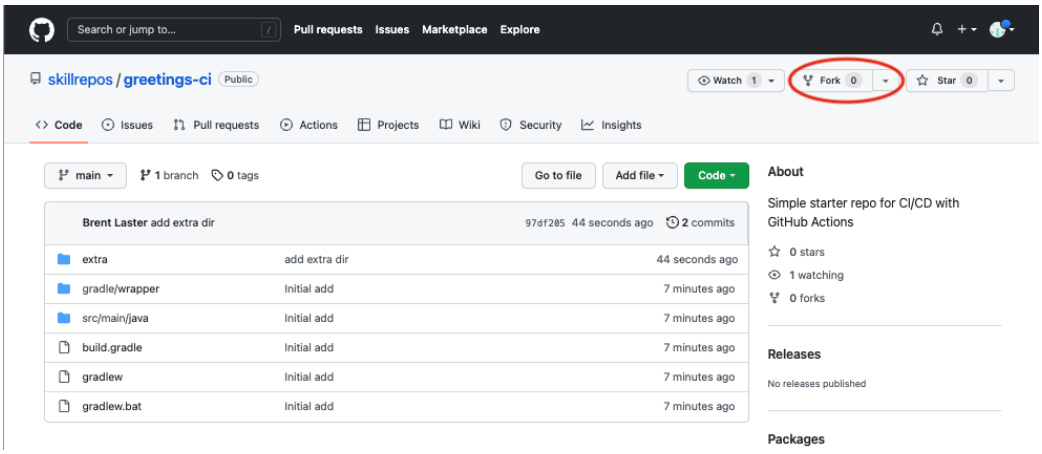
Tech Skills Transformations LLC / Brent Laster

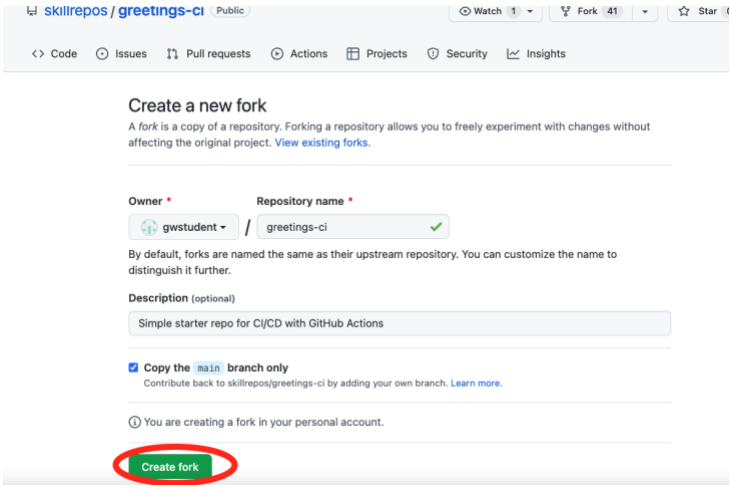
Lab 1 – Creating a simple example

Purpose: In this lab, we'll get a quick start learning about CI with GitHub Actions by creating a simple project that uses them. We'll also see what a first run of a workflow with actions looks like.

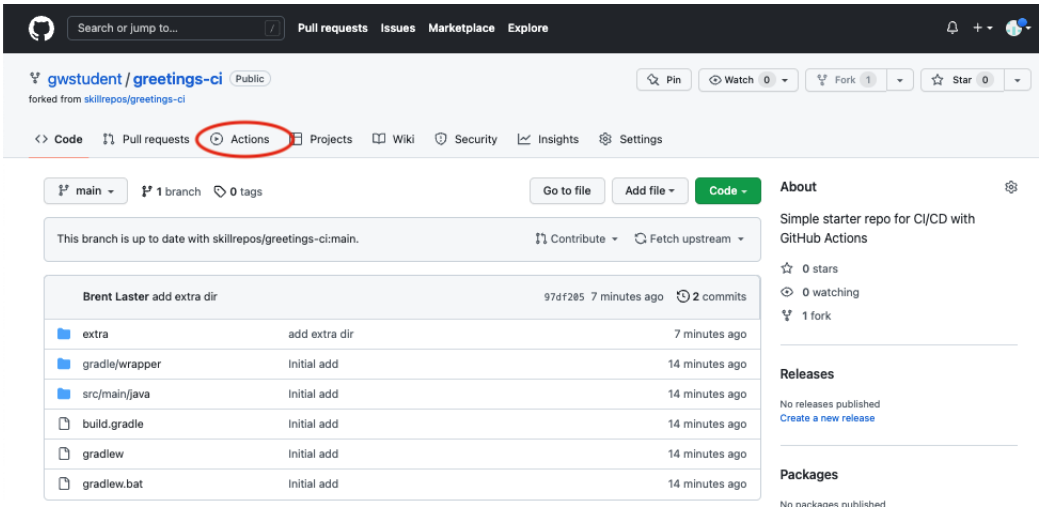
1. Log in to GitHub with your primary GitHub account.

2. Go to https://github.com/skillrepos/greetings-ci and fork that project into your own GitHub space. After this, you'll be on the project in your user space.

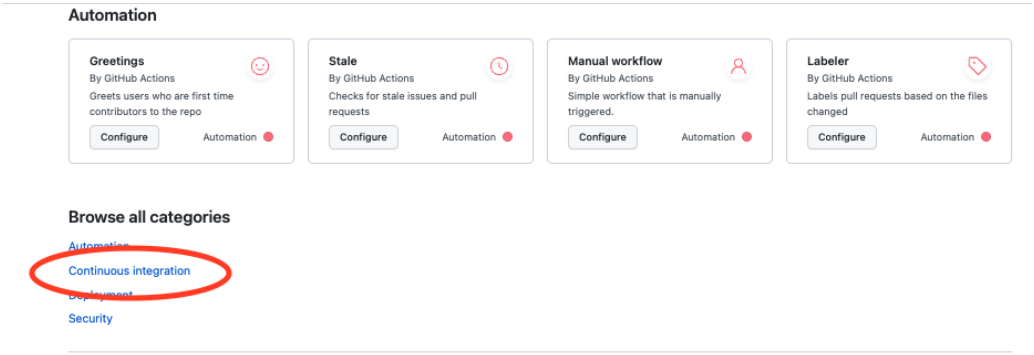




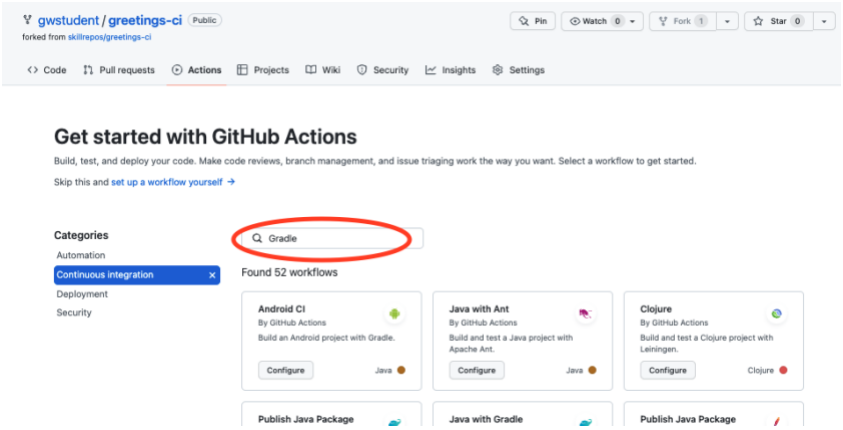
3. We have a simple java source file named *echoMsg.java* in the subdirectory *src/main/java*, a Gradle build file in the root directory named *build.gradle,* and some other supporting files. We could clone this repository and build it manually via running Gradle locally. But let's set this to build with an automatic CI process specified via a text file. Click on the *Actions* button in the top menu under the repository name.



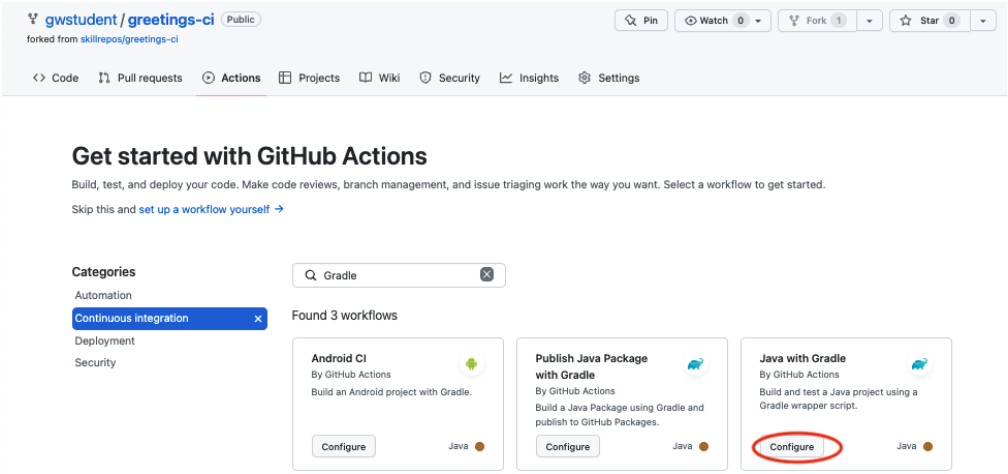
4. This will bring up a page with categories of starter actions that GitHub thinks might work based on the contents of the repository. We'll select a specific CI one. Scroll down to near the bottom of the page under "Browse all categories" and select "Continuous integration".



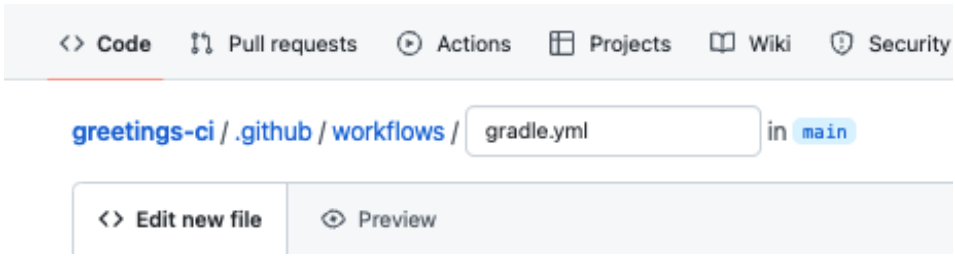
5. In the CI category page, let's search for one that will work with Gradle. Type "Gradle" in the search box and press Enter.



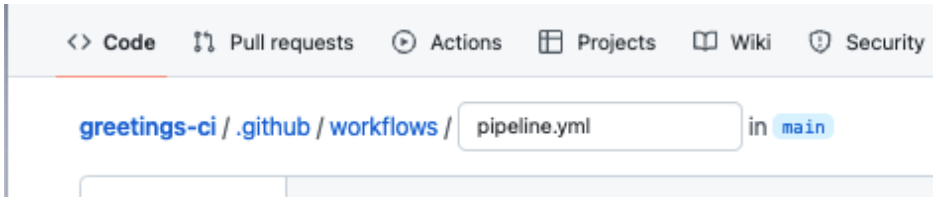
6. From the results, select the "Java with Gradle" one and click the "Configure" button to open a predefined workflow for this.



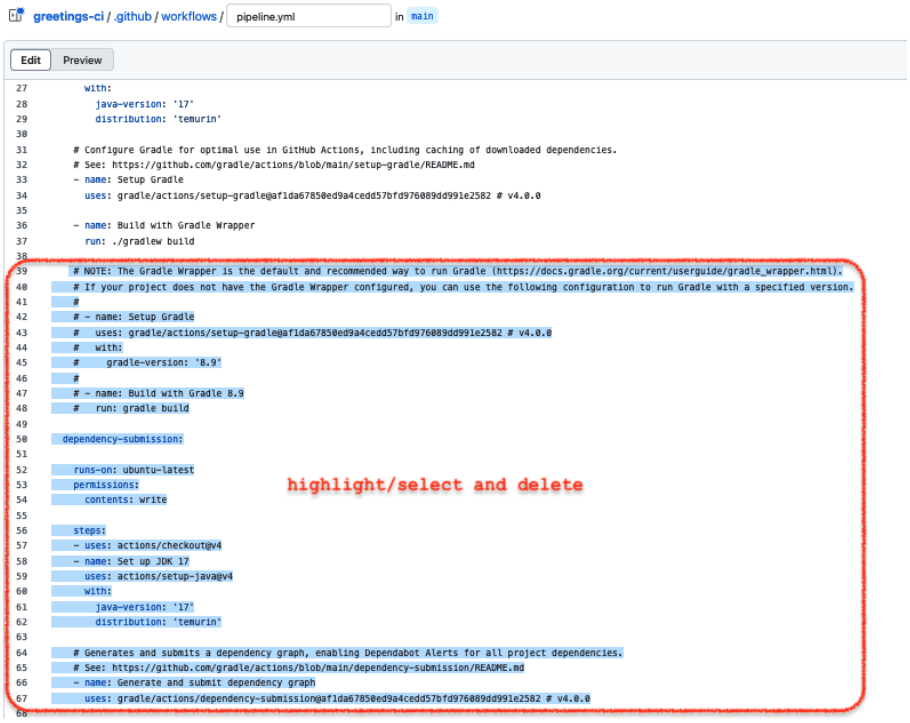
7. This will bring up a page with a starter workflow for CI that we can edit as needed. There are two edits we want to make here. The first is to change the name. In the top section where the path is, notice that there is a text entry box around "gradle.yml". This is the current name of the workflow. Click in that box and edit the name to be "pipeline.yml". (You can just backspace over or delete the name and type the new name.)



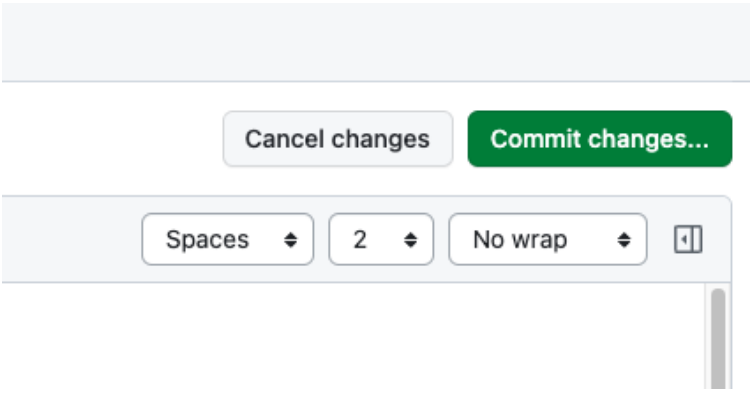
TO

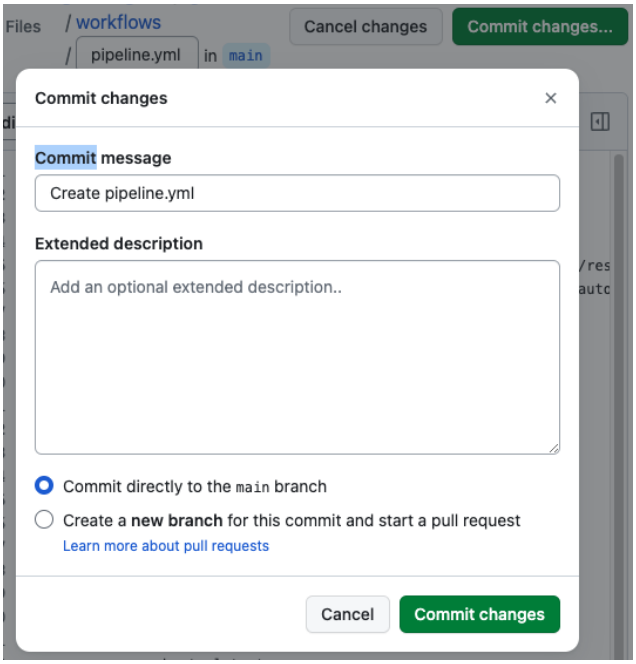


8. The second edit is to remove the second job in this workflow since it currently has issues. To do this we will just highlight/select the code from line 39 on and hit delete. (If you have trouble just selecting that code, try starting at the bottom and selecting/highlighting from the bottom up.) The code to be deleted is highlighted in the next screenshot.

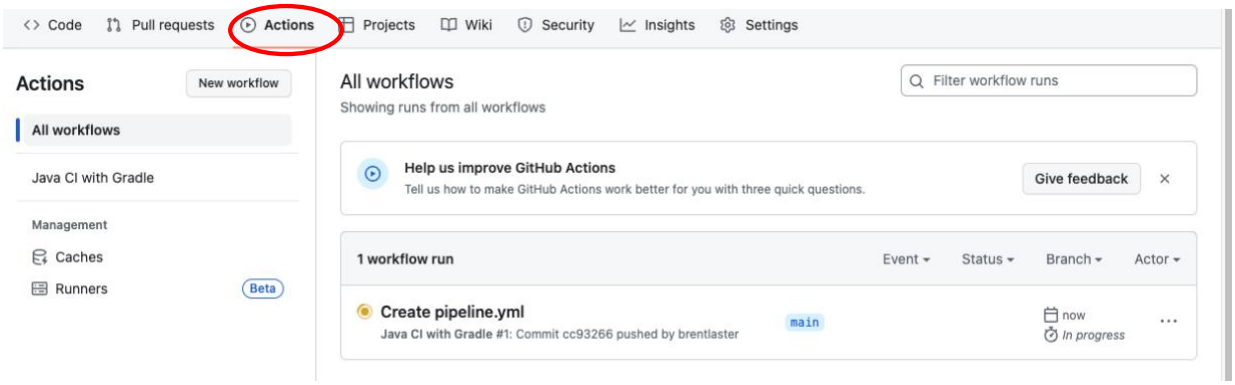


9. Now, we can go ahead and commit the new workflow via the “Commit changes…” button in the upper right. In the dialog that comes up, you can enter an optional comment if you want. Leave the “Commit directly…” selection checked and then click on the “Commit changes” button.

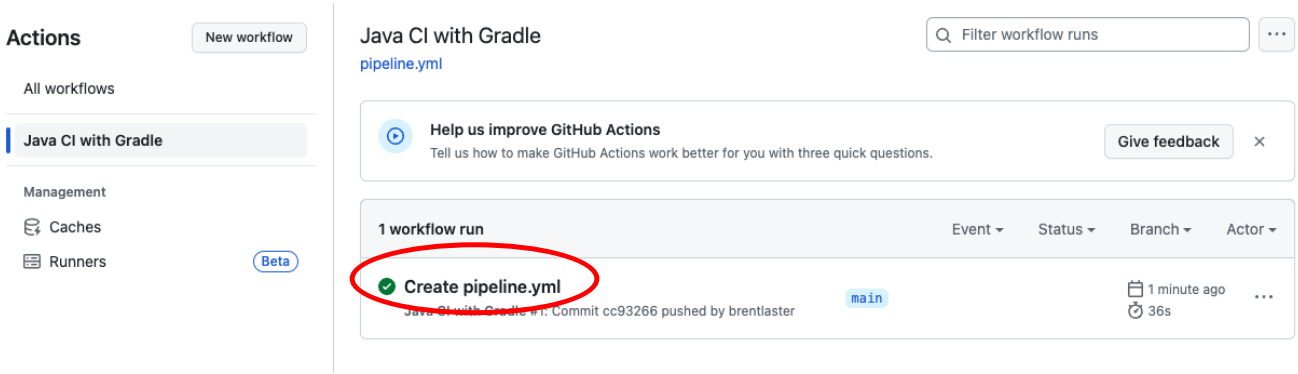




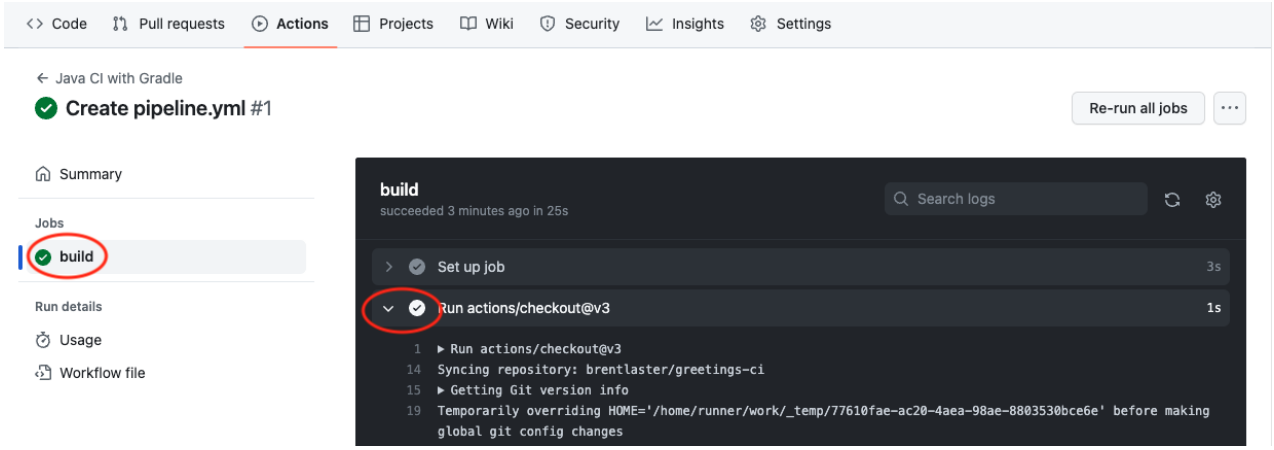
10. Since we've committed a new file and this workflow is now in place, the "on: push:" event is triggered and the CI automation kicks in. Click on the Actions menu again to see the automated processing happening.



11. After a few moments, the workflow should succeed. (You may need to refresh your browser.) After it is done, you can click on the commit message for the run to get to the details for that particular run.



12. From here, you can click on the build job in the graph or the "build" item in the list of jobs to get more details on what occurred on the runner system. You can expand any of the steps in the list to see more details.

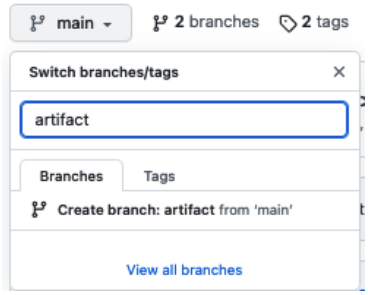


END OF LAB

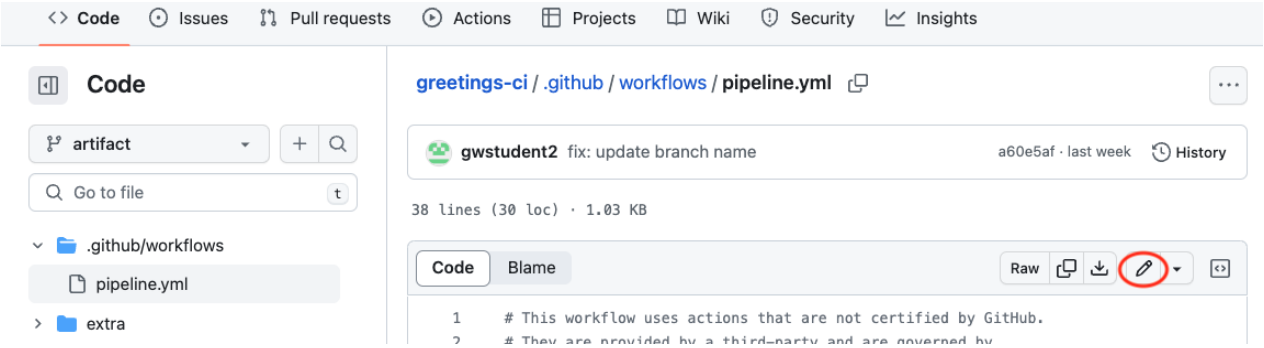
Lab 2 – Managing Artifacts

Purpose: In this lab, we'll look at how to do simple artifact management – an important part of Continuous Delivery.

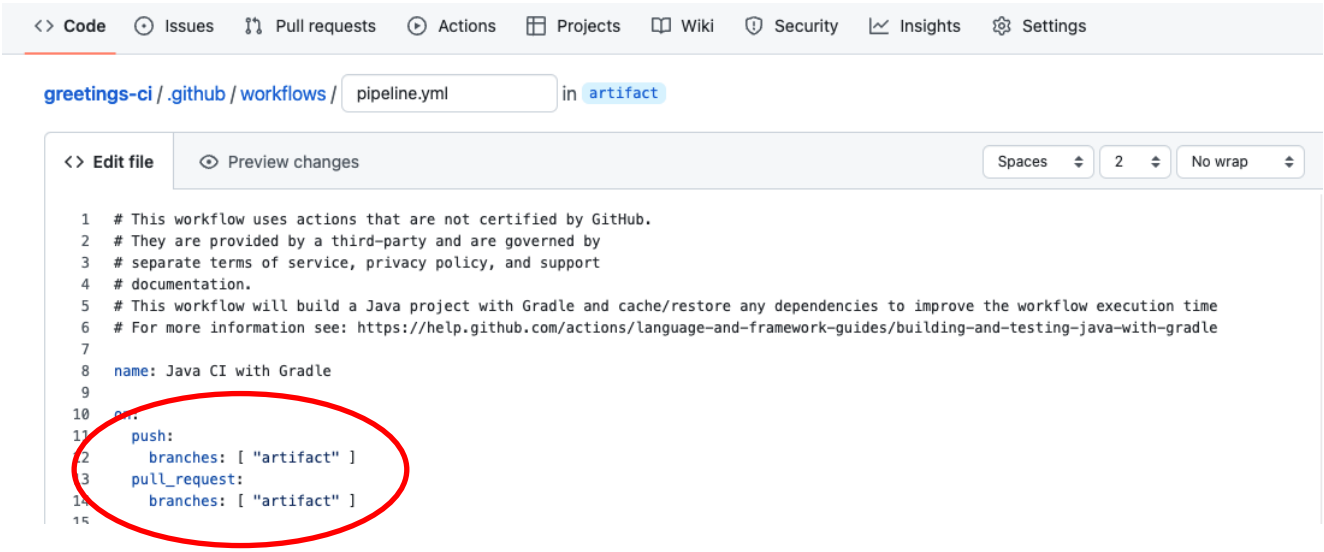
1. As a best practice for building out the pipeline as a larger project, let's create a separate branch to work in for managing the versioning and storage of the artifact. We'll call it "artifact". In the "Code" tab, click on the branch dropdown that says "main". Then in the text area that says "Find or create a branch…", enter the text "artifact". Then click on the **"Create branch: artifact from 'main'"** link.



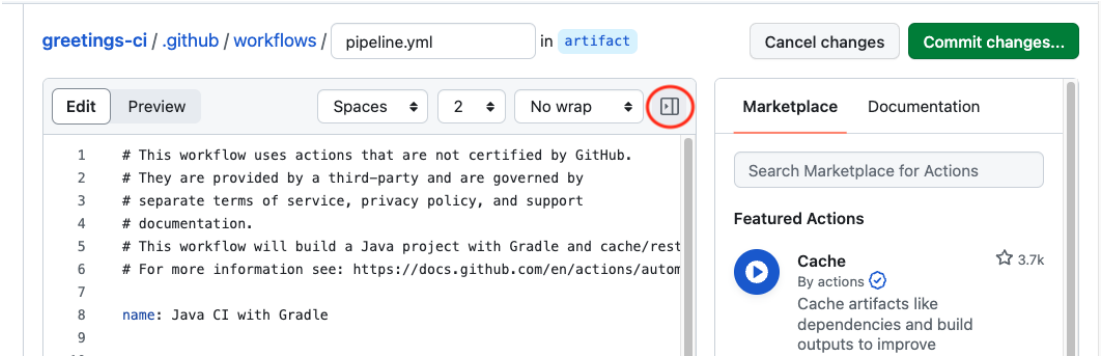
2. Now you should be on the *artifact* branch. We're going to first add the code to persist the artifact that we built in our build step. We want to persist this for use with other jobs in our pipeline such as ones that might test it. Open the .github/workflows/pipeline.yaml file (click on the name) and edit it by clicking on the pencil icon.



3. Change the references in the “on:” clause to be just the “artifact” branch so we don't trigger action runs on the other branches while we are working on this one. Make sure you are on the *artifact* branch before you proceed.



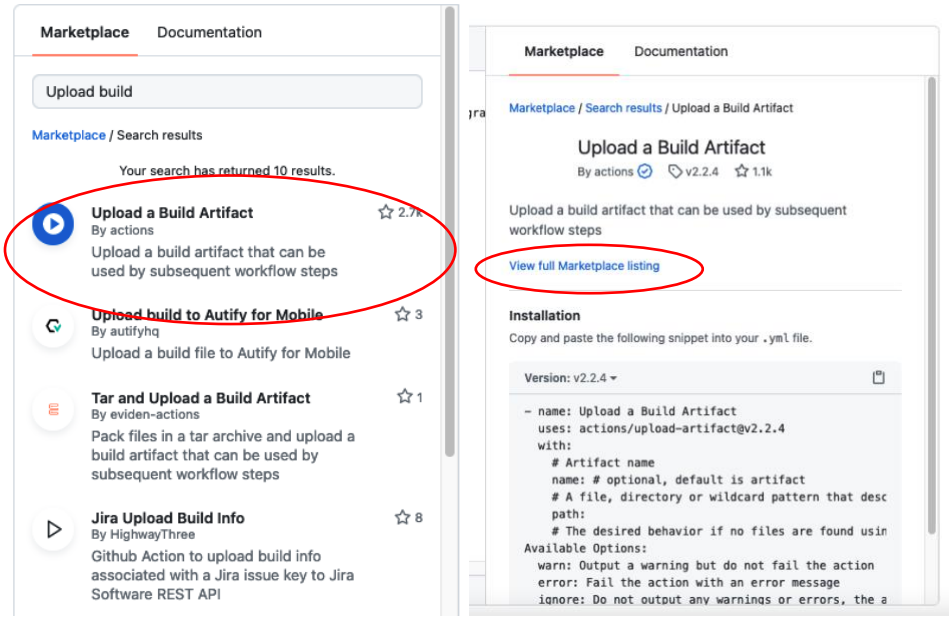
4. Click on the square to the right to show the pane to search for actions (if search pane isn’t shown).



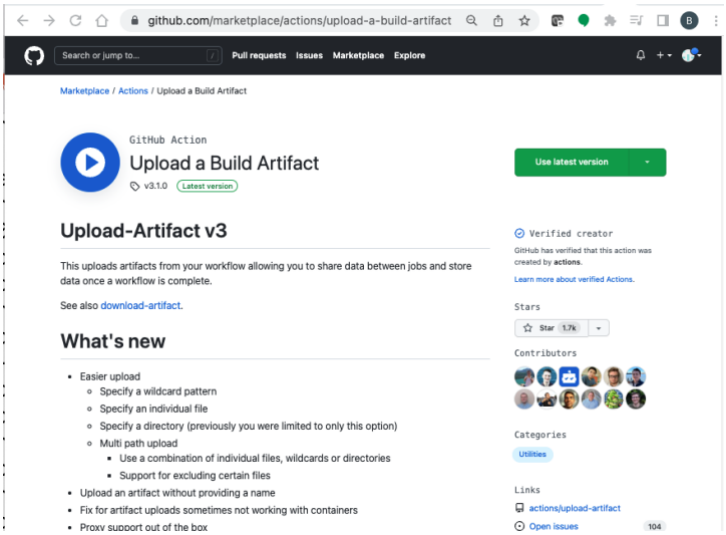
5. To the right, you should see a pane with references to GitHub actions. We're going to add a job to our workflow to upload an artifact. Let's find actions related to uploading.

   In the "Search Marketplace for Actions" box on the upper right, enter "Upload build" and see what's returned.

   Find the one that is named "Upload a Build Artifact By actions" and click on it. Take a look at the page that comes up from that. Let's look at the full listing on the Actions Marketplace. Click on the "View full Marketplace listing".



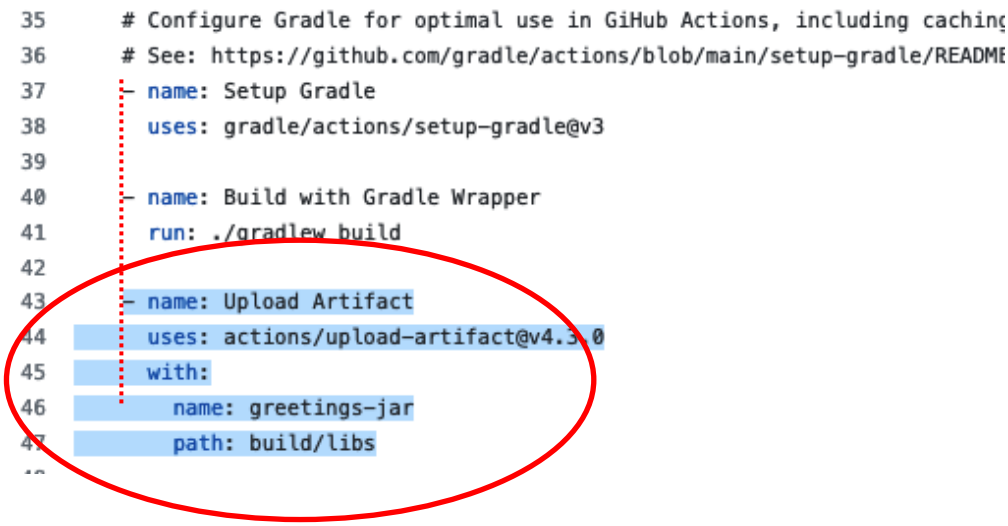
6. This should open the full GitHub Actions Marketplace listing for this action. Notice the URL at the top - https://github.com/marketplace/actions/upload-a-build-artifact.



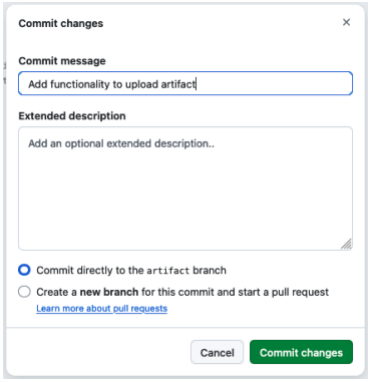
7. Switch back to the browser tab where you are editing the workflow for greetings-actions. Update the build job to include a new step to use the "upload-artifact" action to upload the jar the build job creates. To do this, add the following lines inline with the build job steps. **Pay attention to the indenting.** See the screenshot (lines 40-44) for how this should look afterwards. (Your line numbers may be different.)

   The code to add is immediately below. You can copy and paste but may need to adjust the indenting. Notice this should go after the *Build with Gradle Wrapper* step.

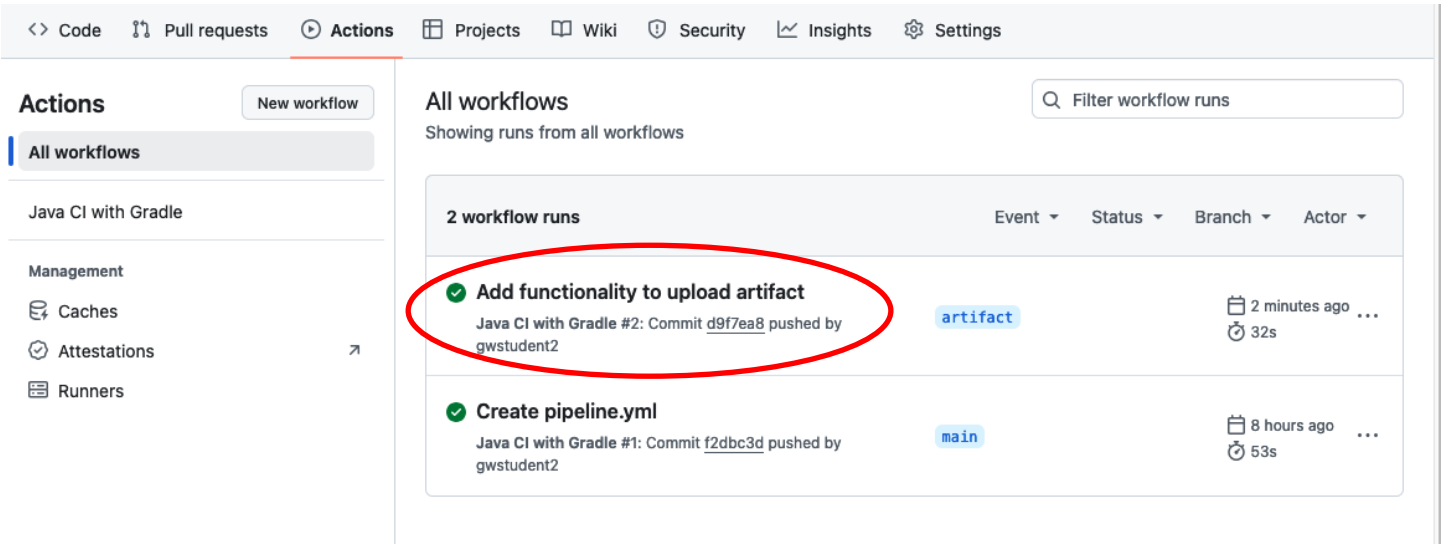
```
- name: Upload Artifact
  uses: actions/upload-artifact@v4.3.0
  with:
    name: greetings-jar
    path: build/libs
```

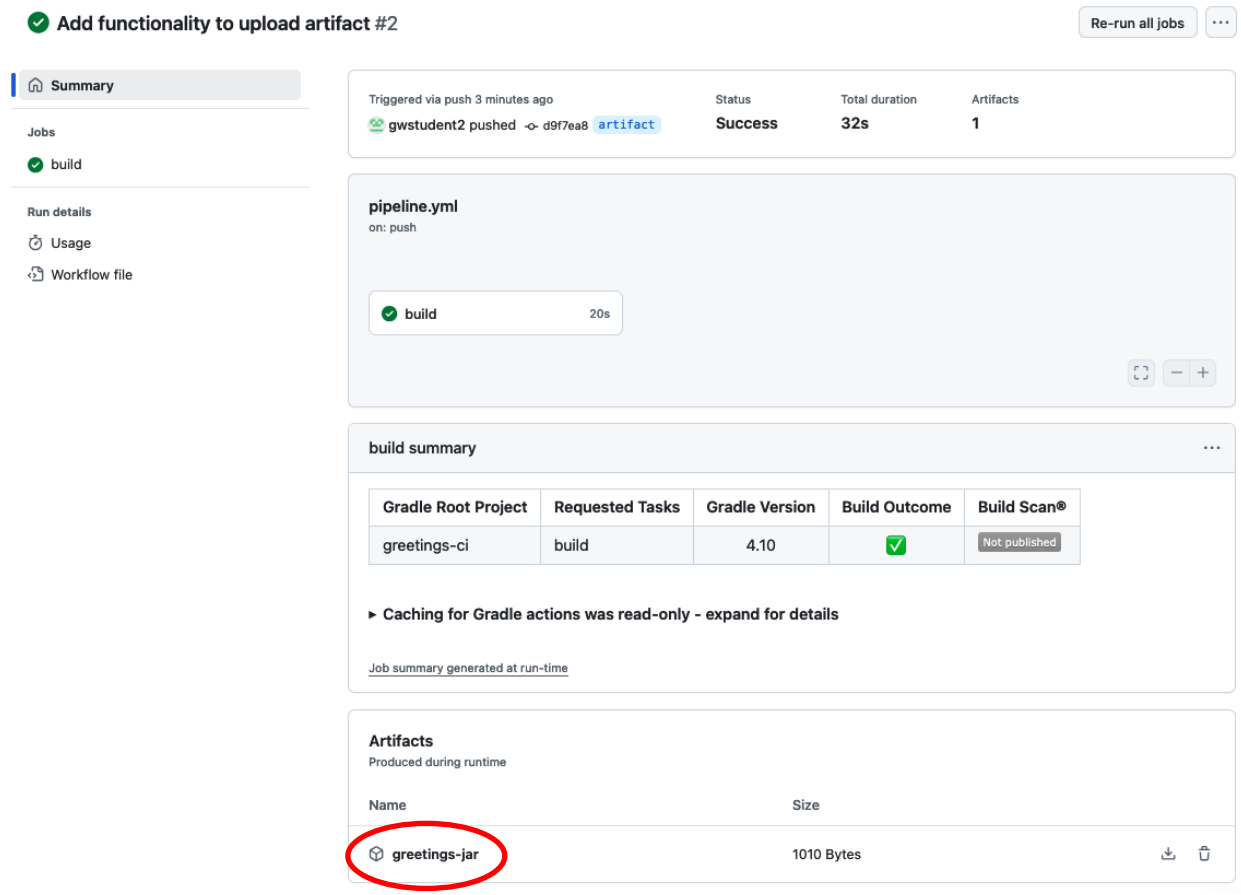


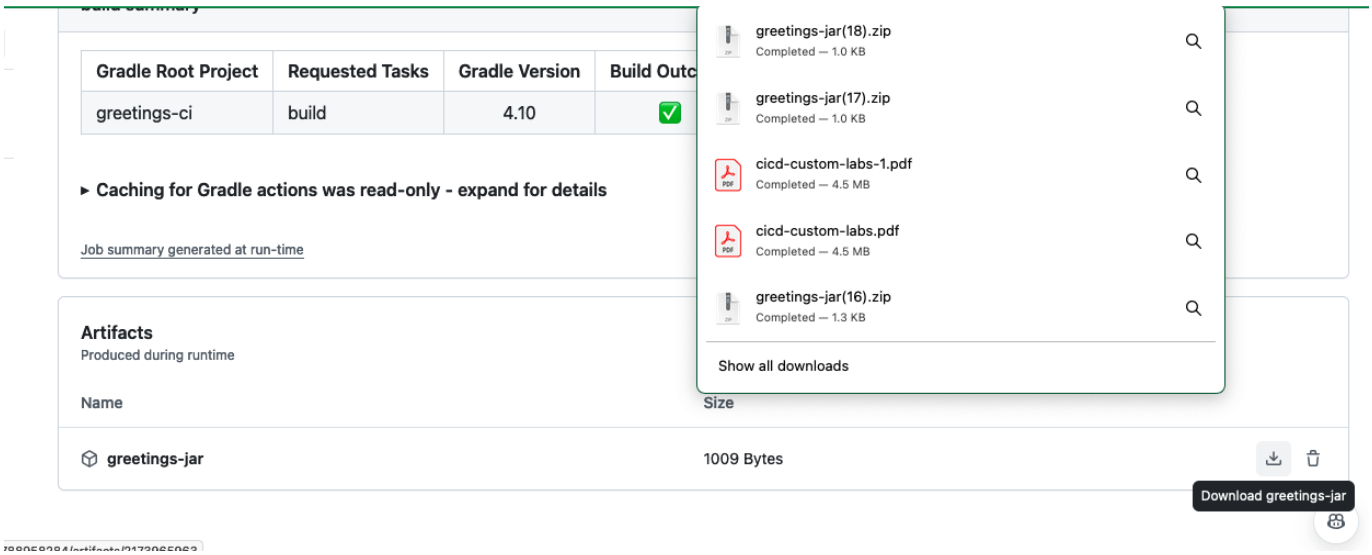
8. Click on the green "Commit changes…" button in the upper right. In the dialog that comes up, add a commit message like "Add functionality to upload artifact", then click the green "Commit changes" button to make the commit.



9. Switch to the "Actions" tab in your repository to see the workflow run. After a few moments, you should see that the run was successful. Click on the title of that run "Add functionality to upload artifact". On the next screen, in addition to the graph, there will be a new section called "Artifacts" near the bottom of the page You can download the artifact from there. Click on the name of the artifact to try this.







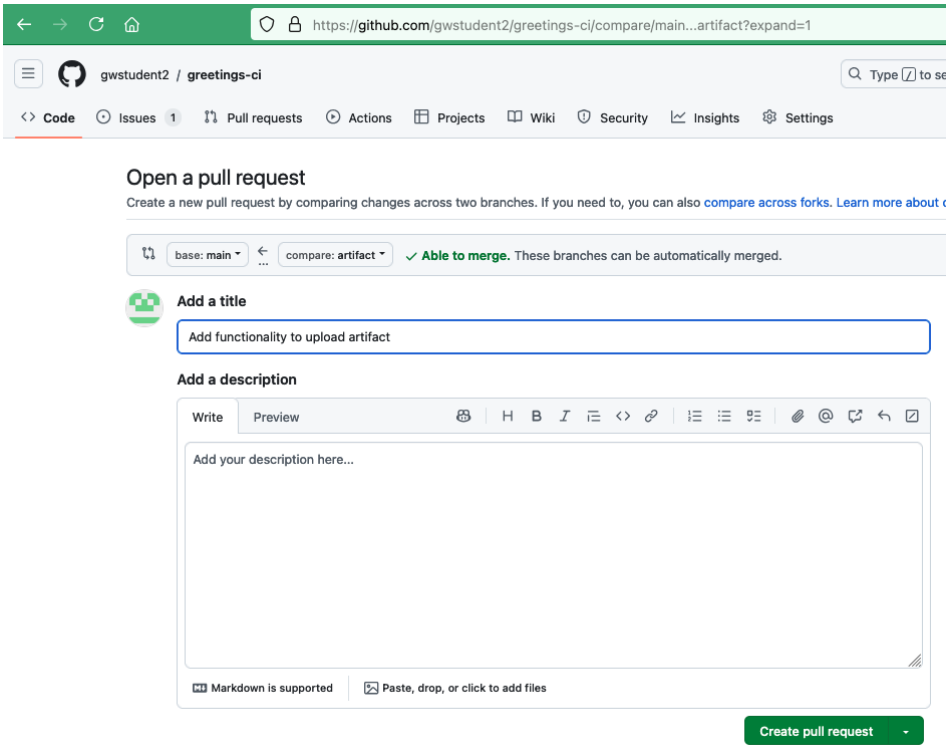
END OF LAB

Lab 3 – Merging changes with a pull request

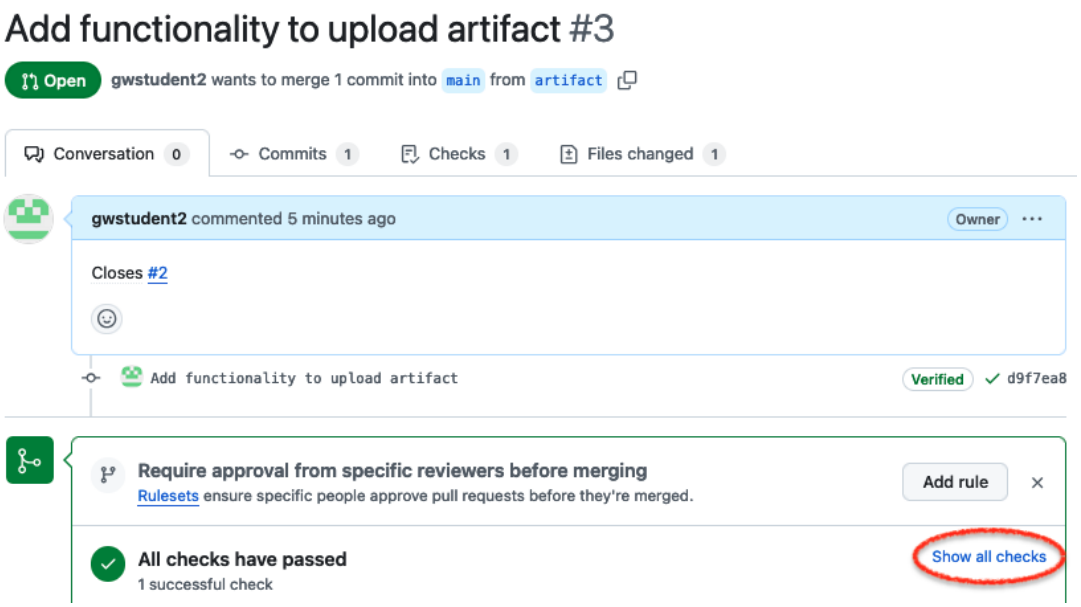
Purpose: In this lab, we'll see how to merge changes with a GitHub pull request

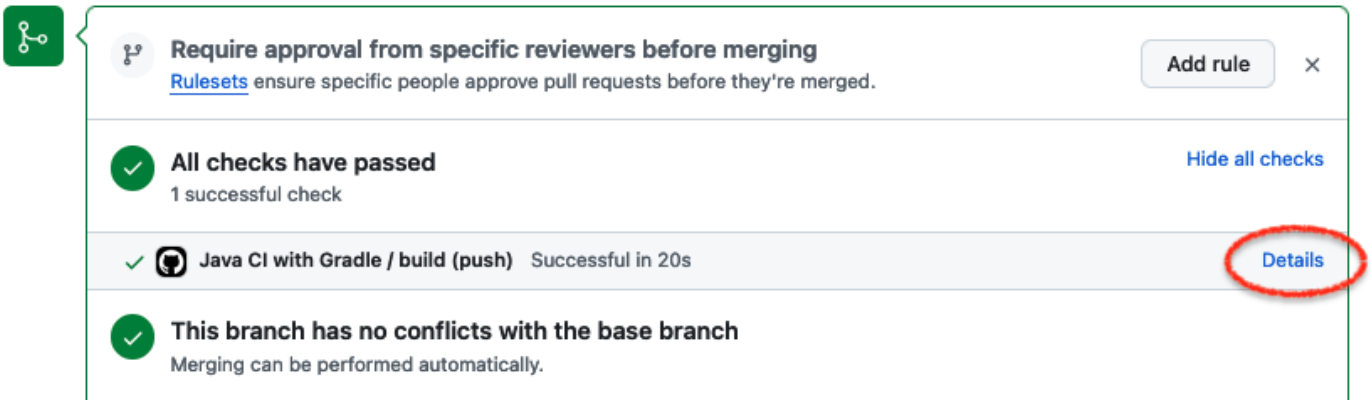
1. Now, let's open up a new pull request for the actual merging. To keep things simple, and avoid targeting the wrong repository, copy the link below, insert your GitHub userid in place of *<github-userid>* and then paste and go to the link. You can scroll to the bottom to see the actual file changes.

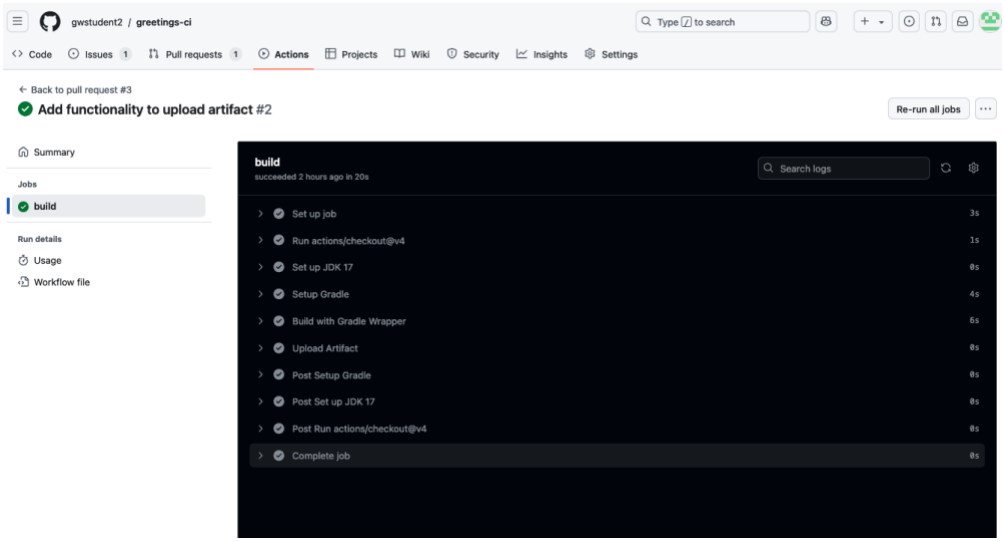
https://github.com/<github-userid>/greetings-ci/compare/main...artifact?expand=1



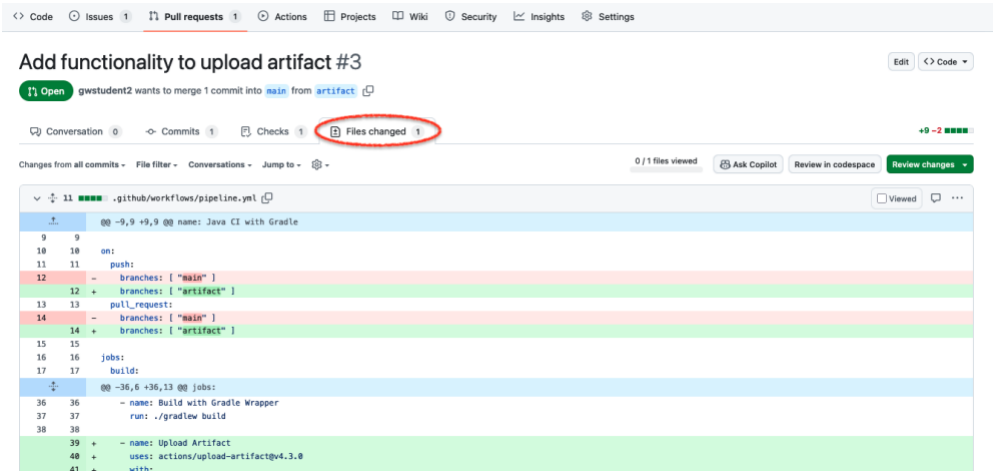
2. When done, click on the *Create pull request* button to actually create the pull request. On the next page, you'll see the pull request with all of the tabs for it. The main one is the *Conversation* tab. Notice the *All checks have passed* section in the middle. This where our GitHub Actions workflow ran because of the *on: pull_request* trigger. Click on the *Show all checks* link to see the workflow run. Then click on the *Details* link to go to the run of the workflow. You can expand the steps of the workflow and look around if you want.







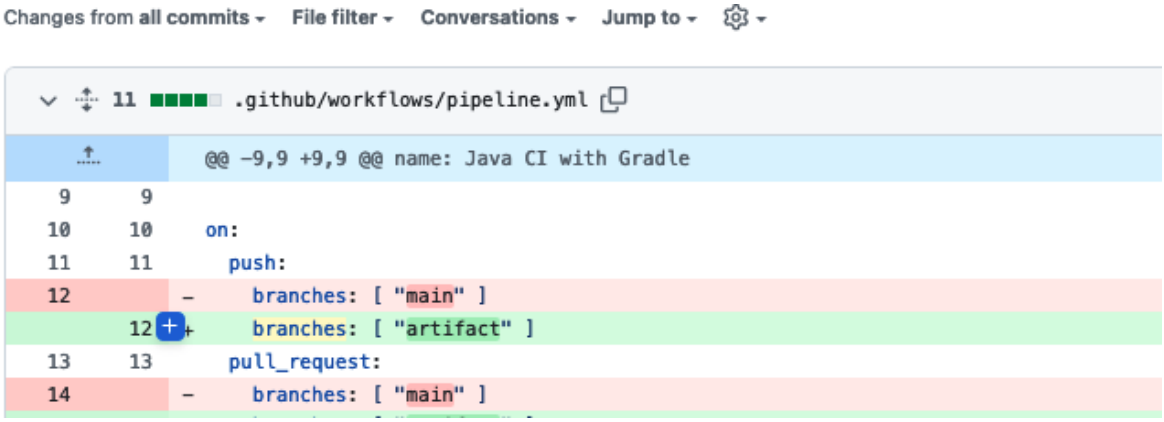
3. When done, go back to the pull request. (You can just click the "back arrow" in your browser.) Now, let's review the changes (as another user would if we added them as a reviewer). Click on the *Files changed* tab at the top to get to a color-code view of the changes.

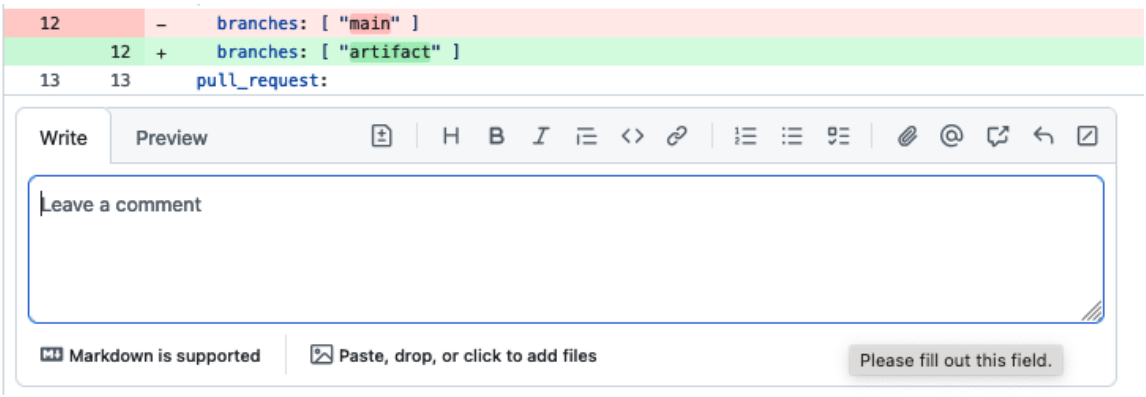


4. You may have noticed that there is a problem. The branches value in main would be overwritten with the value of "artifact" that we had in that branch.

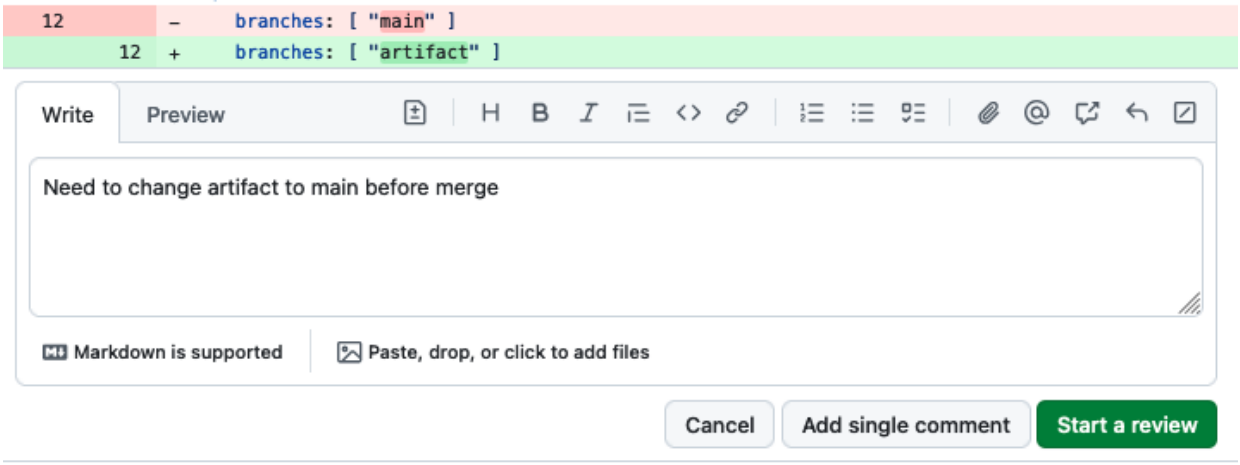
   Let's make some review comments on this. Hover over one of the lines in green that says *branches: ["artifact"].*

   You should see a + sign popup. Click on that and you'll get a pop-up to add a comment.

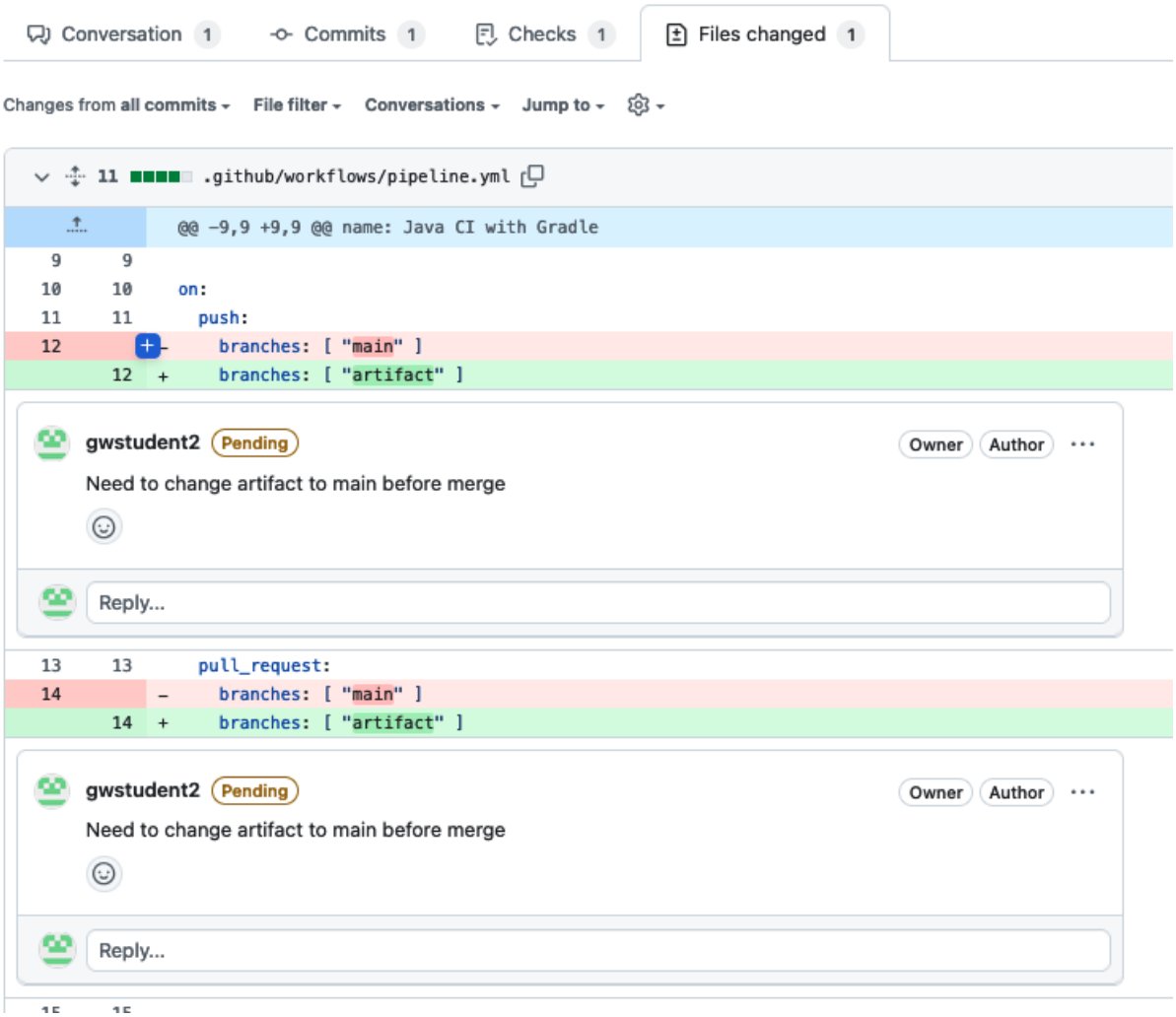




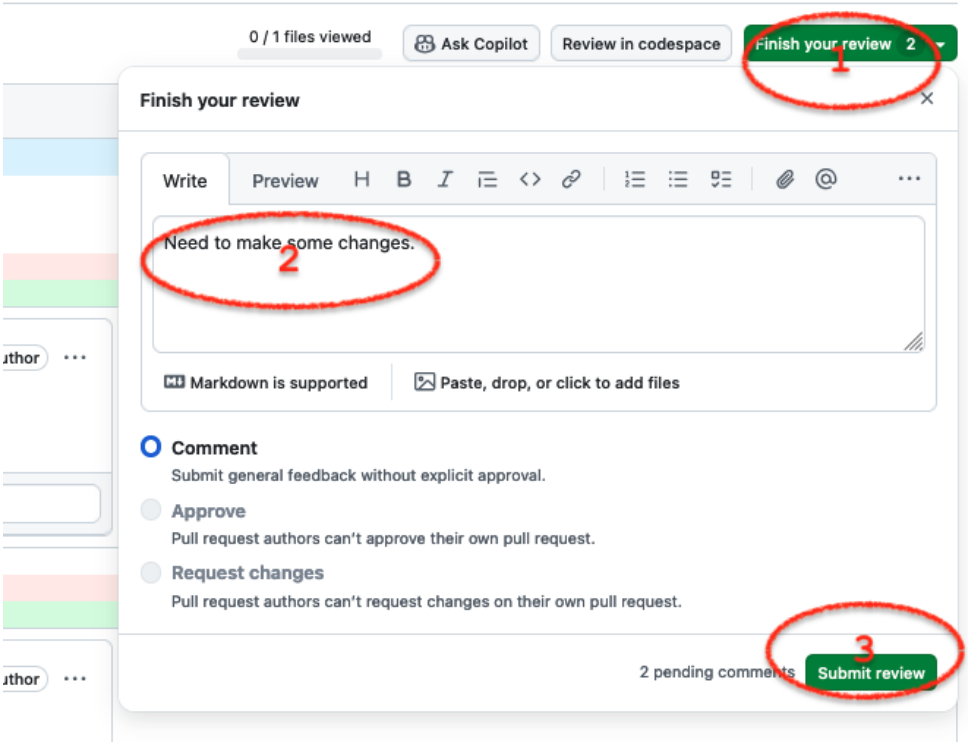
5. In the comment field, enter text like "*Need to change artifact to main before merge*". Then click on the *Start a review* button.



6. This will have started a larger review with your comment pending. Repeat the samesteps above for the other line below that one with *branches: ["artifact"] except click on the "Add review comment"* when done.



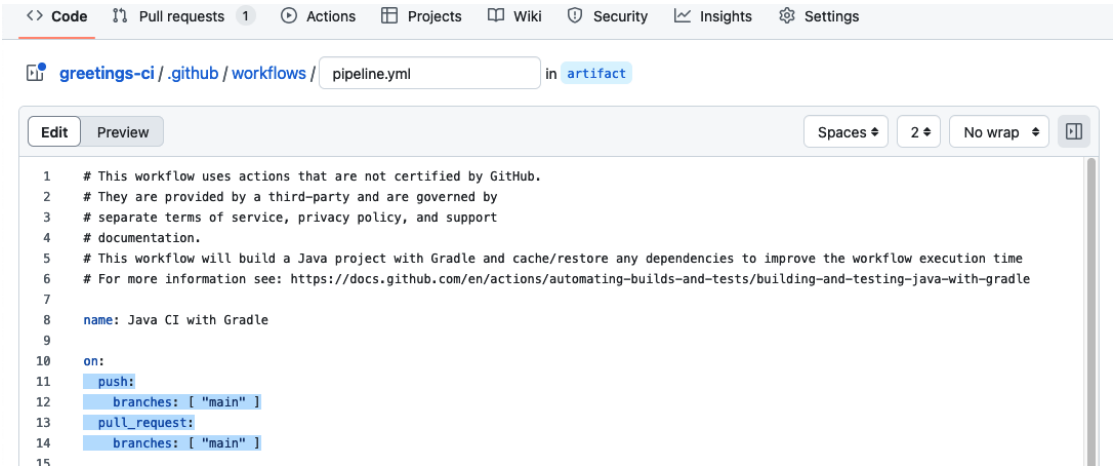
7. Now that we've completed the individual comments, we can finish the review. Click on the *Finish your review (2)* button in the top right. In the main comment section, you can just add some text like "Need to make some changes". As the author of the pull request, your only option will be to leave an overall comment (vs *Approve* or *Request changes*). When ready, click on the green *Submit review* button.



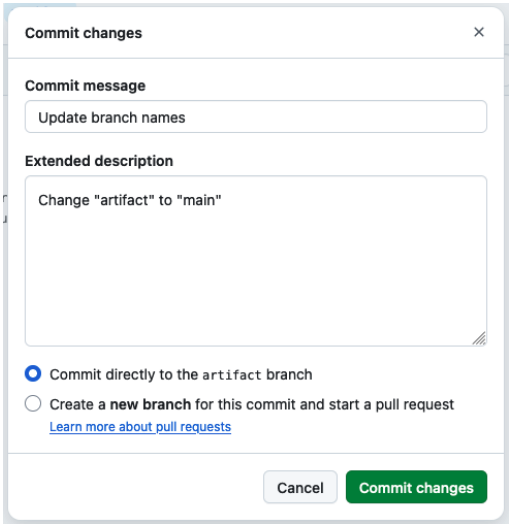
8. After this, you'll see an indicator that your review was submitted successfully. Since we've realized we need to make some changes, let's edit the file in the browser. To do this, let's edit the *pipeline.yml* file in the *artifact* branch. To get there, you can just use the quick link below and substitute your GitHub userid for *<github-userid>*. You can do this in a different browser tab if you want.

https://github.com/<github-userid>/greetings-ci/edit/artifact/.github/workflows/pipeline.yml

9. In the edit screen, change *["artifact"]* to *["main"]* in both lines.



10.. Click on the *Commit changes...* button in the upper right. In the dialog that comes up, you can change the commit message and add an extended description if you want. Leave the option set to "*Commit directly to the artifact branch*" and click "*Commit changes*" when ready.



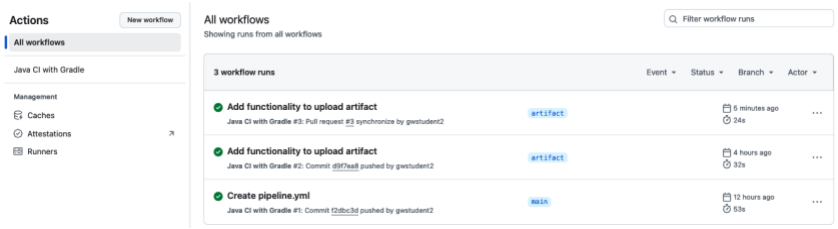
11.. Now if you go back to the pull request, you can look at the *Commits* tab and see both of our commits as well as look at the *Files changed* tab and see that the only difference now is the addition of the step as intended.

Direct links are below (assuming you replace *<github-userid>* with your actual GitHub userid. (Also replace the "1" with a different number if your pull request has a different number.)

https://github.com/<github-userid>/greetings-ci/pull/1/commits

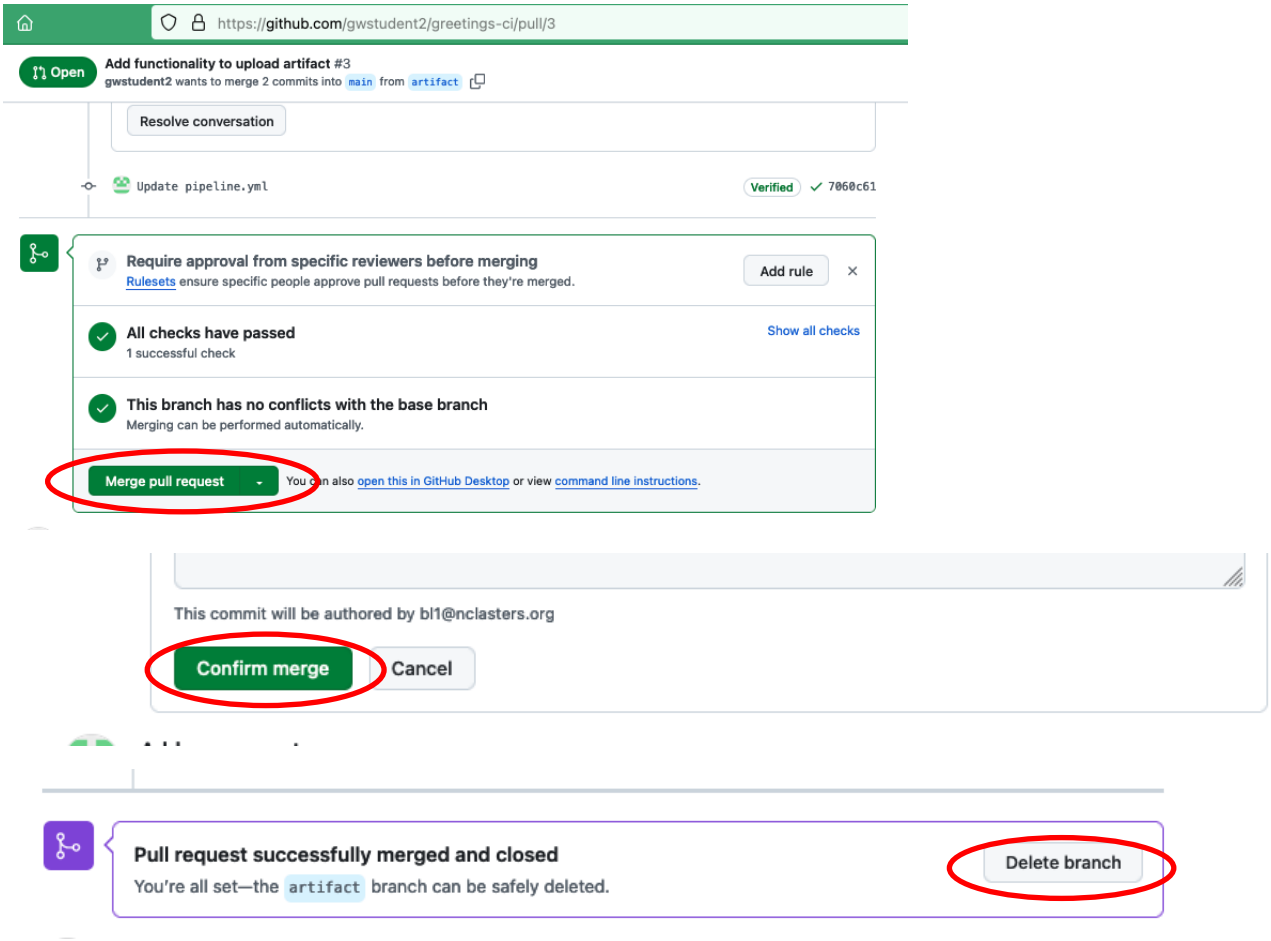
https://github.com/<github-userid>/greetings-ci/pull/1/files

(Another workflow run will have been kicked off, but this should have also succeeded.)

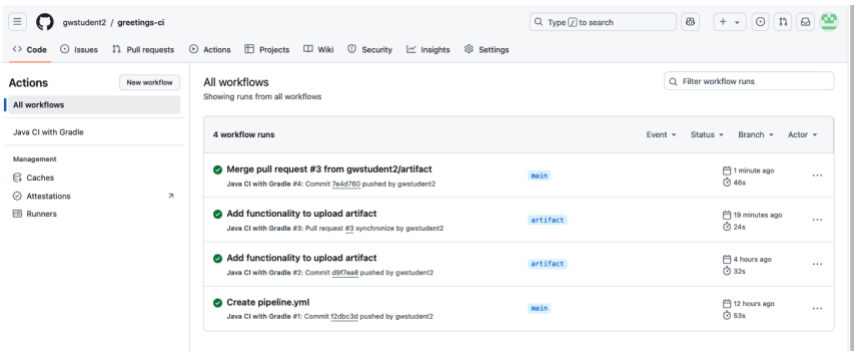


12. Now, you can go to the main page of the pull request (via the *Pull requests* tab at the top or link below) and then click on the *Merge pull request* button. Then click on the *Confirm merge* button. After this, you can click the Delete branch button if you want.

https://github.com/<github-userid>/greetings-ci/pull/1



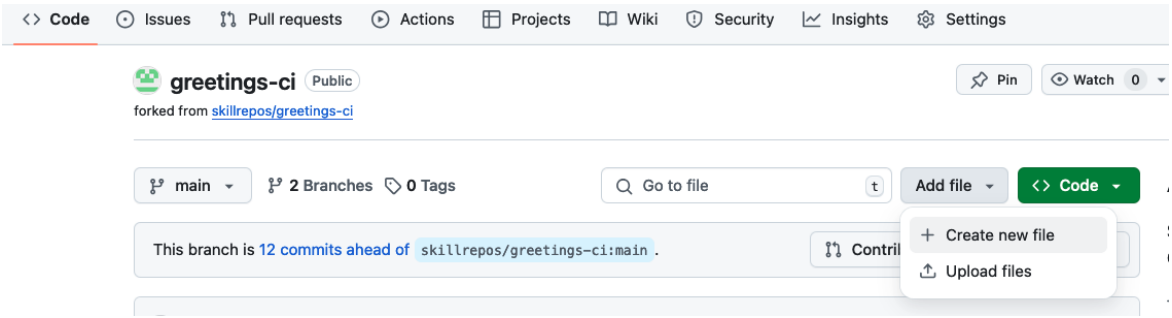
(After this, if you got to the *Actions* menu, you can see a final run of the workflow that was initiated when the merge was done.)



Lab 4 – Adding in a test case

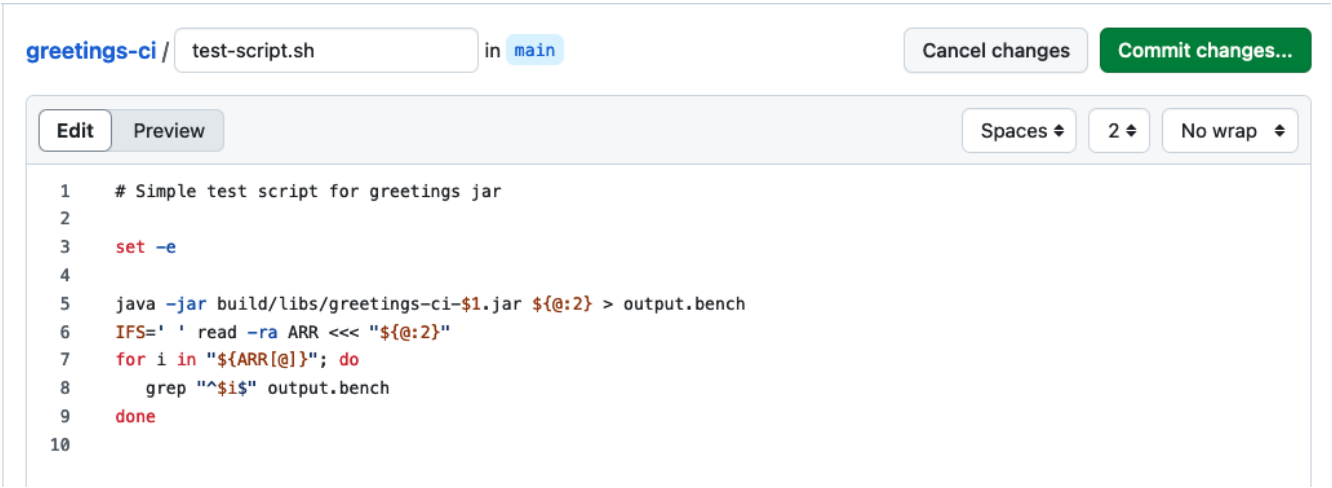
Purpose: In this lab, we'll add a simple test case to download the artifact and verify it

1. Now, let's create a new script to test our code. To create a new file via the browser, go back to the "Code" tab at the top**. To save time, we'll just do this on the *main* branch.** Click on the *Add file* button next to the green *Code* button. From the list that pops up, select *Create new file*.

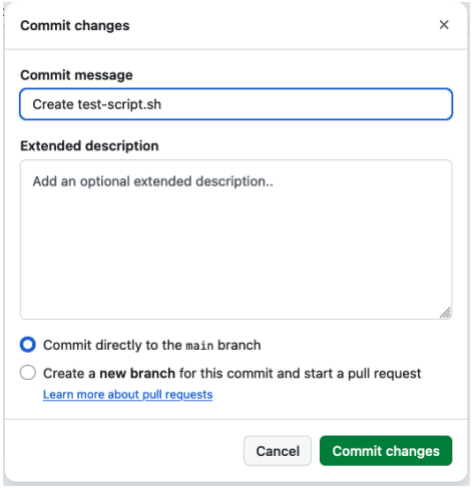


2. In the new editor that pops up, you'll be at the location to type in a name. You can name this "test-script.sh". Then copy and paste the following code into the new file. (A screenshot is shown after the code so you can see how things line up.) (This code is also available at https://gist.github.com/brentlaster/f1c922ff4266882f0e5f2982da053cde)

```
# Simple test script for greetings jar

set -e

java -jar build/libs/greetings-ci-$1.jar ${@:2} > output.bench
IFS=' ' read -ra ARR <<< "${@:2}"
for i in "${ARR[@]}"; do
   grep "^$i$" output.bench
done
```



3. This script takes the version of the jar to run as its first parameter and the remaining values passed in as the rest of the parameters. Then it simply cycles through all but the first parameter checking to see if they print out on a line by themselves. Go ahead and click on the *Commit changes...* button to commit this file into the repository on the *test* branch. Just leave the dialog settings as-is or add a description if you want.



4. Let's add code to version our artifact. For simplicity, we'll just use the current run number as the "patch" component of the version number. And we'll just do a simple move (rename) of the file with the version number to get the artifact "versioned".

   As you've done before, edit the *pipeline.yml* file (select it and click on the *pencil* icon).

   Add the code below in the *build* job between the steps for "*Build with Gradle Wrapper*" and "*Upload Artifact*". The code to add is below and the screenshot shows where to put it. (Make sure to align it with the other steps in the *build* job.)

```
- name:  Version Artifact
  run: >
    mv build/libs/greetings-ci.jar
    build/libs/greetings-ci-0.0.${{ github.run_number }}.jar
```

```
31        # Configure Gradle for optimal use in GitHub Actions, including caching of downloaded dependencies.
32        # See: https://github.com/gradle/actions/blob/main/setup-gradle/README.md
33        - name: Setup Gradle
34          uses: gradle/actions/setup-gradle@af1da67850ed9a4cedd57bfd976089dd991e2582 # v4.0.0
35
36        - name: Build with Gradle Wrapper
37          run: ./gradlew build
38
39        - name:  Version Artifact
40          run: >
41            mv build/libs/greetings-ci.jar
42            build/libs/greetings-ci-0.0.${{ github.run_number }}.jar
43
44        - name: Upload Artifact
45          uses: actions/upload-artifact@v4.3.0
46          with:
```

5. Since each job executes on a separate runner system, we need to make sure our new test script is available on the runner that will be executing the tests. For simplicity, we can just add it to the list of items that are included in the uploading of artifacts. Modify the **path** section of the "Upload Artifact" step in the "build" job to look like below.

```
- name: Upload Artifact
    uses: actions/upload-artifact@v4.3.0
    with:
      name: greetings-jar
      path: |
        build/libs
        test-script.sh
```

The screenshot below shows where to make this change.

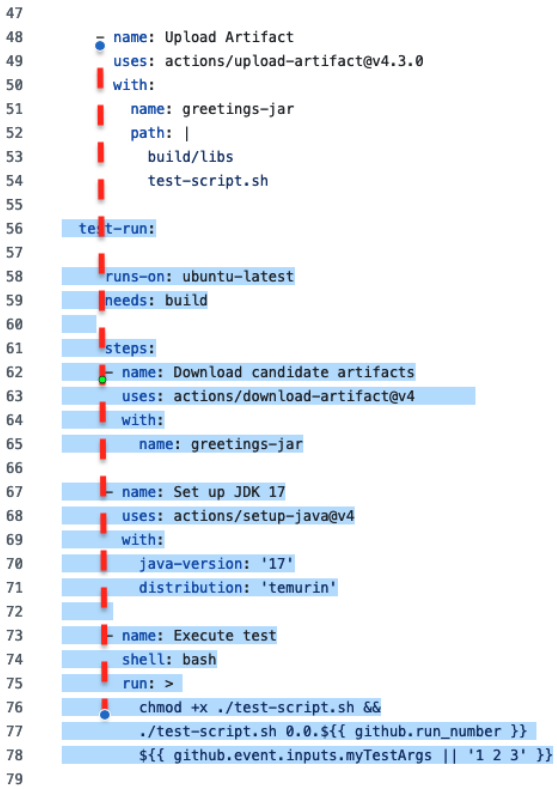
6. Now, we'll add the job definition for a job called "test-run" that runs on ubuntu-latest. What this code does is wait for the build job to complete (the *needs: build* part), then run two steps. The first step downloads the artifacts we uploaded before to have them there for the testing script. And the second step runs the separate testing script against the downloaded artifacts, making it executable first.

   Since we want to test what we built, it will need to wait for the build job to be completed. That's what the "*needs: build*" part does in the code below.

   The screenshot shows where it should go. Pay attention to indentation - *test-run:* should line up with *build:* . (If you see a wavy red line under part of the code, that probably means the indenting is not right.). Also, we will explain the *github.events.input.myTestArgs* in the next section.

```
  test-run:

    runs-on: ubuntu-latest
    needs: build
```

```
 steps:
 - name: Download candidate artifacts
   uses: actions/download-artifact@v4
   with:
     name: greetings-jar

- name: Set up JDK 17
  uses: actions/setup-java@v4
  with:
    java-version: '17'
    distribution: 'temurin'

 - name: Execute test
   shell: bash
   run: >
     chmod +x ./test-script.sh &&
     ./test-script.sh 0.0.${{ github.run_number }}
     ${{ github.events.input.myTestArgs || '1 2 3' }}
```



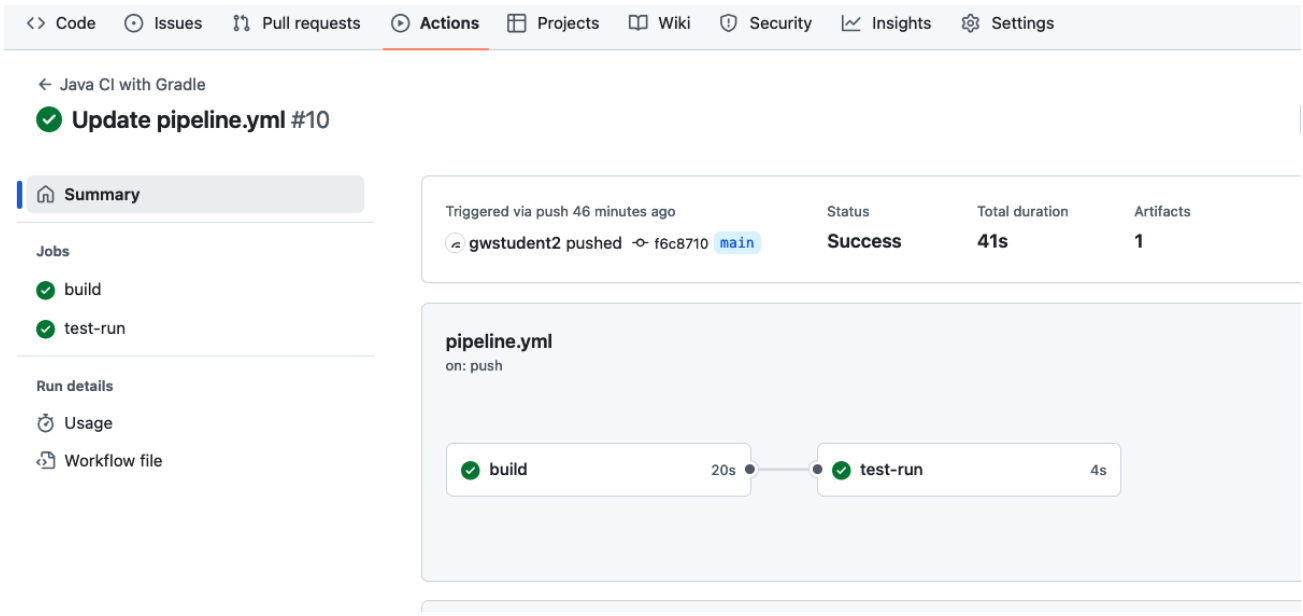
```
47
48        - name: Upload Artifact
49          uses: actions/upload-artifact@v4.3.0
50          with:
51            name: greetings-jar
52            path: |
53              build/libs
54              test-script.sh
55
56      test-run:
57
58        runs-on: ubuntu-latest
59        needs: build
60
61        steps:
62        - name: Download candidate artifacts
63          uses: actions/download-artifact@v4
64          with:
65            name: greetings-jar
66
67        - name: Set up JDK 17
68          uses: actions/setup-java@v4
69          with:
70            java-version: '17'
71            distribution: 'temurin'
72
73        - name: Execute test
74          shell: bash
75          run: >
76            chmod +x ./test-script.sh &&
77            ./test-script.sh 0.0.${{ github.run_number }}
78            ${{ github.event.inputs.myTestArgs || '1 2 3' }}
79
```

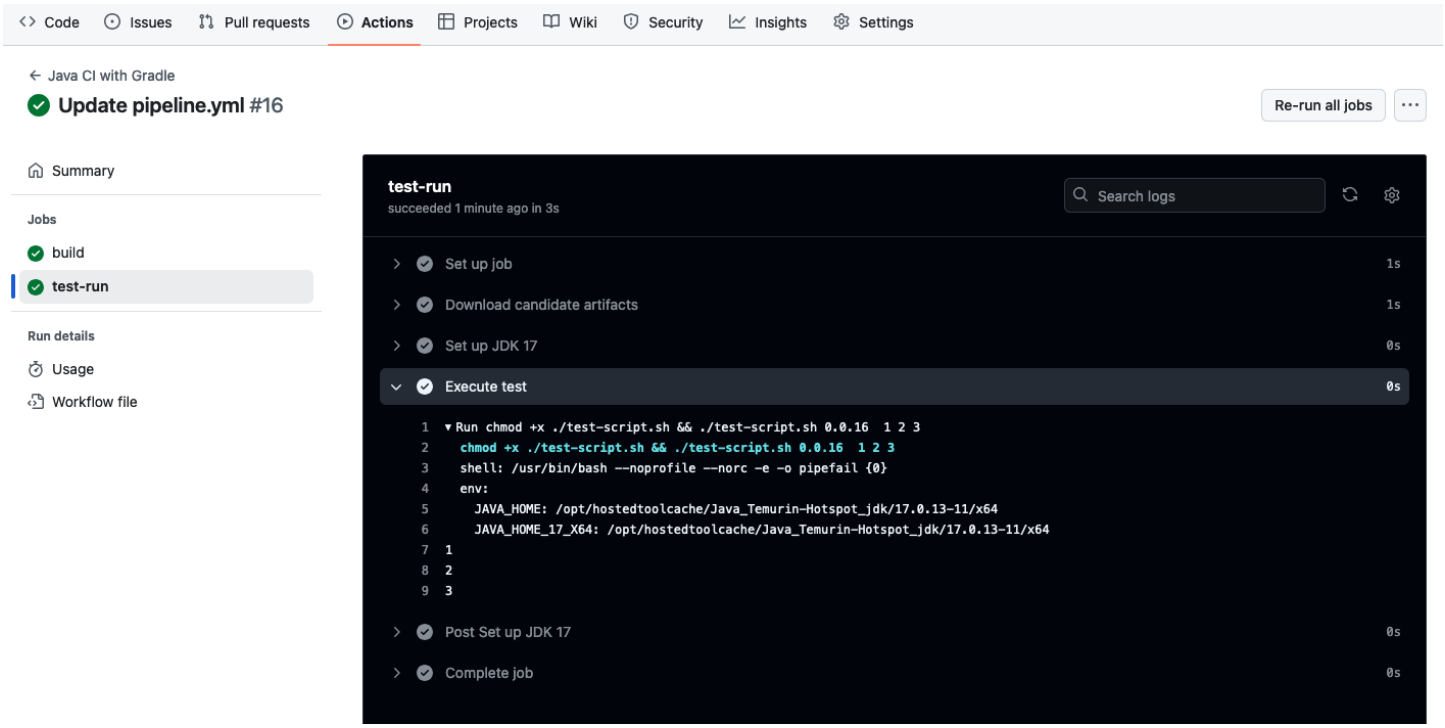
(Note: You can also get this code from https://gist.github.com/techupskills/abb65b4d56ddf8758aad34ecee8f62c4)

7. Commit the changes as before (to the "main" branch using the "Commit changes..." button).

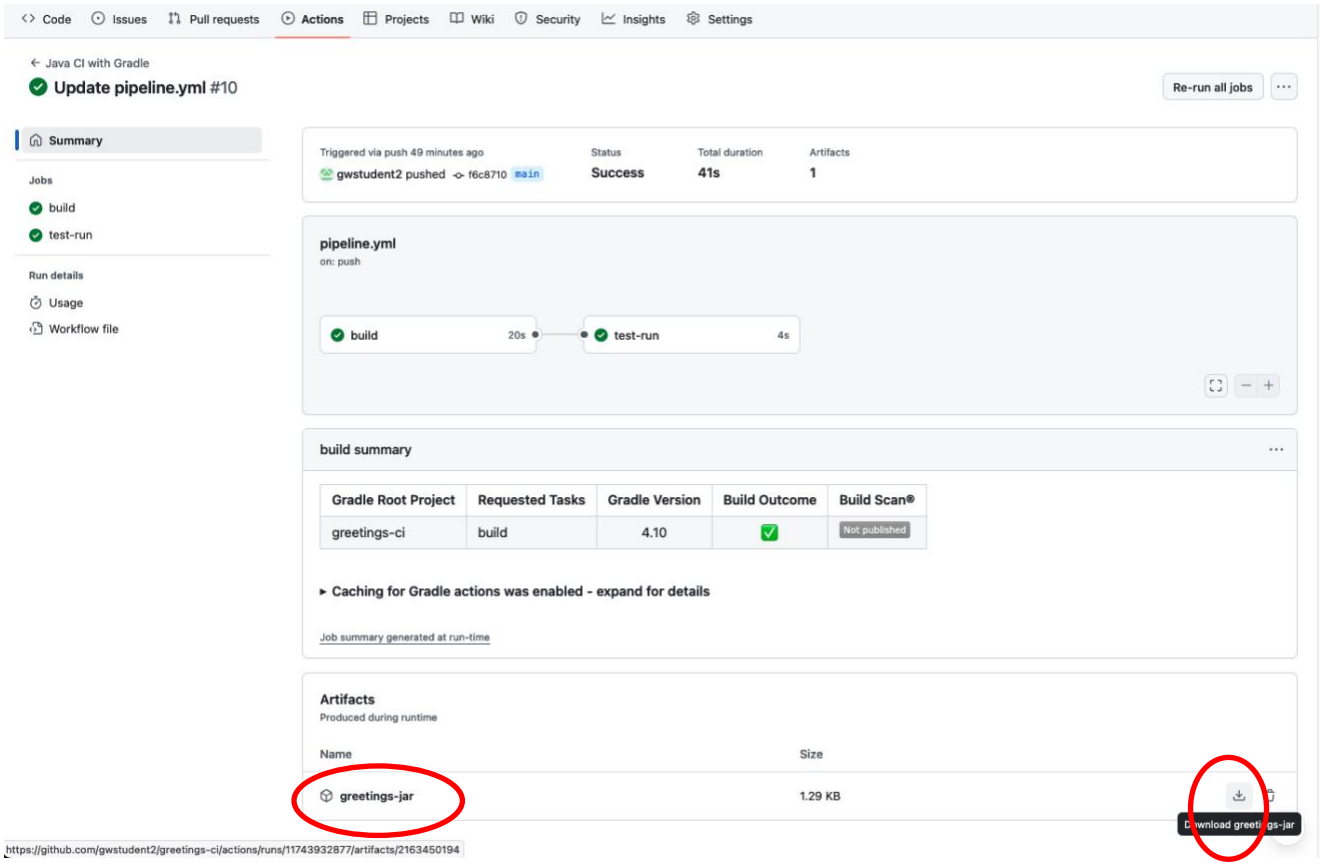
8. After the commit, if you switch to the *Actions* tab, you should see a new run of the workflow that executes both the *build* and *test-run* jobs. You can click on the run, then drill down via the job names to see what got executed.

Note: If you have any errors, go back and check the code, especially the alignment.

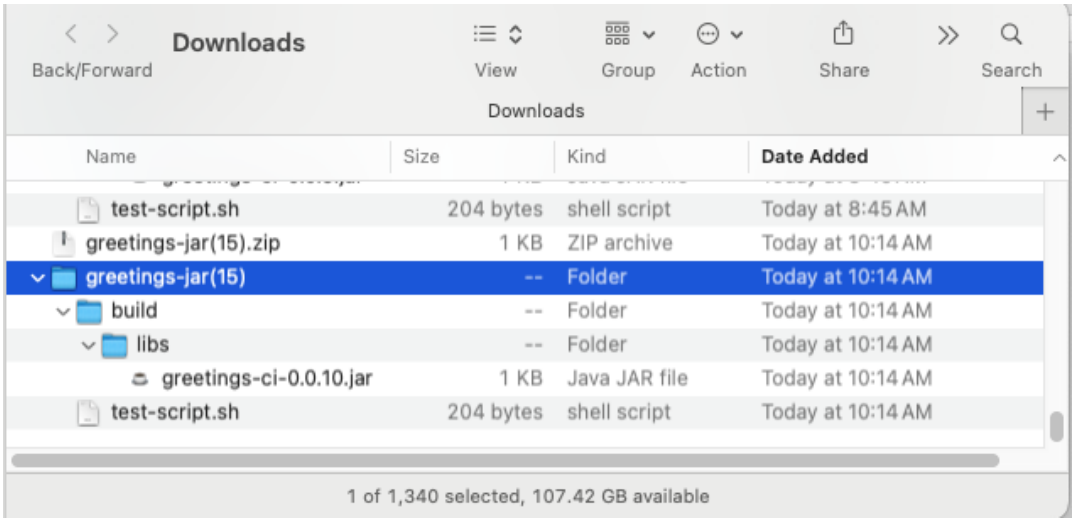




9. Go back to the page for the workflow run and scroll to the bottom. There you'll find our artifact named "greetings-jar". Click on either the download link or the artifact name (red circles) and download the artifact locally to your machine.



. 10. After downloading the artifact, you will have a zip file locally on your machine. Expand that and you should be able to see the test script, and also the versioned artifact.

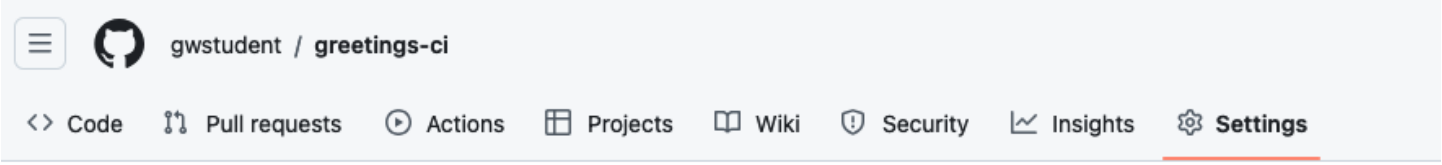


Lab 5: Working with fast feedback and automatically reporting issues

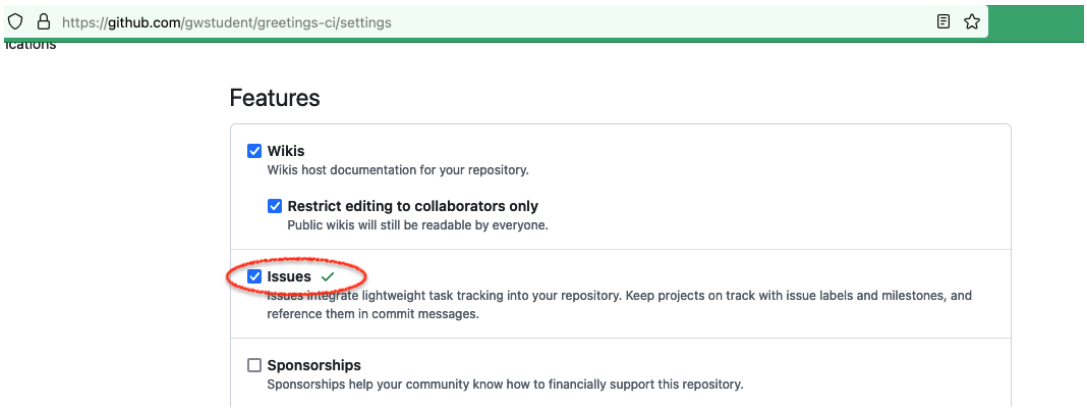
Purpose: Learning how to get fast feedback and automatic failure reporting in our pipeline

1. For this lab, we're going to leverage a reusable workflow that will be able to automatically create a GitHub issue in our repository. And then we will invoke that workflow from our current workflow. First though, we need to make sure that the *Issues* feature is turned on for your repository.

   Go to the *Settings* tab for your repository or to https://github.com/<github-userid>/greetings-ci/settings.



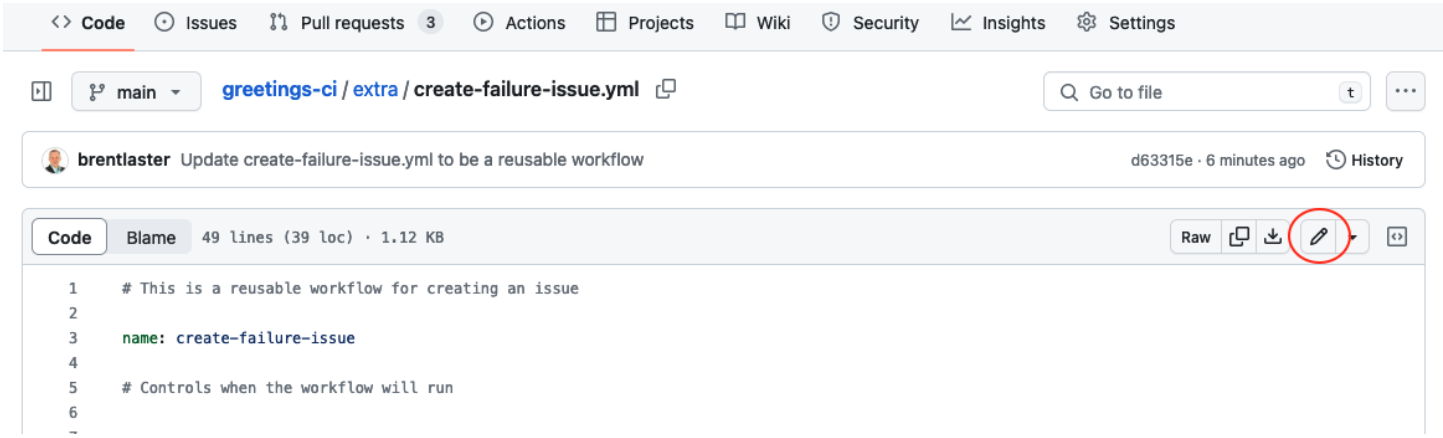
   Then scroll down on the *Settings* page, find the "Features" section and check the box to enable use of *Issues* on this repository.



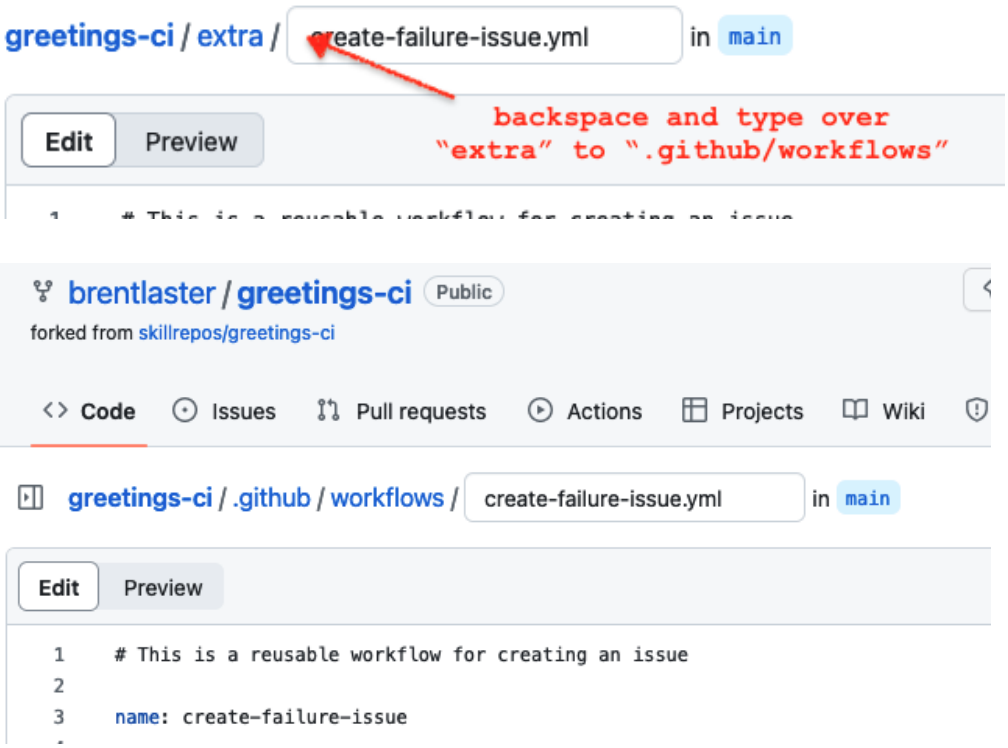
2. The workflow to create the issue using a REST API call is already written to save time. It is in the main project under "***extra/create-failure-issue.yml***". You need to get this file into the *.github/workflows* directory so it can be used as a workflow. You can just move it via a rename in GitHub with the following steps.

   a. In the repository, browse to the "extra" folder and open the "create-failure-issue.yml" file.
   b. Take a few moments to look over the file and see what it does. Notice that:
      i. It has a *workflow_call* section in the "on" area, which means it can be run from another workflow.
      ii. It has a *workflow_dispatch* section in the "on" area, which means it can be run manually.
      iii. It has two inputs - a title and body for the issue.
      iv. The primary part of the body is simply a REST call (using the GITHUB_TOKEN) to create a new issue.

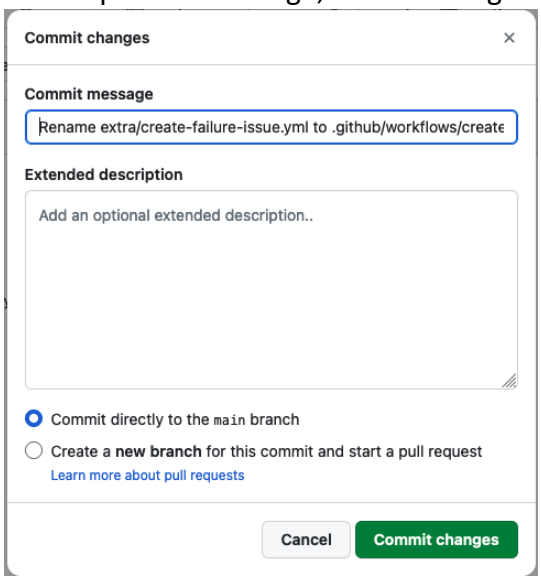
c. Click the pencil icon to edit it.



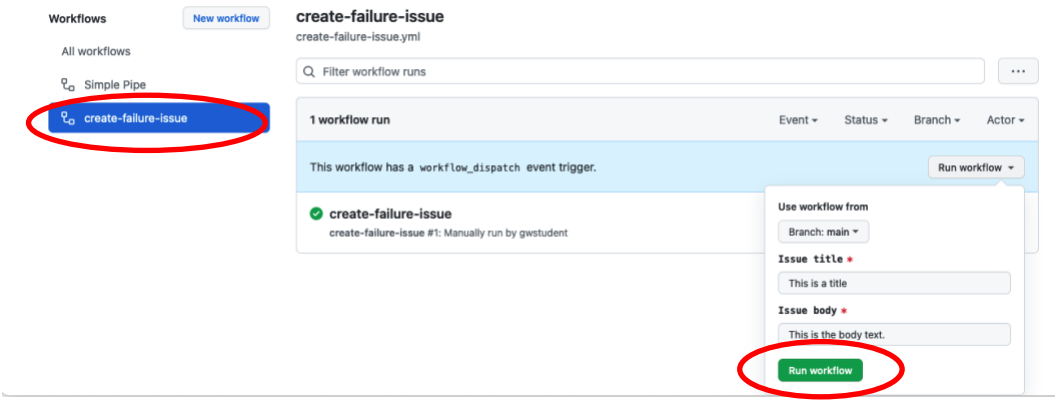
d. In the filename field at the top, change the name of file. Use the backspace key to backspace over "*extra/*" making sure to backspace over the word. Then type in the path to put it in the workflows *".github/workflows/create-failure-issue.yml*".



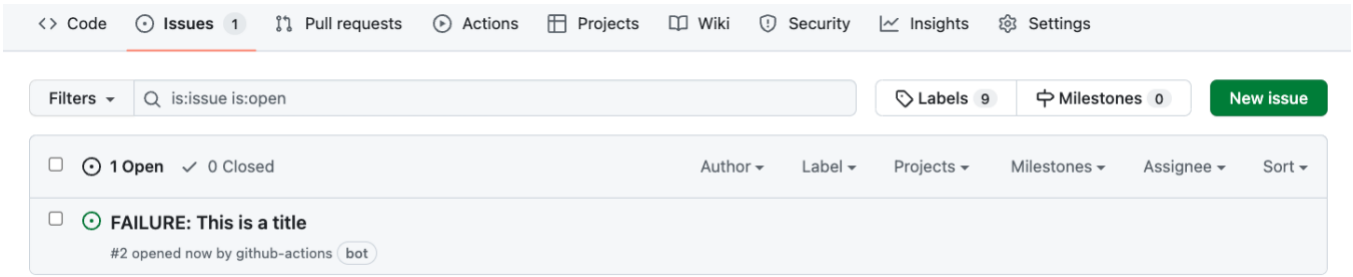
e. To complete the change, click on the green "Commit changes" button



3. Go back to the Actions tab. You'll see a new workflow execution due to the rename. Also, in the Workflows section on the left, you should now see a new workflow titled "***create-failure-issue***". Click on that. Since it has a *workflow_dispatch* event trigger available, we can try it out. Click on the "*Run workflow*" button and enter in some text for the "*title*" and "*body*" fields. Then click "*Run workflow*".



4. After a moment, you should see the workflow run start and then complete. If you now click on the Issues tab at the top, you should see your new issue there.



5. Now that we know that our new workflow works as expected, we can make the changes to the previous workflow to "call" this if we fail. Edit the pipeline.yml file and add the following lines as a new job and set of steps at the end of the workflow. (For convenience, these lines are also in the file "*extra/job-create-issue-on-failure.txt*" if you want to copy and paste from there.). The "*create-issue-on-failure*" line should line up with the other job names.

   We have some other changes to make so don't commit yet.

```
create-issue-on-failure:

  needs: test-run

  permissions:
    issues: write

  if: always() && failure()
  uses: ./.github/workflows/create-failure-issue.yml
  with:
    title: "Automated workflow failure issue for commit ${{ github.sha }}"
    body: >
      "This issue was automatically created by the workflow ${{ github.workflow }}"
```

```
68          uses: actions/setup-java@v4
69          with:
70            java-version: '17'
71            distribution: 'temurin'
72        - name: Print Inputs
73          shell: bash
74          run: echo github.event.inputs.myTestArgs = "${{ github.event.inputs.myTestArgs }}"
75
76        - name: Execute test
77          shell: bash
78          run: >
79            chmod +x ./test-script.sh &&
80            ./test-script.sh 0.0.${{ github.run_number }}
81            ${{ github.event.inputs.myTestArgs || '1 2 3' }}
82
83    create-issue-on-failure:
84
85      needs: test-run
86
87      permissions:
88        issues: write
89
90      if: always() && failure()
91      uses: ./.github/workflows/create-failure-issue.yml
92      with:
93        title: "Automated workflow failure issue for commit ${{ github.sha }}"
94        body: "This issue was automatically created by the workflow ${{ github.workflow }}"
```

6. Let's make one more change to make it easier to run our workflow manually to try things out, start runs, etc. In the "on:" section near the top, add the code below at the bottom of the "on" section. ("workflow_dispatch" should line up with "pull" and "push") and then commit the changes.

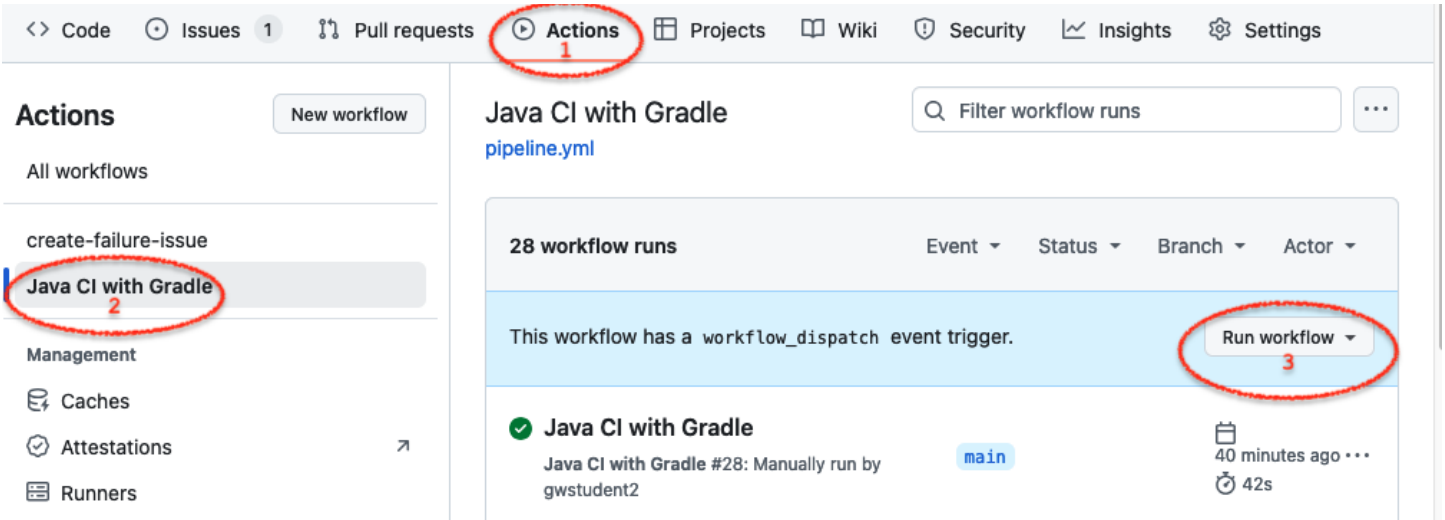
```
workflow_dispatch:
  inputs:
    myTestArgs:
      description: 'Testing values'
```

```yaml
name: Java CI with Gradle

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
  workflow_dispatch:
    inputs:
      myTestArgs:
        description: 'Testing values'


jobs:
  build:

    runs-on: ubuntu-latest
    permissions:
      contents: read
```

7. Commit the changes back to the main branch. This will cause a new workflow run (that should succeed) that you can ignore.

8. The addition of the *workflow_dispatch* trigger means that you can now run the workflow manually and input some test arguments to override the "1 2 3" ones we have for a default. To see how this works, change to the *Actions* tab and then click on the left side under "*All workflows*", click on the "*Java CI with Gradle*" workflow link.

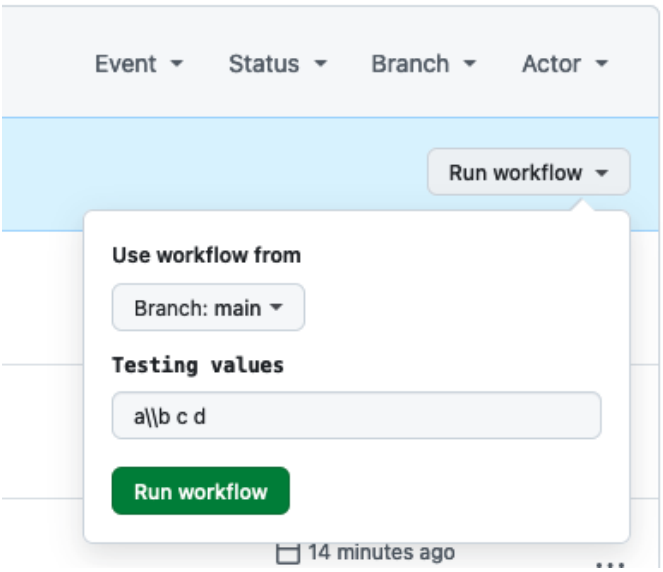
   A blue bar will show up with a "*Run workflow*" button to the right. Click on the "*Run workflow*" button.



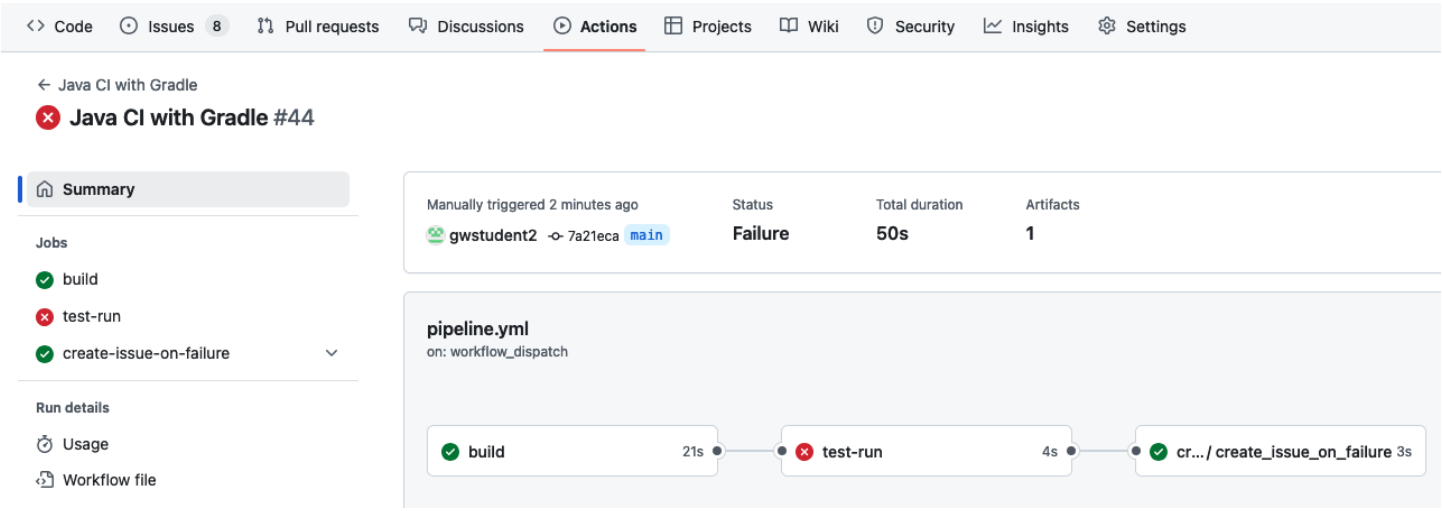
9. For the test argument values we'll put in ones that will cause an issue so that our automatic issue creation will be triggered. To do this, we'll include two backslashes in the input (they can be anywhere, but will cause the test job to fail, and thus cause an issue to be generated.

   So, in the dialog that comes up from the *Run workflow* button, enter in something like the string below and then click on the *Run workflow* button.

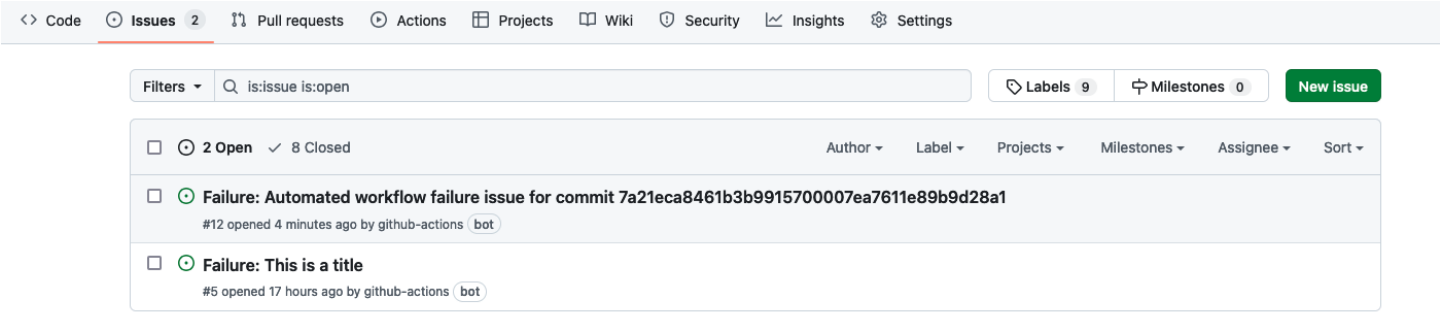
   a\\ b c d



10. After this, the workflow will run, the *test-run* job should fail, and the *create_issue_on_failure* job should succeed in detecting the failure. You'll be able to see this by looking at the job graph.



11. You should also be able to look under the *Issues* tab and see the new issue that resulted.

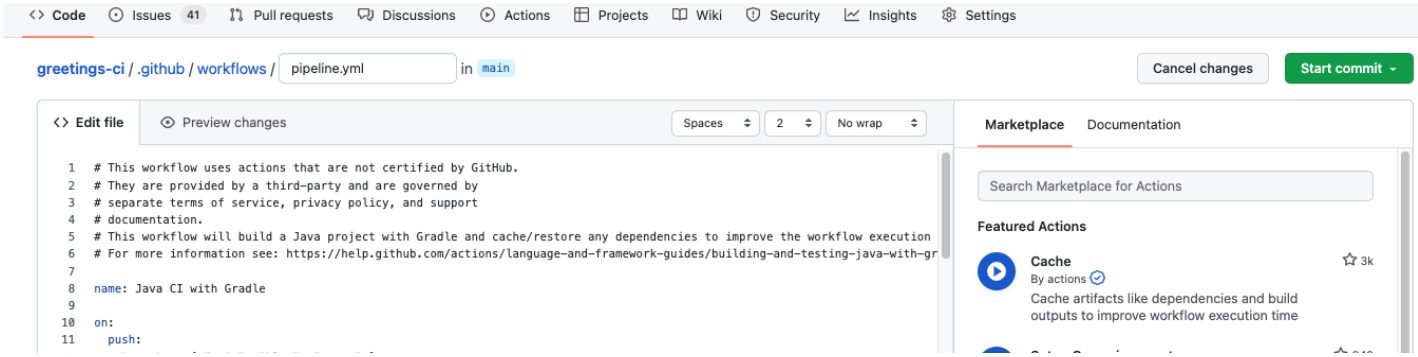


END OF LAB

Lab 6 – Adding Environments and Releases

Purpose: In this lab, we'll look at how to add staging (blue, green) and production environments and releases.

1. Let's add some deploy jobs to our pipeline.yaml file. **Edit the .github/workflows/pipeline.yaml file.** For simplicity, we can just do this **in the main branch**.



2. We can illustrate blue/green deployment with new branches such as "blue" and "green". So, let's modify the "on:" section first to run the workflow on a push to any of these. Modify the **on: push:** command to be like the following.

```
on:
  push:
    branches: [ "main", "blue", "green" ]
```

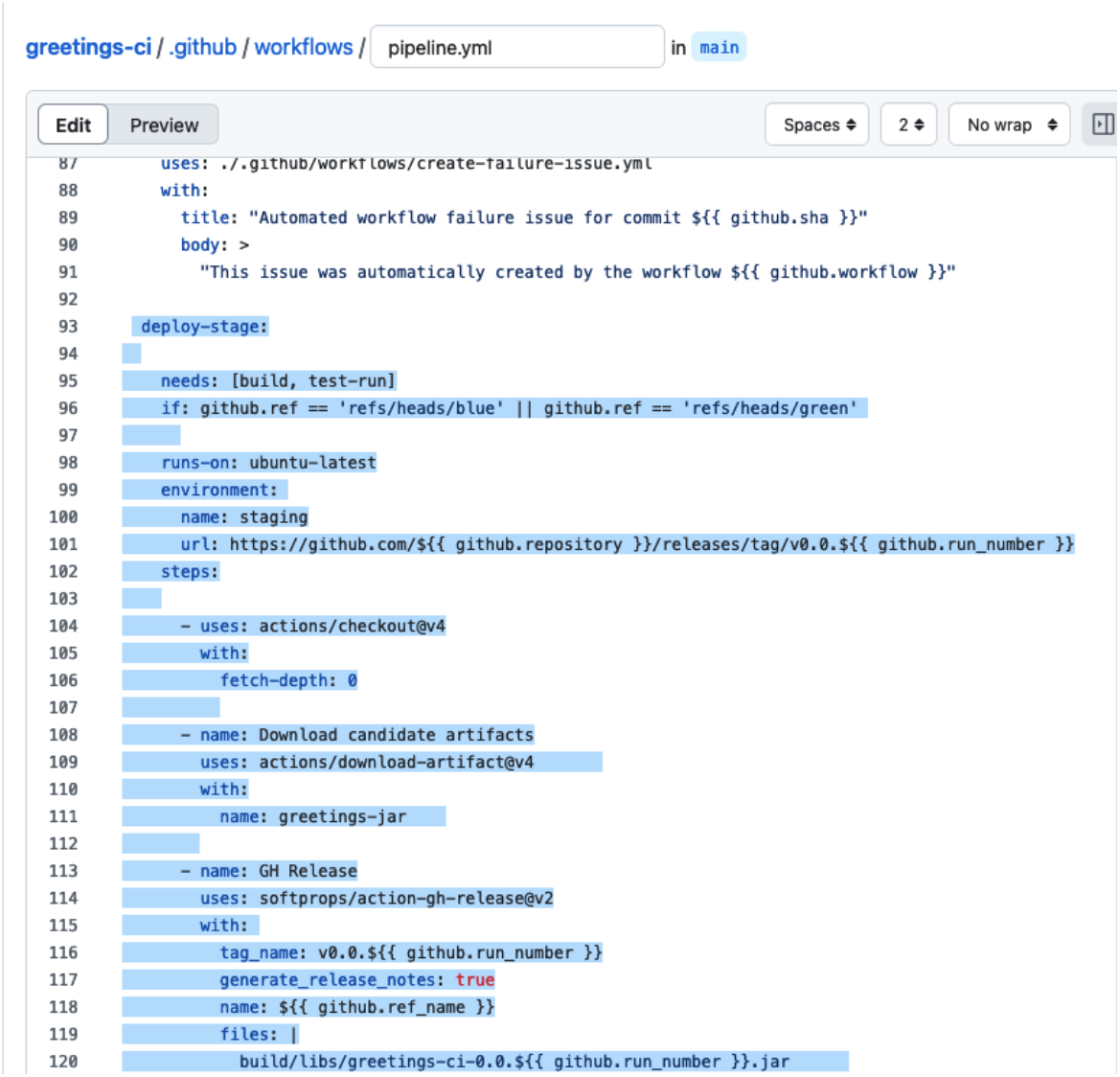
```
 7
 8  name: Java CI with Gradle
 9
10  on:
11    push:
12      branches: [ "main", "blue", "green" ]
13    pull_request:
14      branches: [ "main" ]
```

3. You can also remove the "pull_request" portion.

```
9
10    on:
11      push:
12        branches: [ "main", "blue", "green" ]
13      workflow_dispatch:
14        inputs:
15          myTestArgs:
16            description: 'Testing values'
17
```

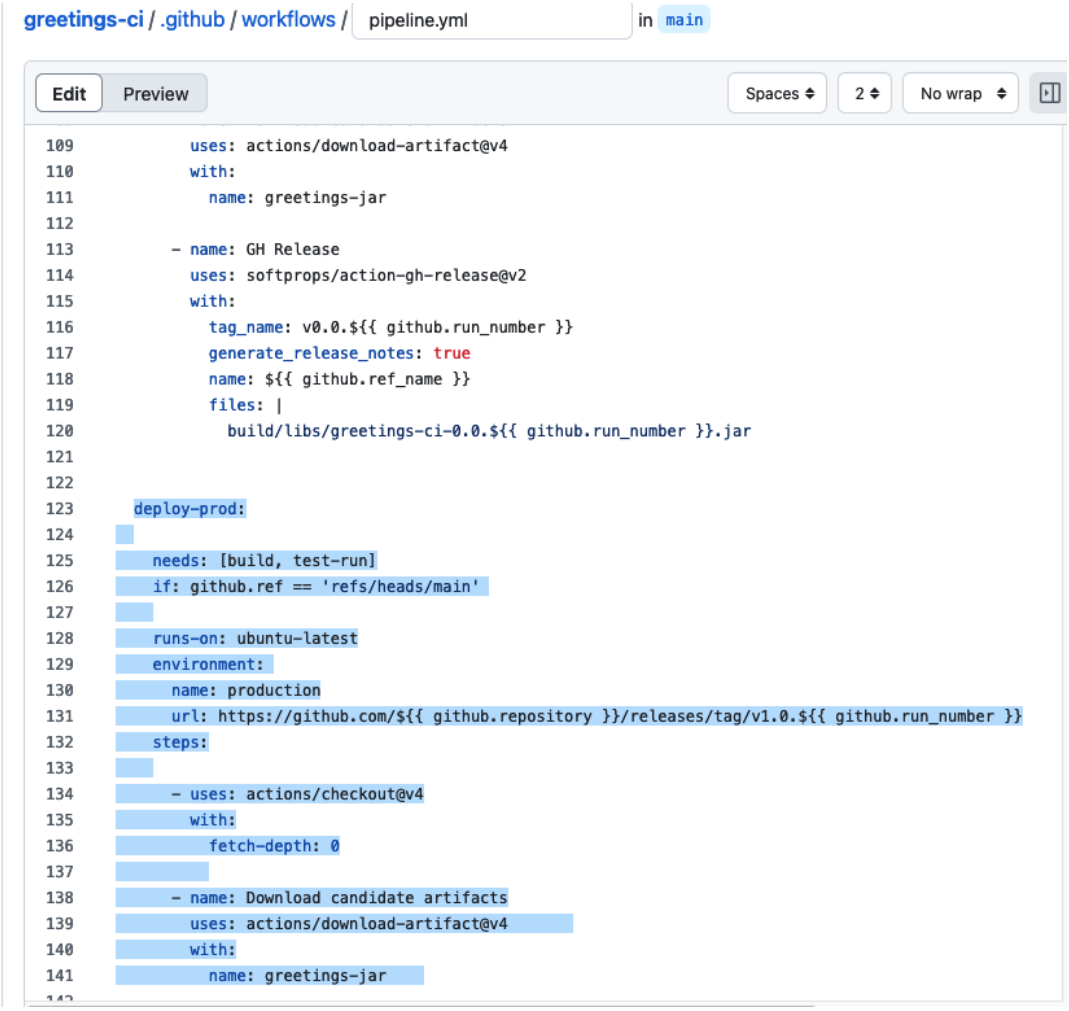
4. Now, let's add the job for deploying a "stage" environment/release. The code for this job is already done for you and can be copied from the file extra/deploy-stage.txt You can go to the same repository in another tab, open that file, and then just copy and paste.

(Note: You may need to add a space or two at the front of the first line of output once pasted to get it to line up correctly.)

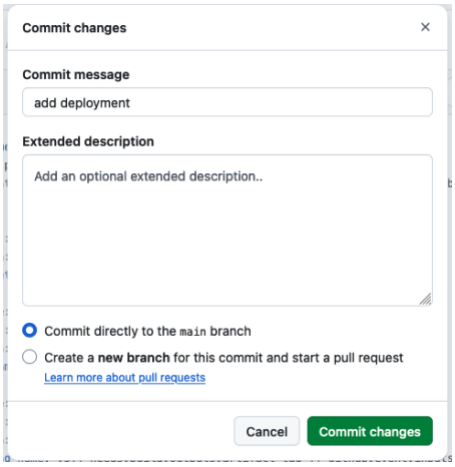


5. Now, let's add the job for deploying a "prod" (production) environment/release from a pull-request being merged into "main". The code for this job is already done for you and can be copied from the file extra/deploy-prod.txt Just copy and paste. You can copy and paste this one the same way from the other tab.

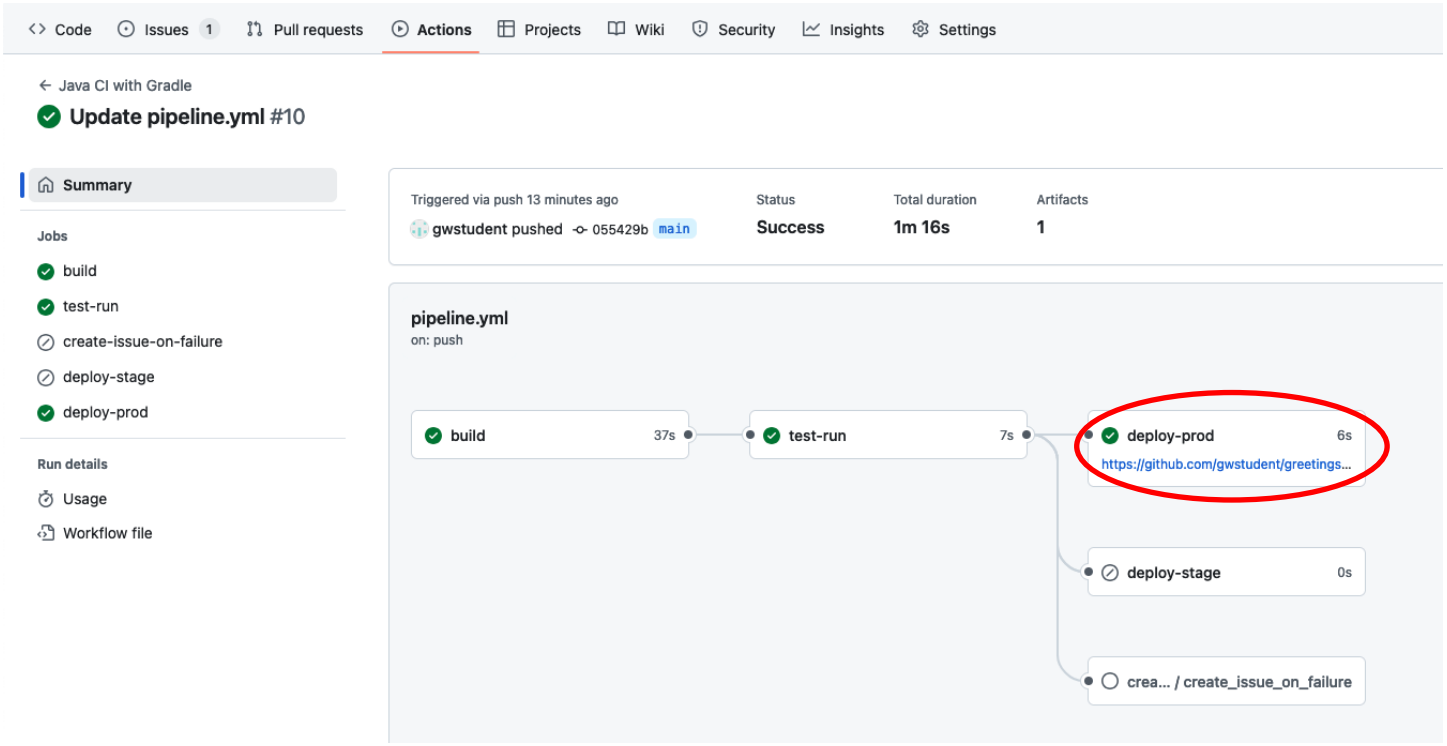
(Note: You may need to add a space or two at the front of the first line of output once pasted to get it to line up correctly.)

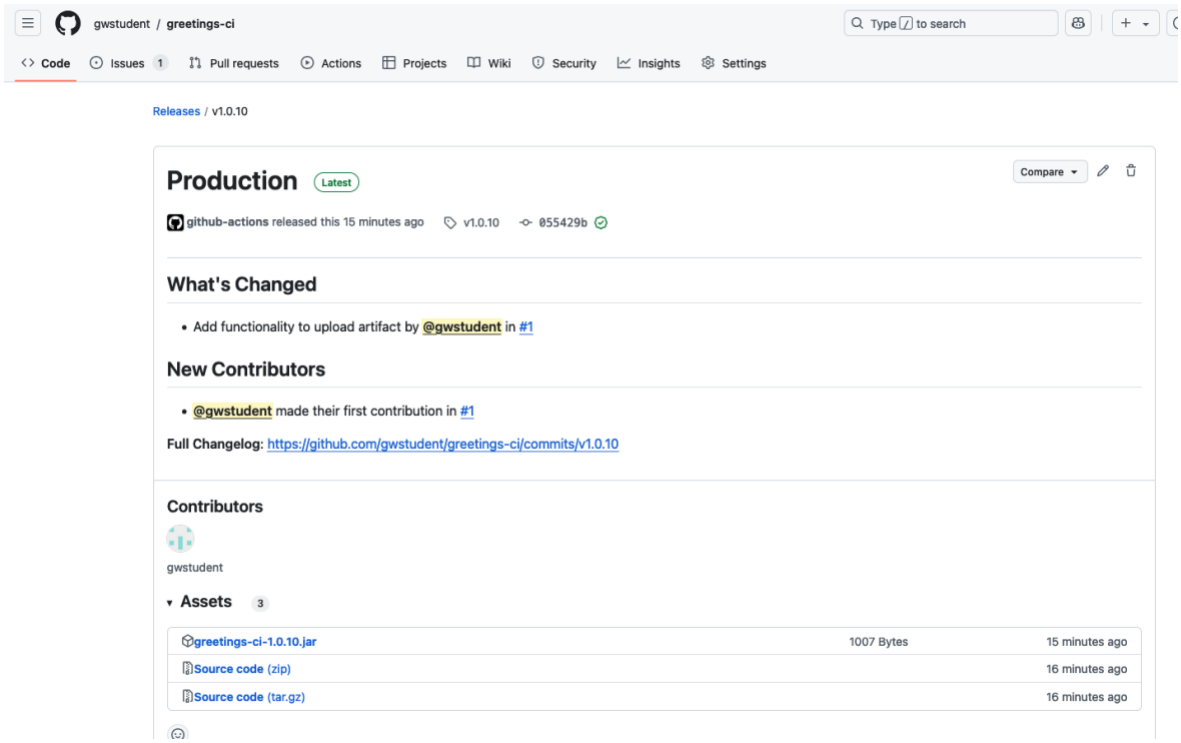


6. Go ahead and commit your changes **to the main branch** with an appropriate commit message.



7. This will kick off a new run of the workflow and will create an initial production deployment because of a change in main. After the run completes, you can click on the link in the deploy-prod job in the "Jobs" view to see the release.



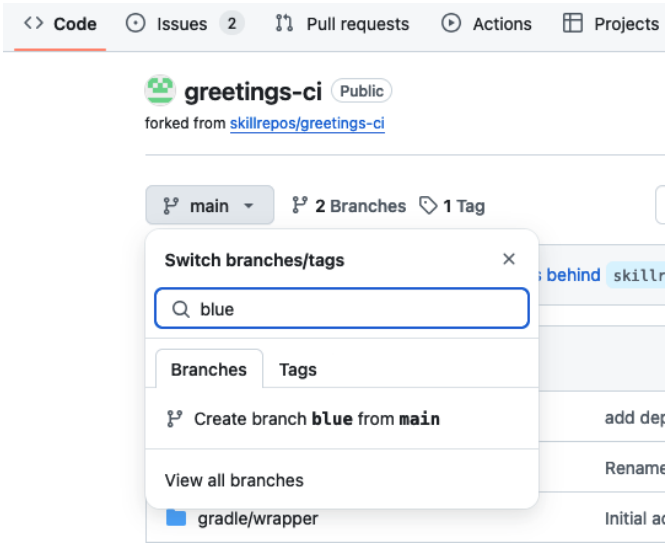


END OF LAB

Lab 7 – Exercising the entire workflow

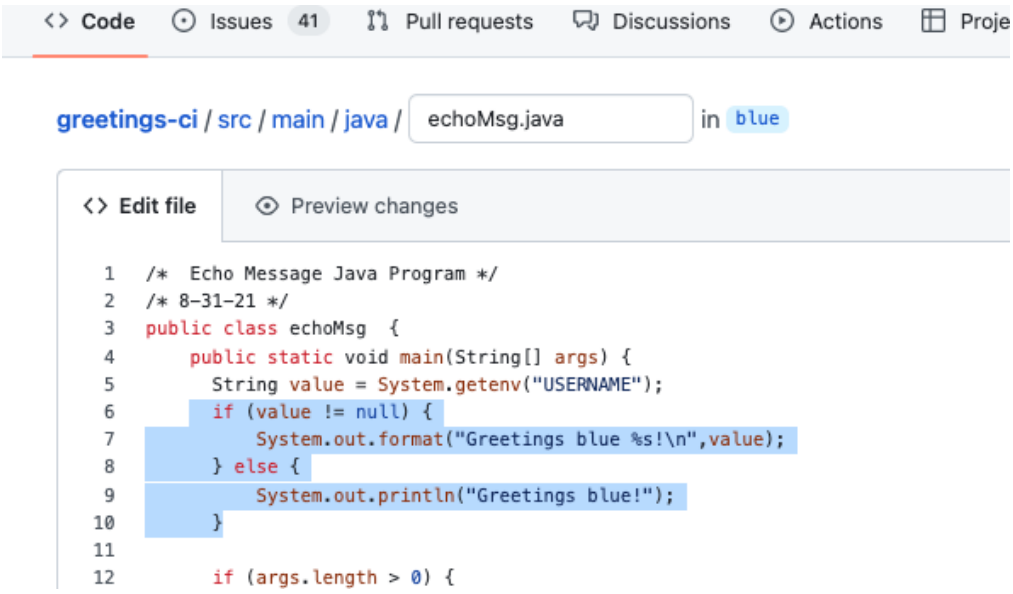
Purpose: In this lab, we'll see how to make a change in source code and have it processed through the pipeline.

1. For the example of using a "blue/green" environment, let's **create a branch called "blue" from the "main" branch** to make some changes on. Do this just as you've done before. (Note that you will need to make sure you're on the main branch before creating the blue one. Make sure "main" shows up and you are on the "branches" tab, not the "tags" tab.)

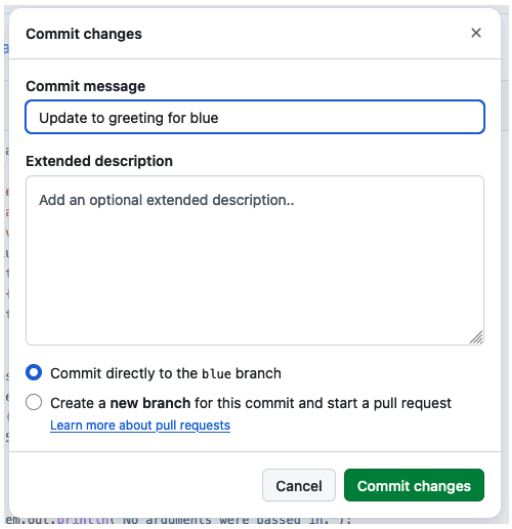


2. Make sure you are in the "*blue*" branch, and edit the file "*src/main/java/echoMsg.java*". Make a simple, non-breaking change like adding "blue" to the lines that print out "Greetings". See text and figure below.

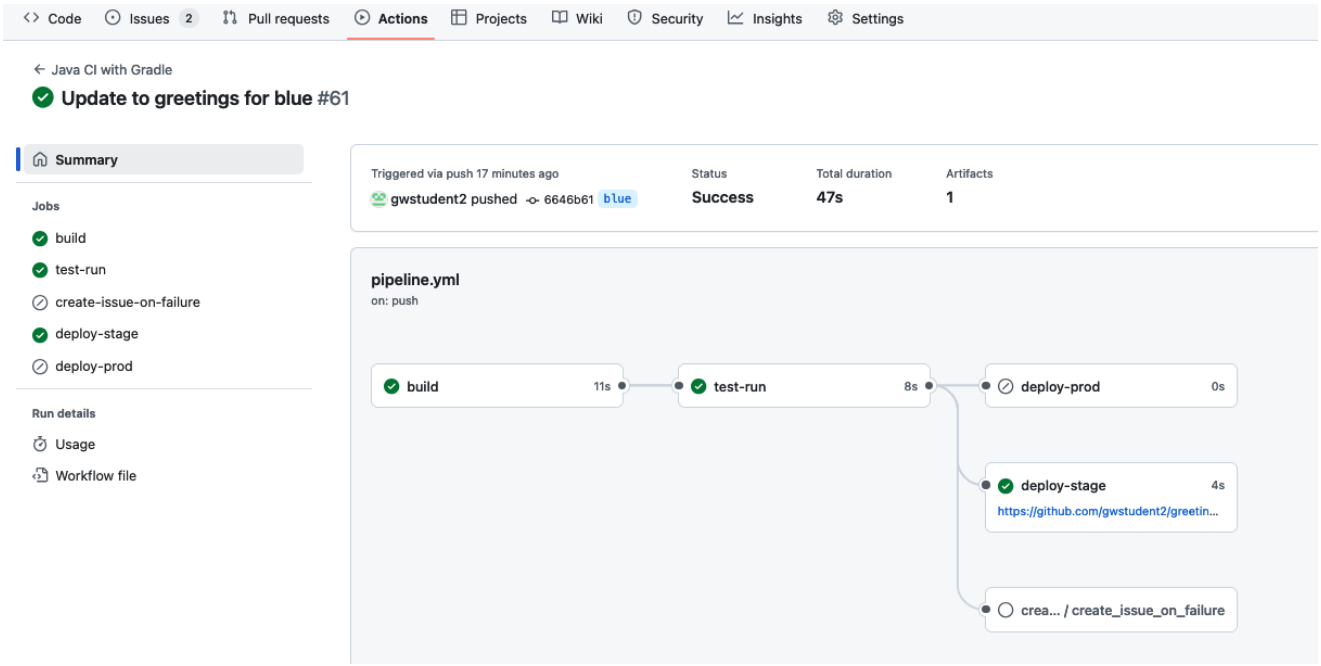
```
if (value != null) {
    System.out.format("Greetings blue %s!\n",value);
} else {
    System.out.println("Greetings blue!");
}
```



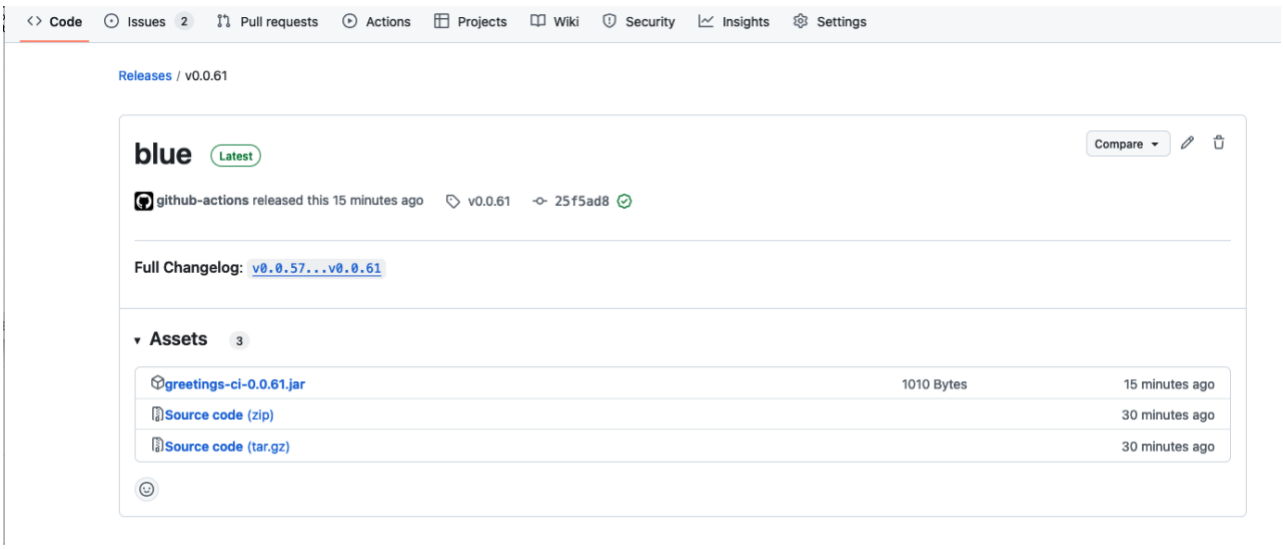
3. Commit the changes with an appropriate commit message.



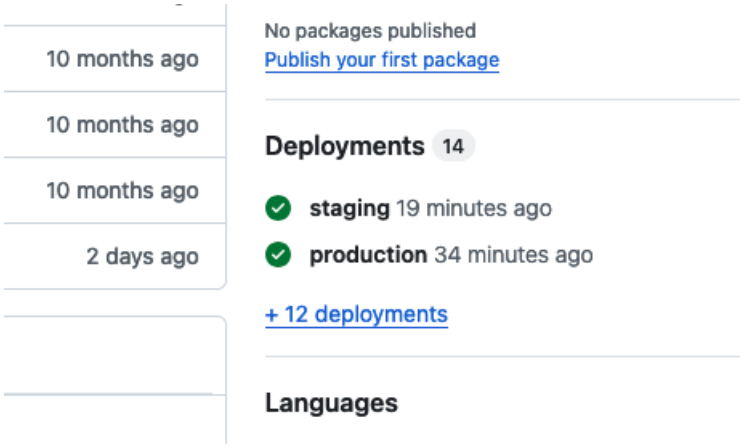
4. After the workflow run completes, you can click on the run and look at the job graph. You should be able to see that it executed the build and test pieces and then deployed it to the stage environment.

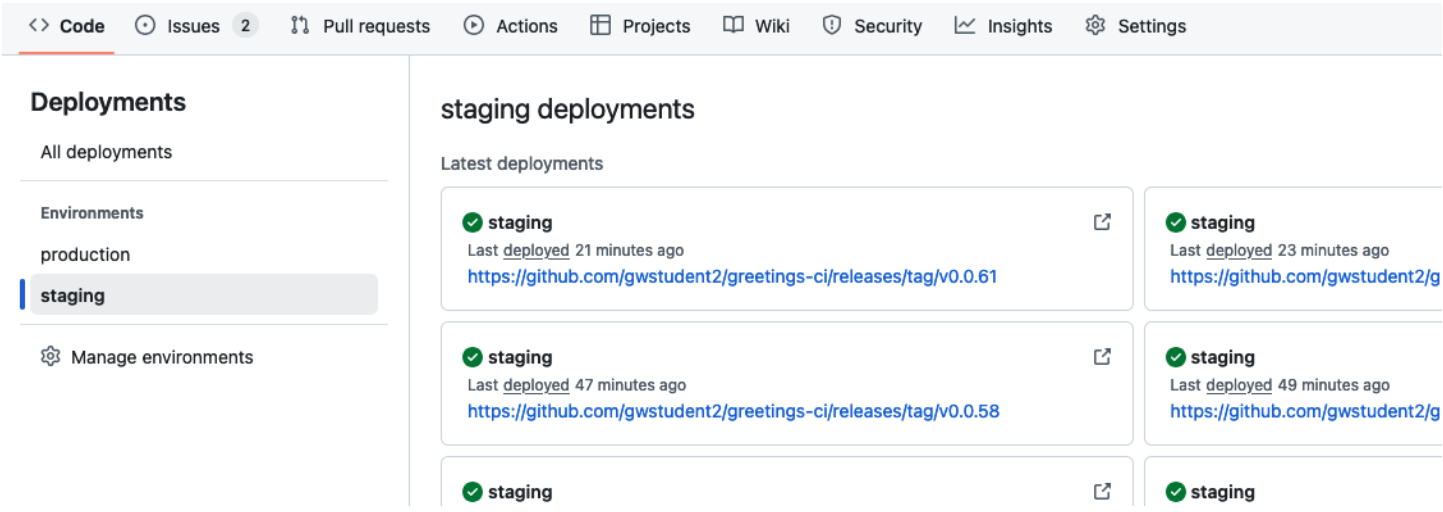


5. Now, click on the **link** in the "deploy-stage" box. This will take you to the staging release page for the *blue* deployment.



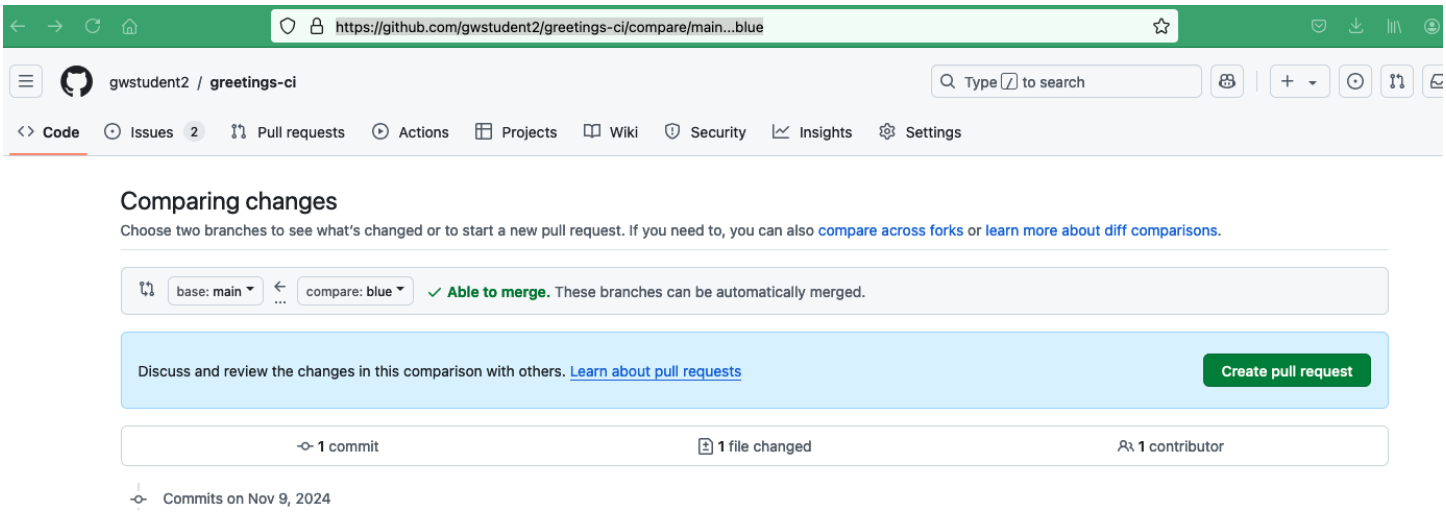
6. And, if you click on the main code page, in the lower right, you'll be able to see a new "staging" deployment. You can click on that to see a list of recent deployments there.



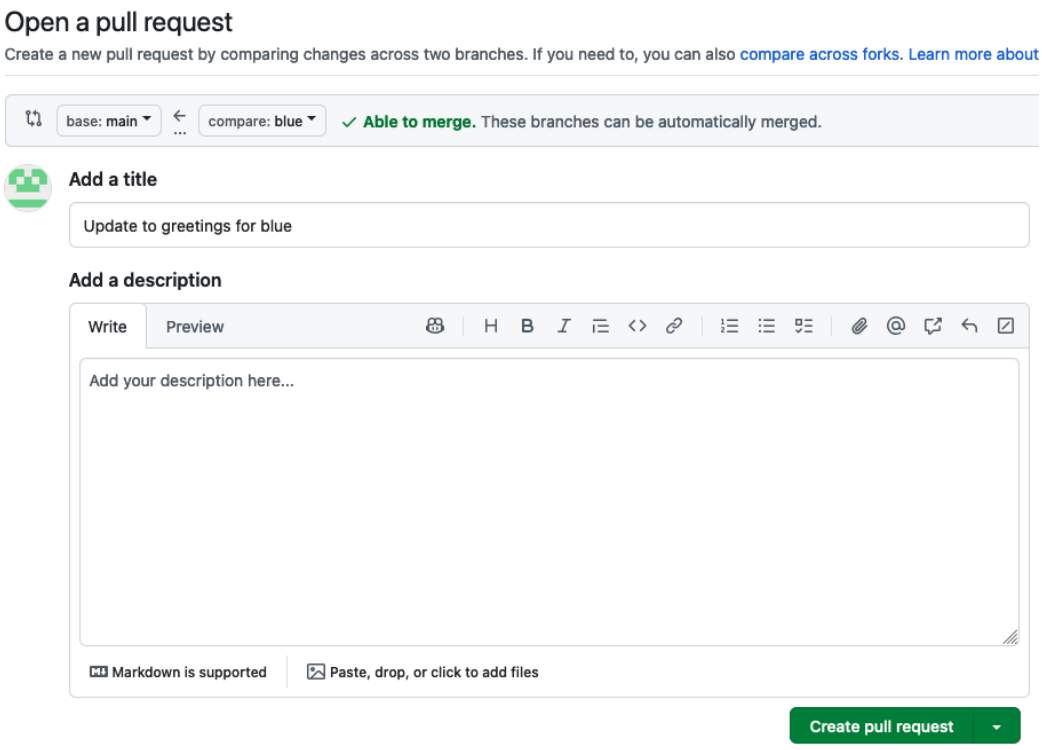


7. Since everything built ok, we can deploy this change to the production environment. To merge the changes, we can just create a pull request to main and merge it. To keep things simple, here's a link that you can copy and paste (substituting in *your GitHub userid* for *<github-userid>*)

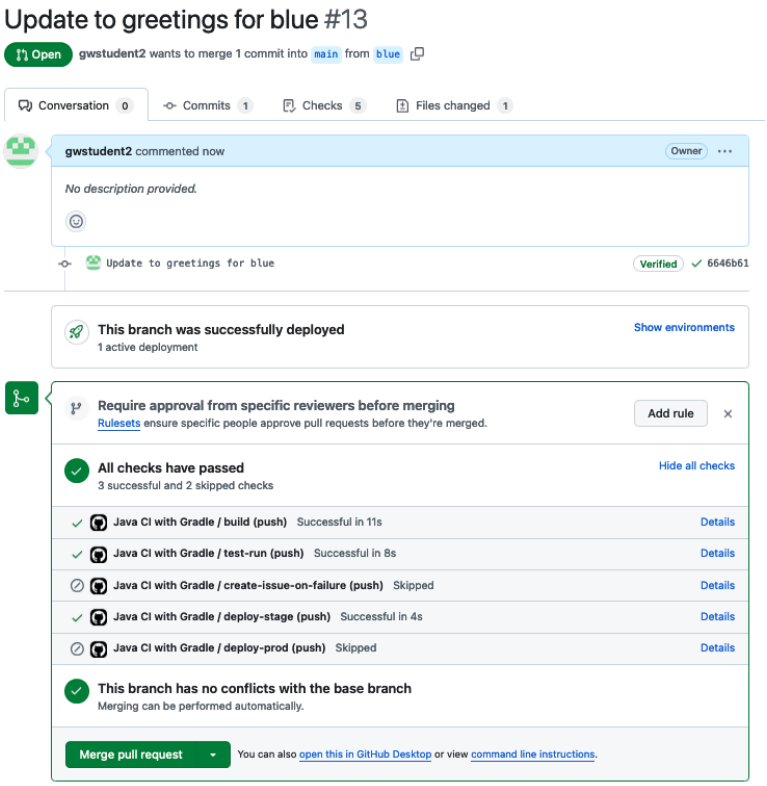
https://github.com/<github-userid>/greetings-ci/compare/main...blue



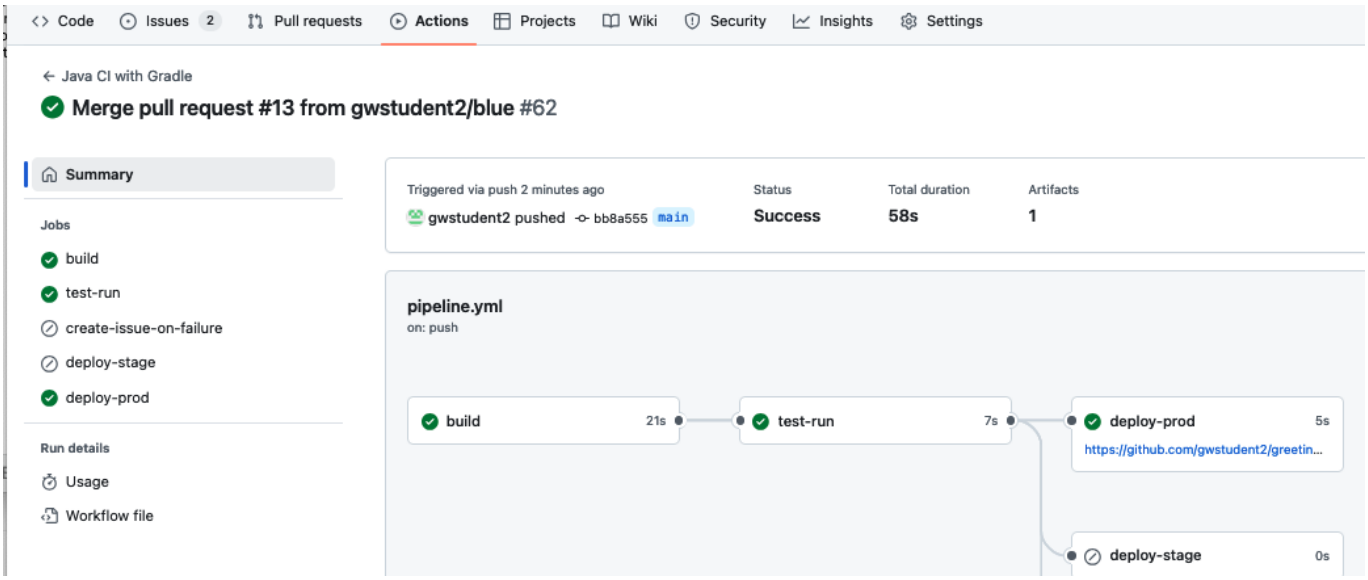
8. Click on the *Create pull request* button to start the new pull request. Then, click on the following *Create pull request* button to open the request.



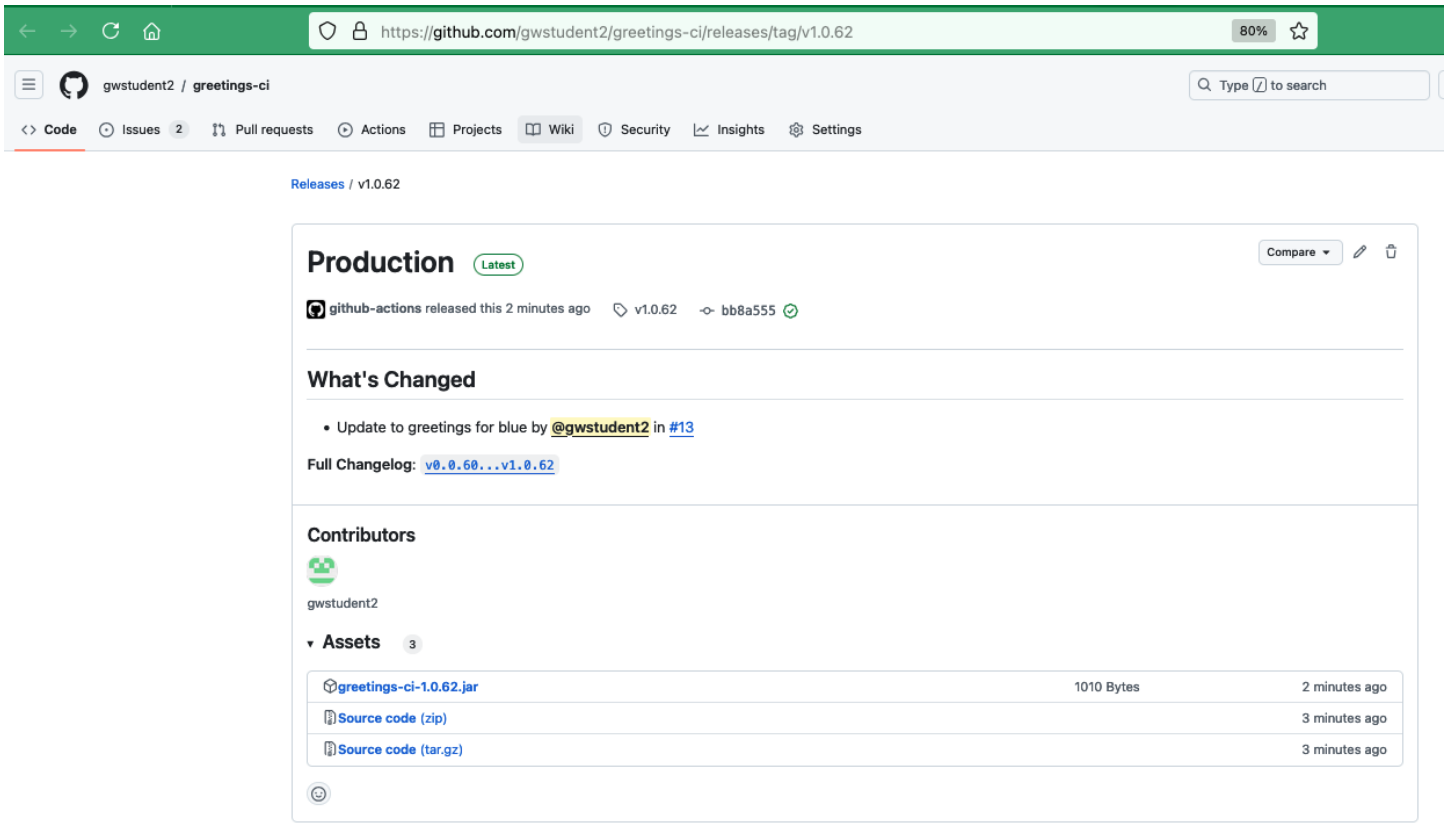
9. You should then be on a page for the pull request that shows all the checks. You can just go ahead and merge and confirm.



10. This should kick off another run of the action workflow in main. When it is done, it should have done a "production" deployment as you can see via the jobs graph for that run.



11. If you click in the link for the deploy-prod job, you should be able to go to the new *Production* deployment page.



END OF LAB

THANKS!

Appendix 1: Alternate ways to "fork" repo if not allowed to use actual "Fork" button

OPTION 1: Using Import

1. Sign into GitHub if not already signed in.

2. Go to https://github.com/new/import

3. On that page, fill out the form as follows:

In "Your source repository details", in the "The URL for your source repository *" field, enter

https://github.com/skillrepos/greetings-ci

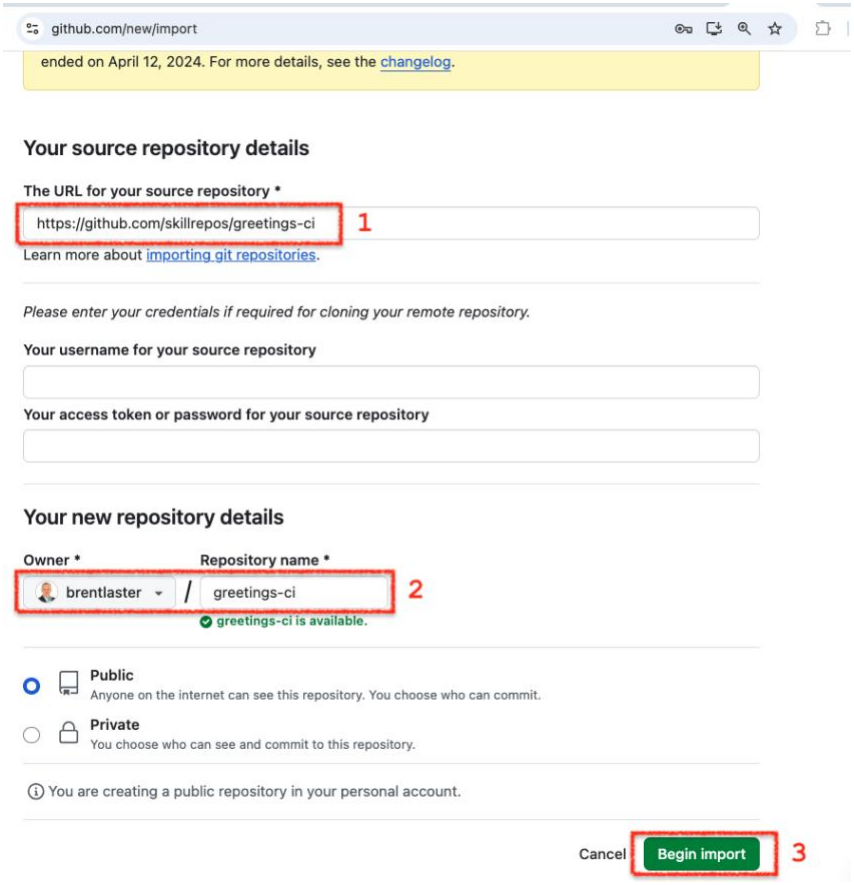
Under "Your new repository details", make sure your userid shows up in the "Owner *" field and enter

greetings-ci

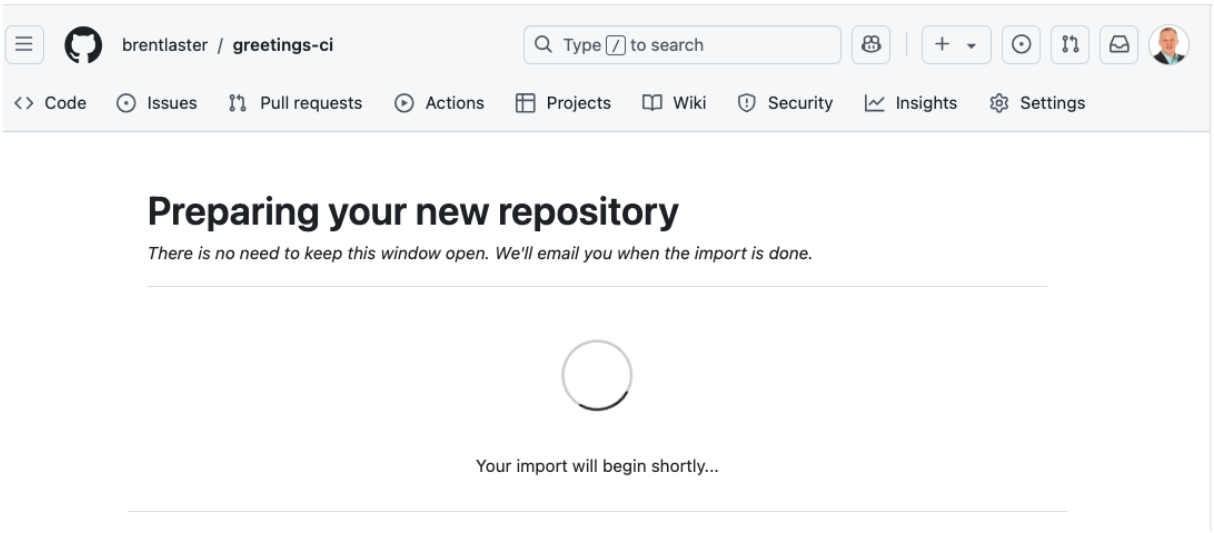
in the "Repository name *" field.

The visibility field should be set to "Public".

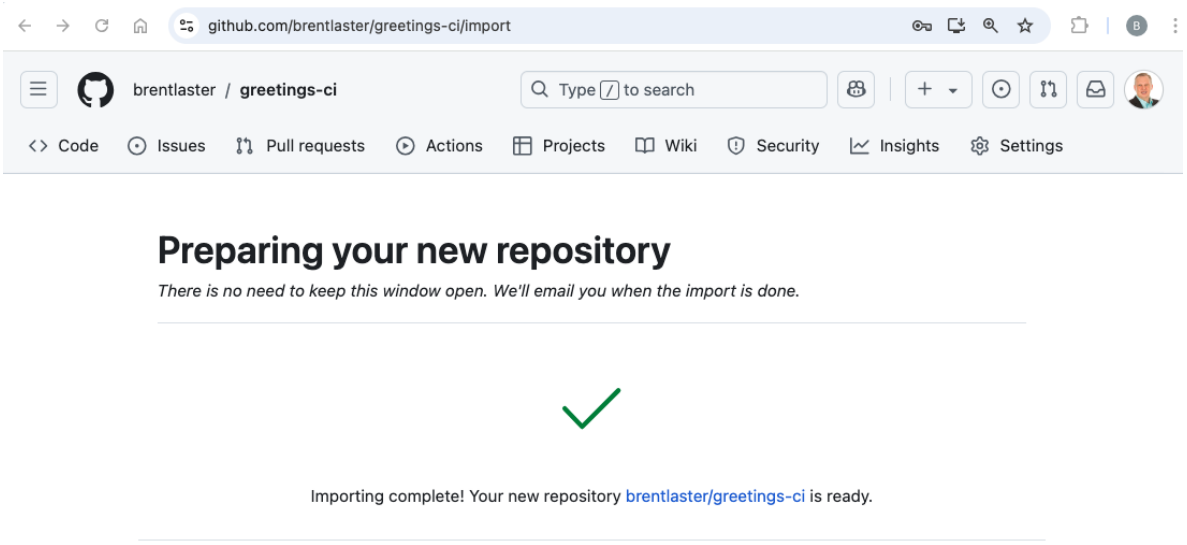
Then click on the green "Begin import" button.



4. After this, you should see the import processing...



5. This will take several minutes to run. When done, you should see a "complete" message and your new repo will be available.



OPTION 2: Using clone and push

1. Sign into GitHub if not already signed in.

2. Create a GitHub token or SSH key. If you are familiar with SSH keys, you can add your public key at https://github.com/settings/keys. Otherwise, you can just create a "classic" token by following the instructions at https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens#creating-a-personal-access-token-classic. If you use a GitHub token, make sure to save a copy of it to use in the push step.

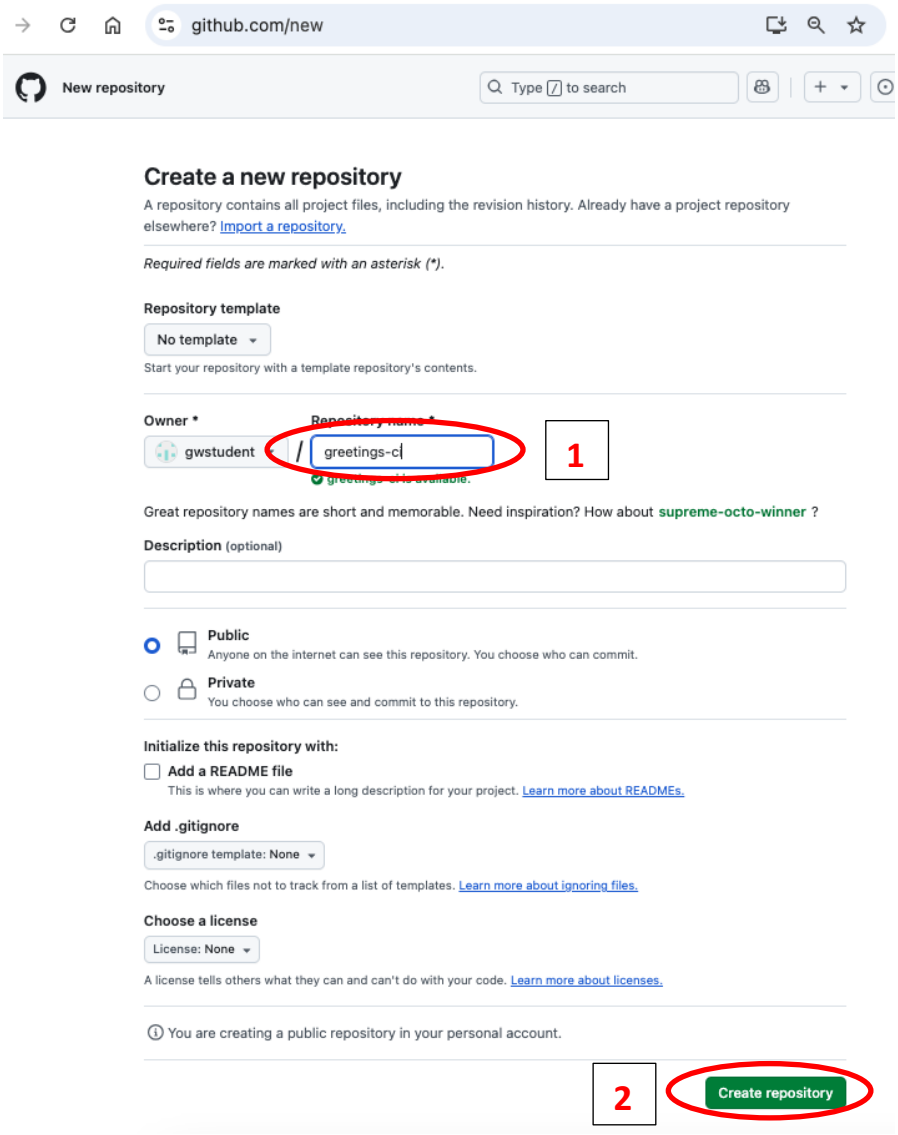
3. Clone down the skillrepos/greetings-ci repository.

git clone https://github.com/skillrepos/greetings-ci (if using token)

or

git clone git@github.com:skillrepos/greetings-ci (if using ssh)

3. Create a new repository in your GitHub space named greetings-ci. Go to https://github.com/new. Fill in the "repo name" field with "greetings-ci" and then click on the "Create repository" button.



4. On the page that comes up after that, select the appropriate protocol (https or ssh) and then follow the instructions for "...or push an existing repository from the command line" to push your content back to the GitHub repository. If you're using https you will be prompted for a password at push time. Just paste in the classic token. (Note that for security reasons, you will not see the token displayed.)