GitHub Actions Deep Dive

Revision 2.3 – 07/17/23

Tech Skills Transformations LLC / Brent Laster

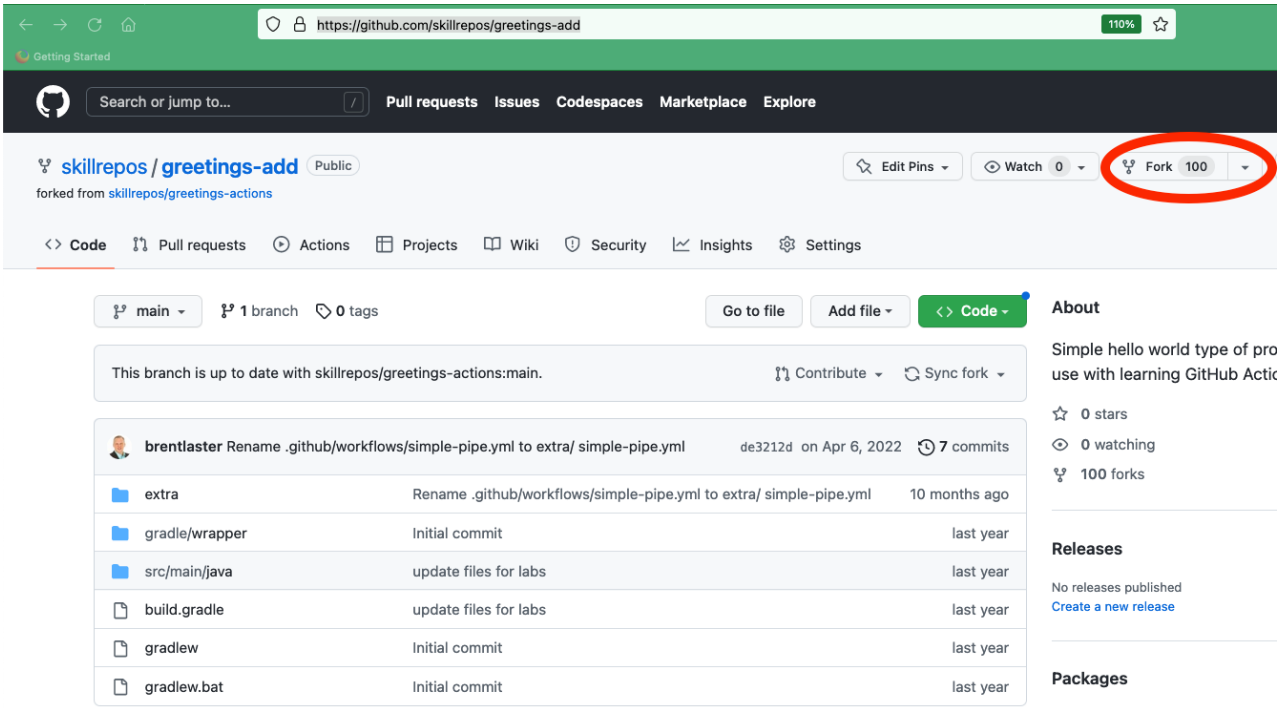
Important Prerequisite: You will need a GitHub account for this. (Free tier is fine.)

Lab 1 – Creating a simple example

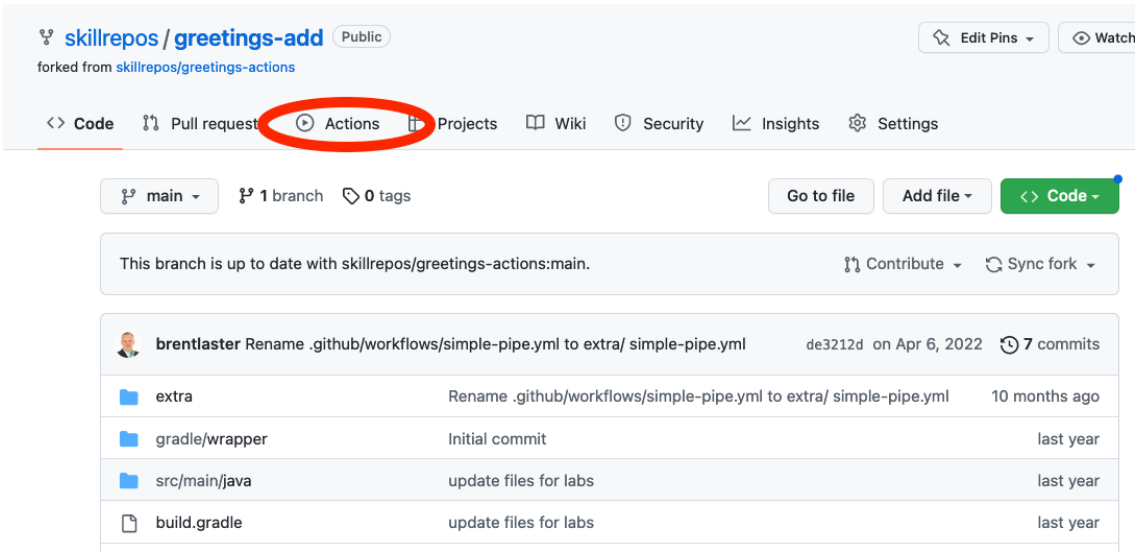
Purpose: In this lab, we'll get a quick start learning about CI with GitHub Actions by creating a simple project that uses them. We'll also see what a first run of a workflow with actions looks like.

1. Log in to GitHub with your GitHub ID.

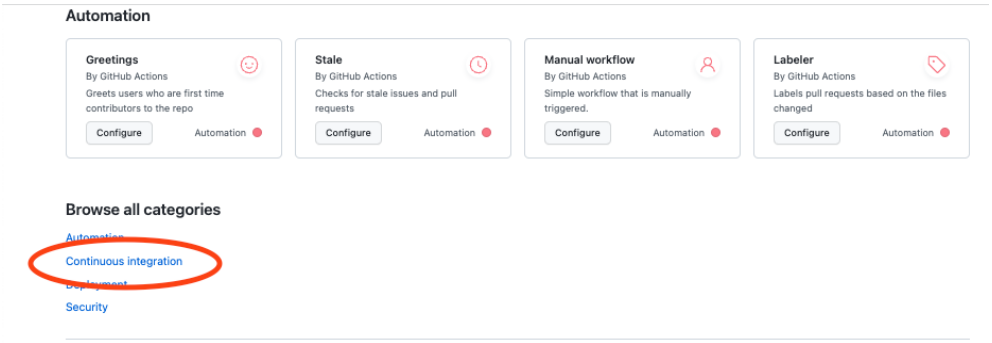
2. Go to https://github.com/skillrepos/greetings-add and fork that project into your own GitHub space. You can accept the default options for the fork and click the "Create fork" button.



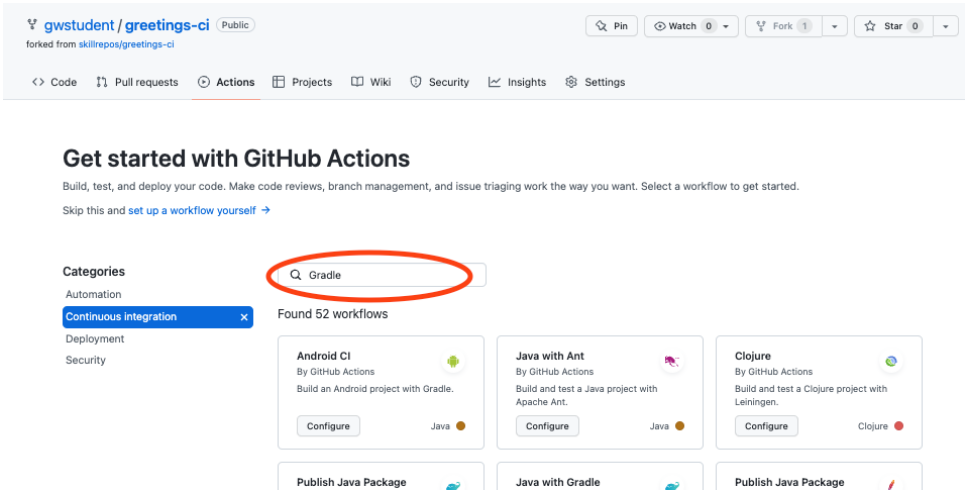
3. We have a simple java source file named *echoMsg.java* in the subdirectory *src/main/java*, a Gradle build file in the root directory named *build.gradle,* and some other supporting files. We could clone this repository and build it manually via running Gradle locally. But let's set this to build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



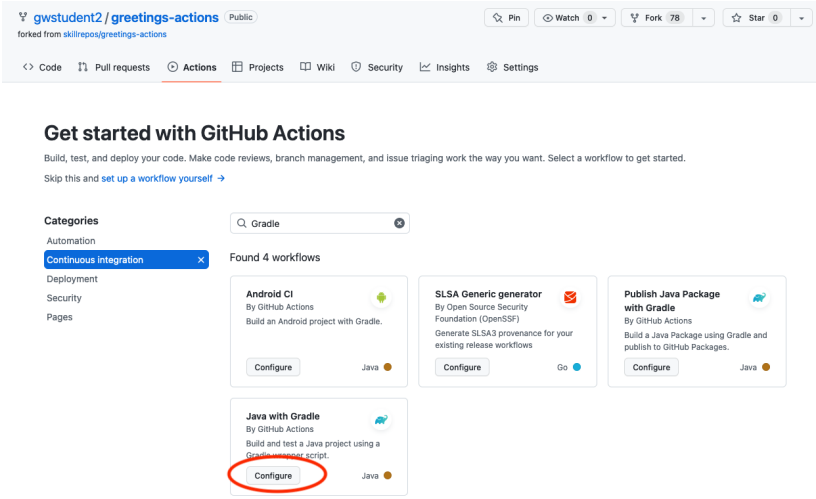
4. This will bring up a page with categories of starter actions that GitHub thinks might work based on the contents of the repository. We'll select a specific CI one. Scroll down to near the bottom of the page under "Browse all categories" and select "Continuous integration".



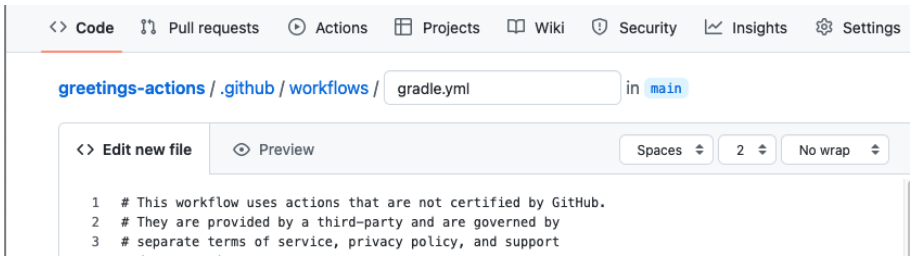
5. In the CI category page, let's search for one that will work with Gradle. Type "Gradle" in the search box and press Enter.



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'll make - changing the name of the file and the name of the workflow. 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.yaml. (You can just backspace over or delete the name and type the new name.)



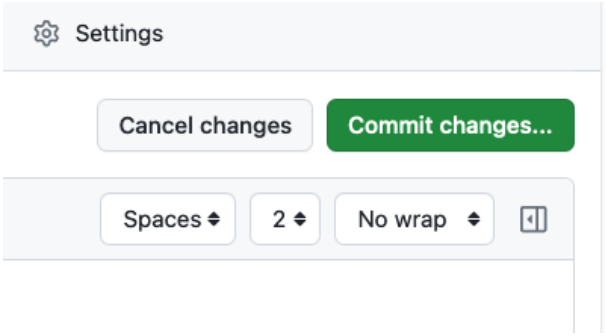
TO

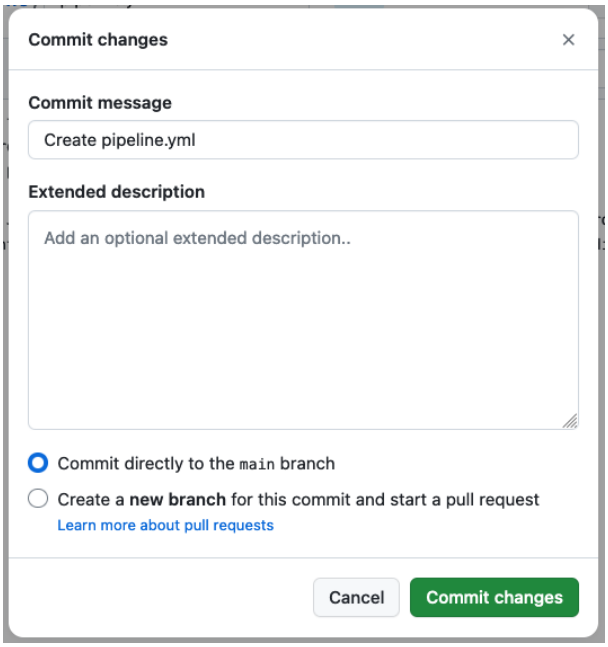
8. Now, edit the name of the workflow - change line 8 from "**name: Java CI with Gradle**" to "**name: Simple Pipe**".

```
5  # This workflow will build a Java project
6  # For more information see: https://docs.
7
8  name: Java CI with Gradle
9
10 on:
11   push:
```

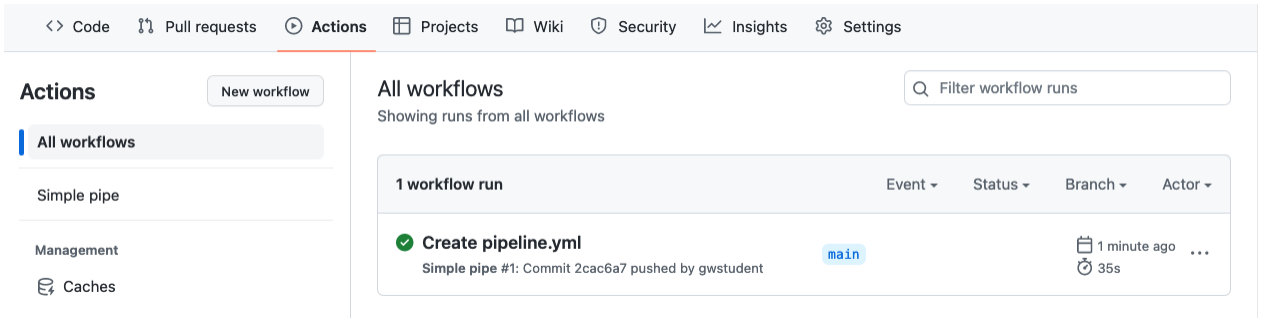
```
5  # This workflow will build a Java project
6  # For more information see: https://docs.
7
8  name: Simple Pipe
9
10 on:
11   push:
```

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.

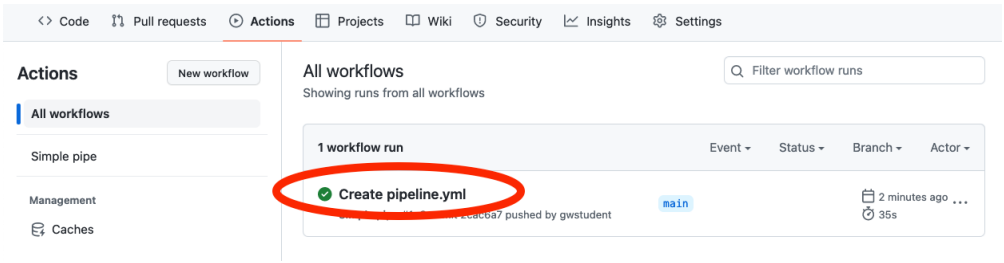




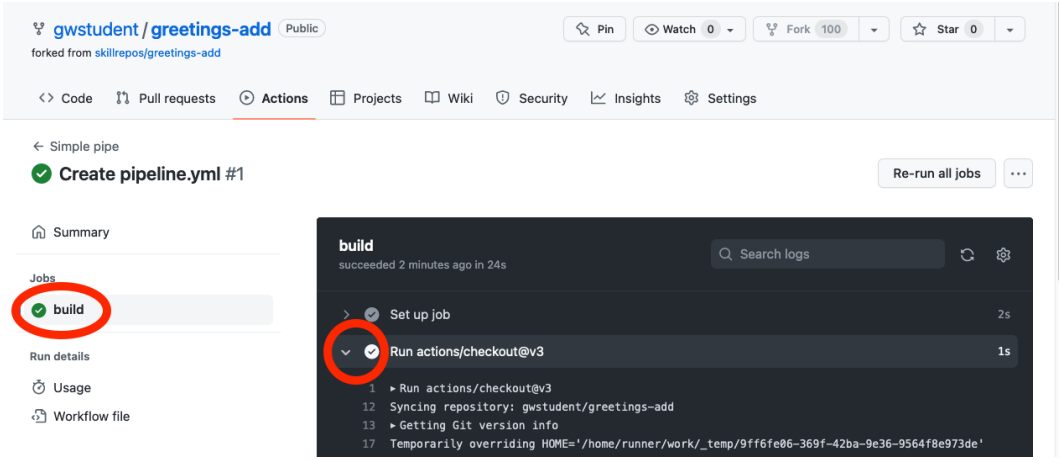
10. Since we’ve committed a new file and this workflow is now in place, the “on: push:” event is triggered, and the CI automation kicks in. Click on the Actions menu again to see the automated processing happening.



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 (next to the green check) for the run to get to the details for that 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 – Learning more about Actions

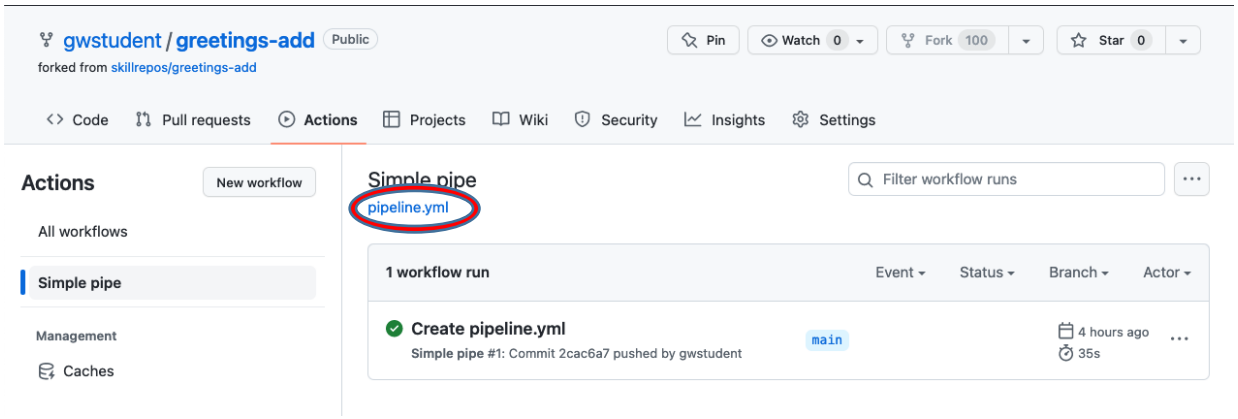
Purpose: In this lab, we'll see how to find and use additional actions as well as persist artifacts.

1. We're going to explore one way in GitHub to update a workflow and add additional actions into it. Start out by opening up the workflow file pipeline.yml. There are multiple ways to get to it but let's open it via the Actions screen.

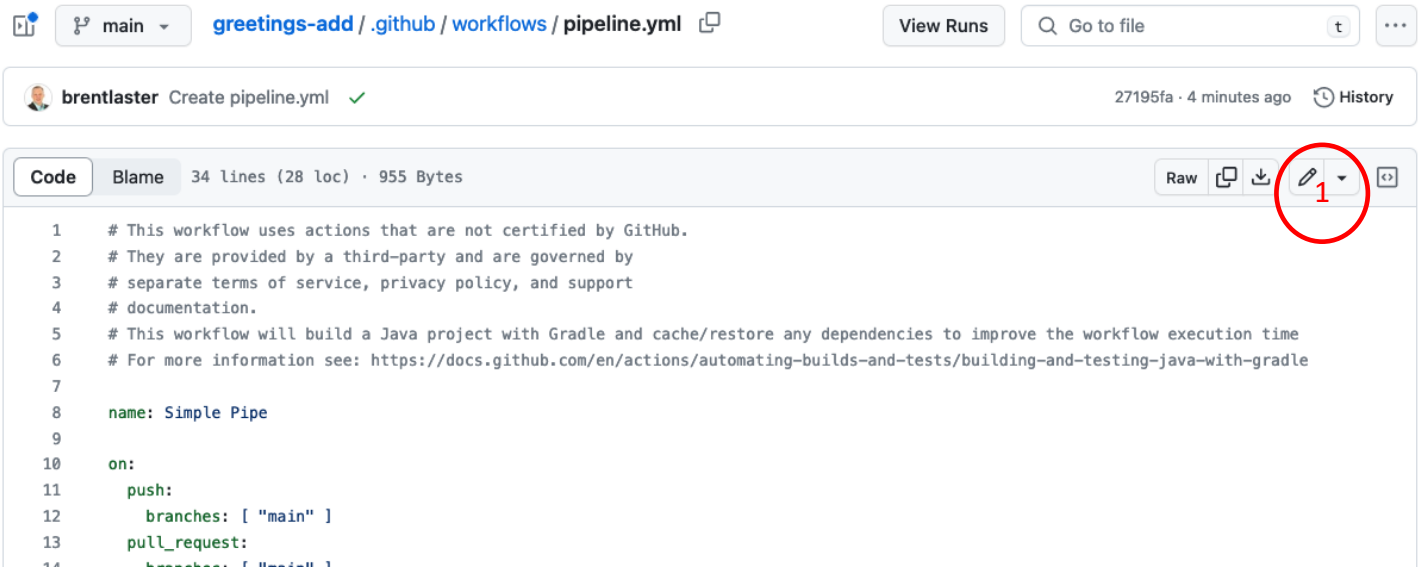
   In your GitHub repository, click the Actions button at the top if not already on the Actions screen.

   Under "All workflows", select the "Simple Pipe" workflow.

   After that, select the "pipeline.yml" link near the middle top.



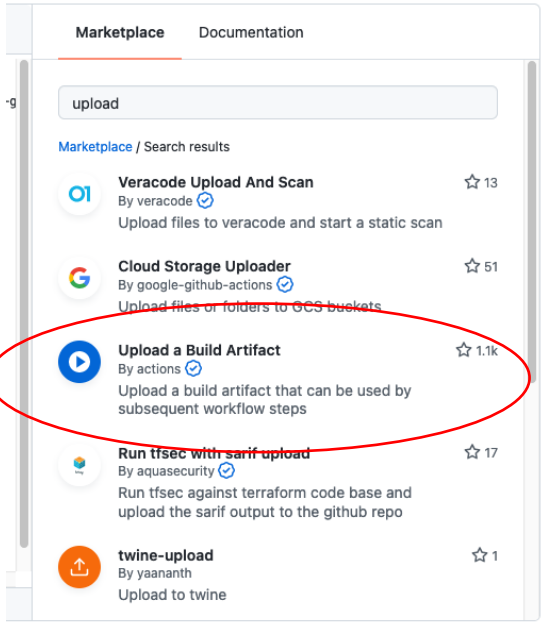
2. Once the file opens up, click on the pencil icon (1) in the top right to edit it.

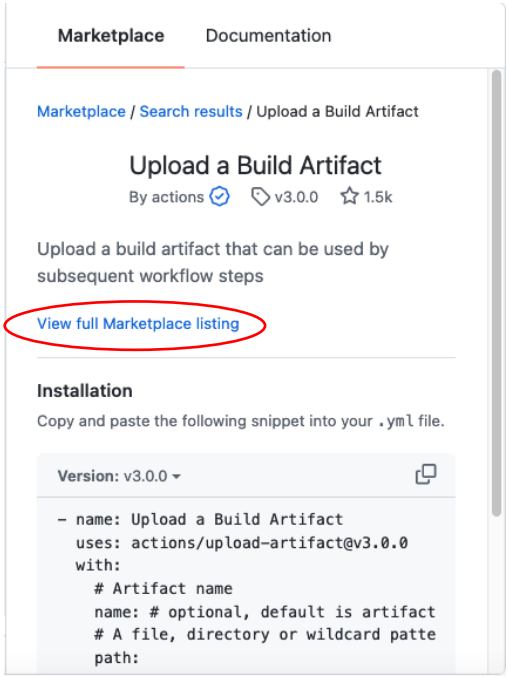


3. You'll now see the file open up in the editor, but also to the right, you should see a new pane with references to the GitHub Actions Marketplace and Documentation. We're going to add a job to our workflow to upload an artifact. Let's find actions related to uploading.

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

   Next, click on the "Upload a Build Artifact" item. 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".

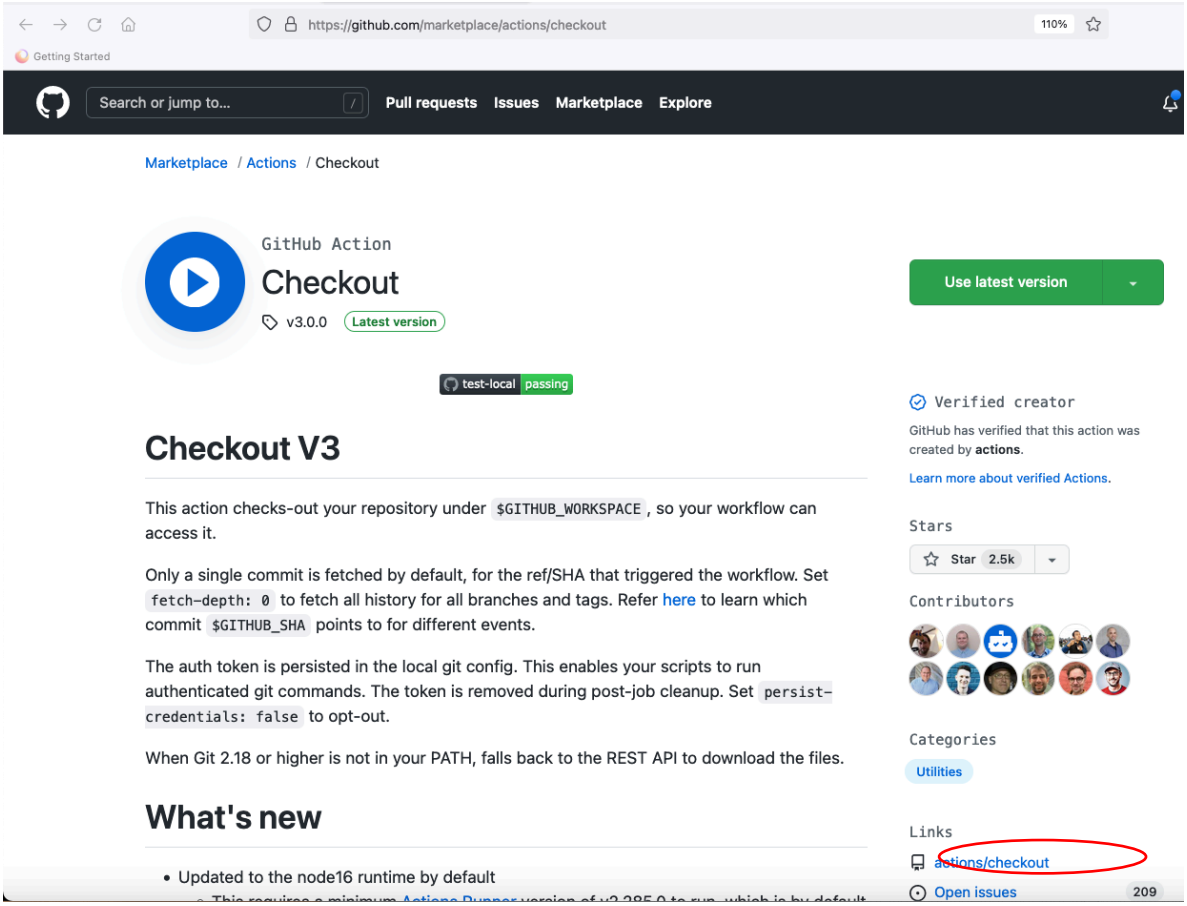




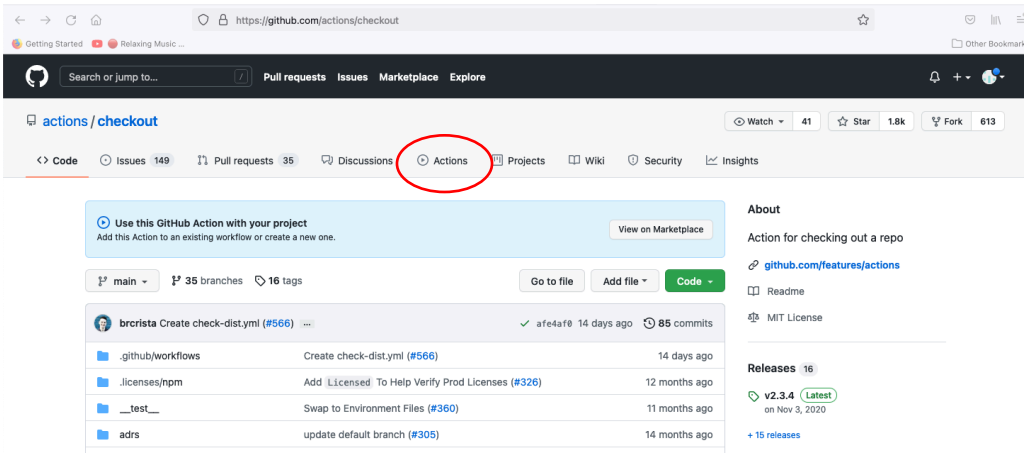
4. This should open up the full GitHub Actions Marketplace listing for this action. Notice the URL at the top - https://github.com/marketplace/actions/upload-a-build-artifact. You can use this same relative URL to see

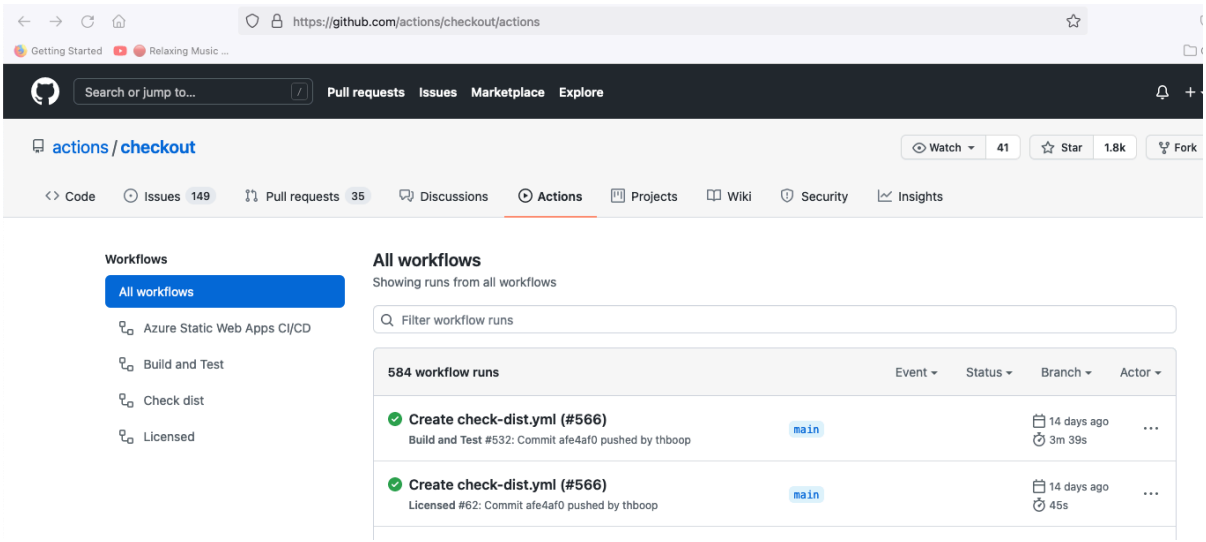
other actions that are in the marketplace. For example, let's look at the checkout one we're already using. Go to https://github.com/marketplace/actions/checkout

Then click on the "actions/checkout" link under "Links" in the lower right.



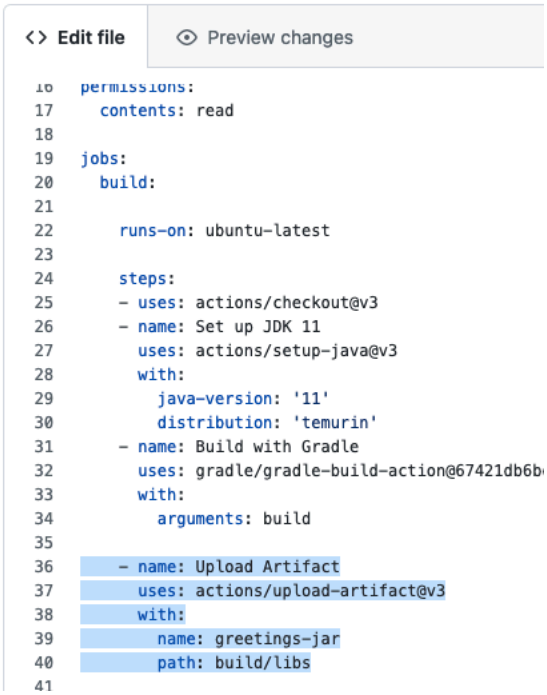
5. This will put you on the screen for the source code for this GitHub Action. Notice there is also an Actions button here. GitHub Actions use workflows that can use other GitHub Actions. Click on the Actions button to see the workflows that are in use/available.



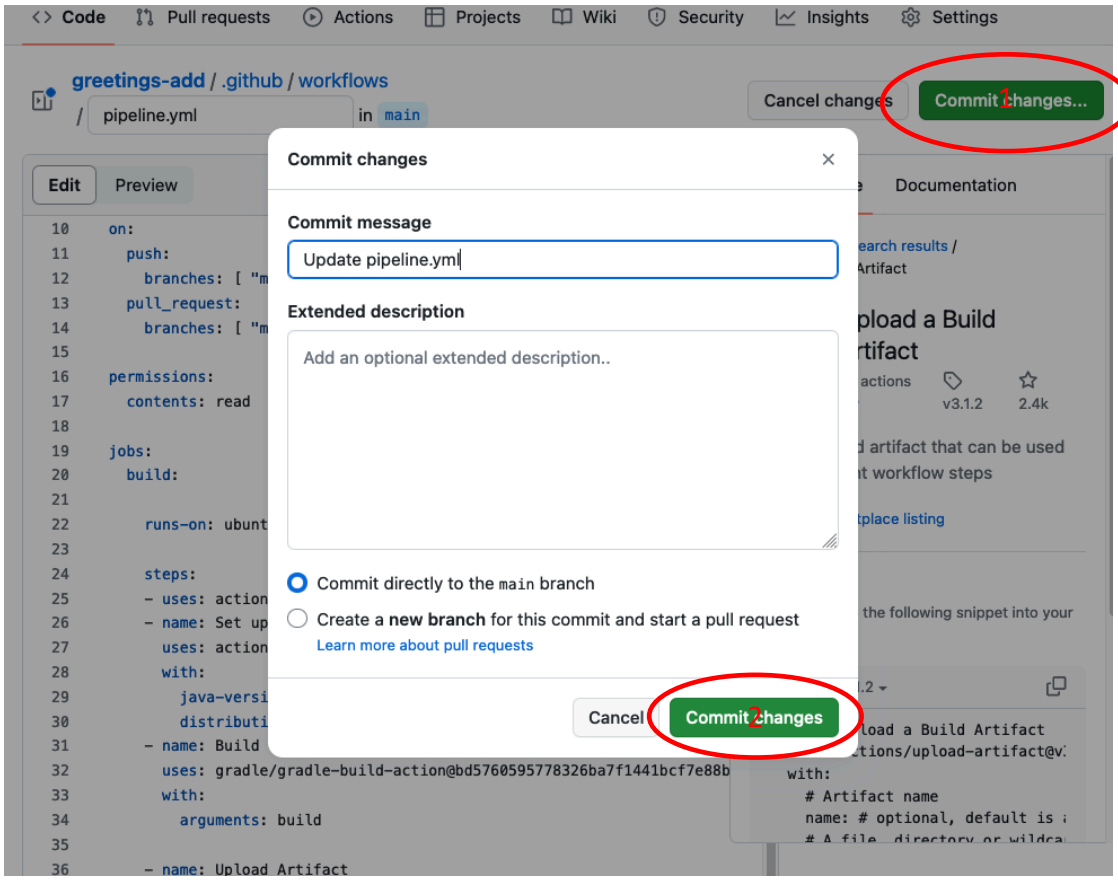


6. Switch back to the browser tab where you are editing the workflow for greetings-actions. Update the build job to include a new step to use the "upload-artifact" action to upload the jar the build job creates. To do this, add the following lines inline with the build job steps. Pay attention to the indenting. If you see red wavy lines under your code, that likely means the indenting is off. See the screenshot (lines 36-40) for how this should look afterwards. (Your line numbers may be different.)

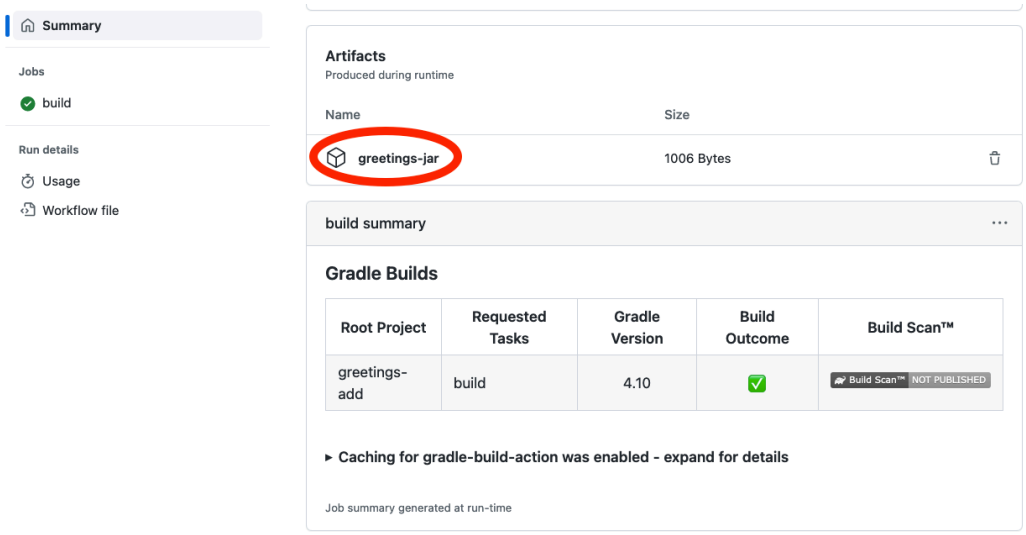
```
- name: Upload Artifact
  uses: actions/upload-artifact@v3
  with:
    name: greetings-jar
    path: build/libs
```



7. Click on the green "Commit changes" button in the upper right. In the dialog that comes up, add a different commit message if you want, then click the green "Commit changes" button to make the commit.



8. 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 "Update pipeline.yml" (or whatever your commit message was). On the next screen, in addition to the graph, there will be a new section called "Artifacts" near the bottom. You can download the artifact from there. Click on the name of the artifact to try this.



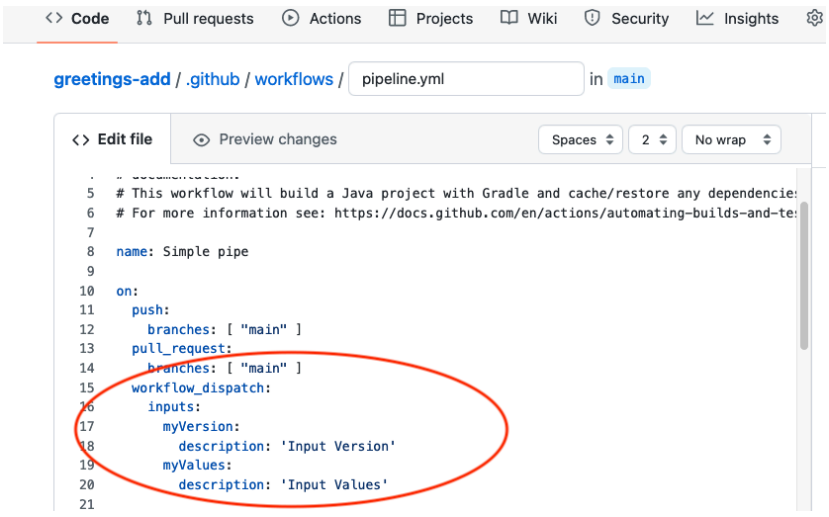
* END OF LAB *

Lab 3 – Alternative ways to invoke workflows

Purpose: In this lab, we'll see how to add a different kind of event trigger that allows us to invoke the workflow manually

1. Let's make a change to make it easier to run our workflow manually to try things out, start runs, etc. We are going to add two input values - one for the version of the artifact we want to create and use and one for input values to pass to a test. Edit the pipeline.yaml file again. 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:
    myVersion:
       description: 'Input Version'
     myValues:
       description: 'Input Values'
```



2. Now let's add a step to our build job to get the timestamp to use to version the artifact. Add the step below AFTER the build step and BEFORE the upload step.

```
- name: Set timestamp
run: echo TDS=$(date +'%Y-%m-%dT%H-%M-%S') >> $GITHUB_ENV
```

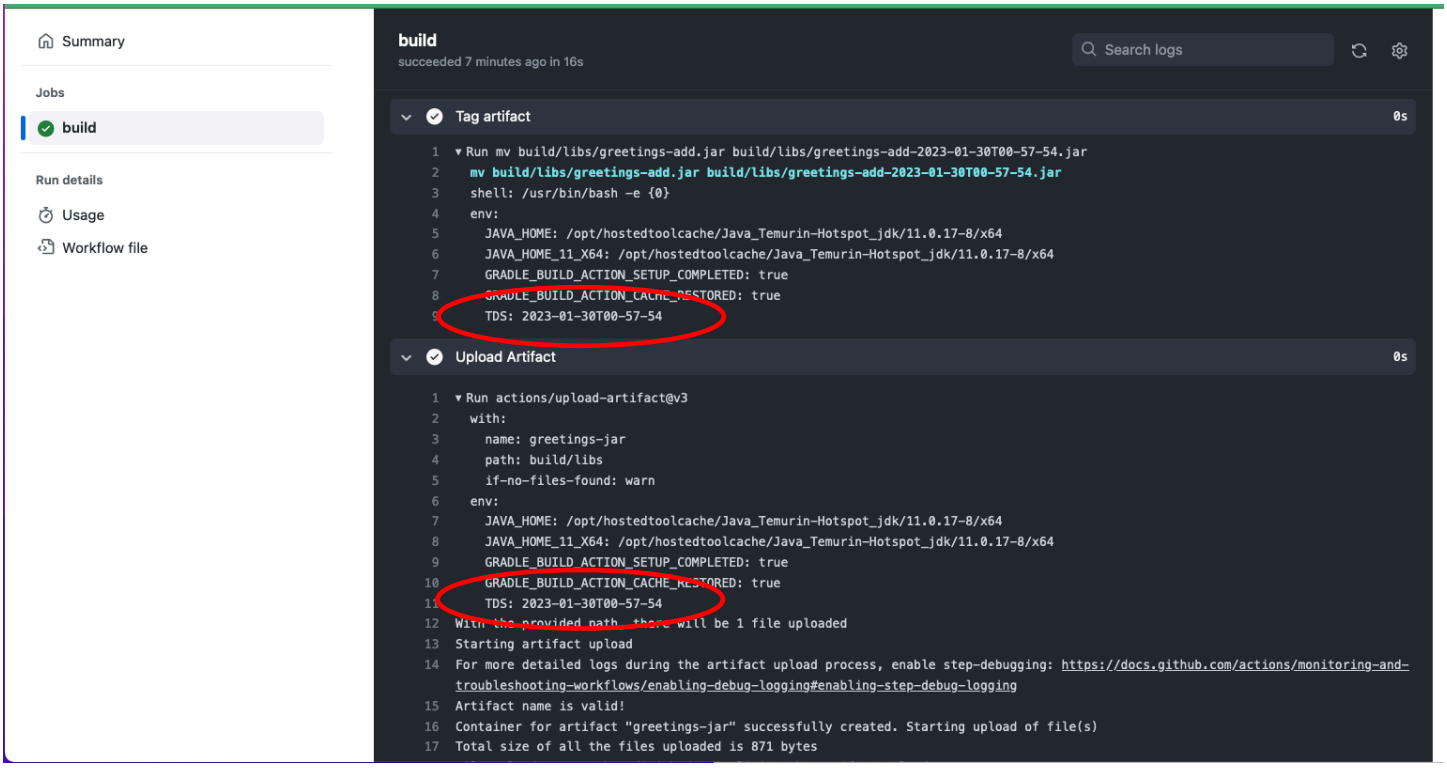
3. Next add another step to "tag" the artifact with the input version (if there is one) and also the timestamp. Add this step right after the previous one and BEFORE the upload step.

```
- name: Tag artifact
  run: mv build/libs/greetings-add.jar build/libs/greetings-add-${{
github.event.inputs.myVersion }}${{ env.TDS }}.jar
```

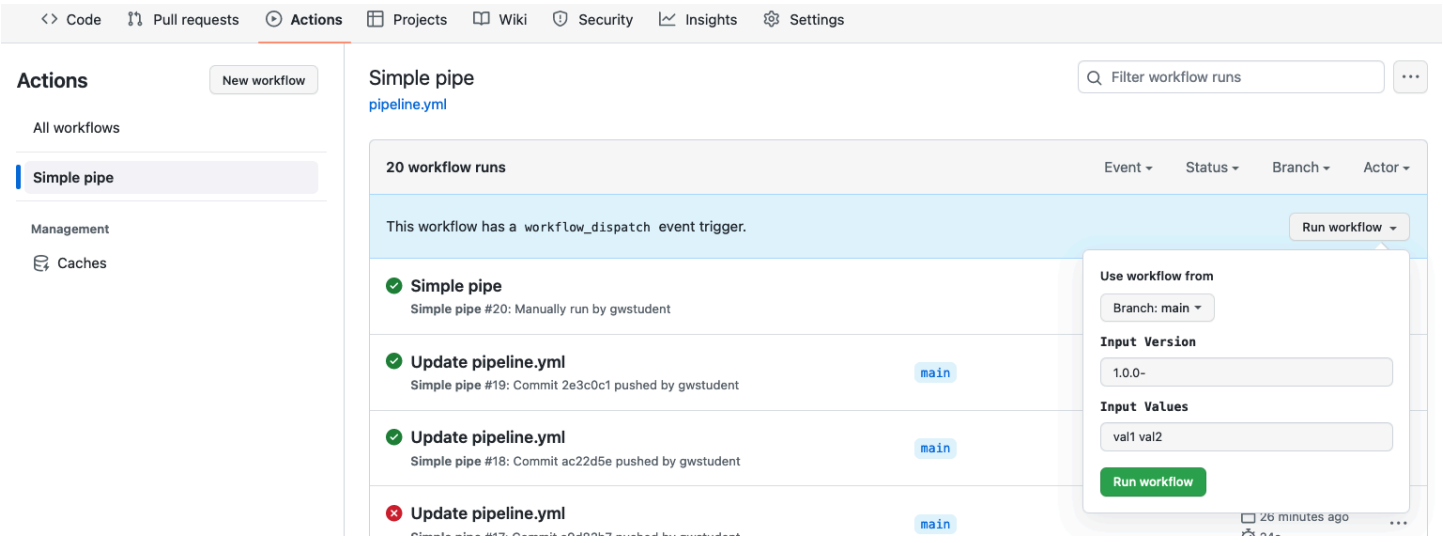
The figure below shows the steps added in the workflow.

```
40      - name: Build with Gradle
41        uses: gradle/gradle-build-action@67421db6bd0bf253fb4bd25b31ebb98943c375e1
42        with:
43          arguments: build
44
45      - name: Set timestamp
46        run: echo TDS=$(date +'%Y-%m-%dT%H-%M-%S') >> $GITHUB_ENV
47
48      - name: Tag artifact
49        run: mv build/libs/greetings-add.jar build/libs/greetings-add-${{ github.event.inputs.myVersion }}${{ env.
50
51      - name: Upload Artifact
52        uses: actions/upload-artifact@v3
53        with:
54          name: greetings-jar
55          path: build/libs
56
```

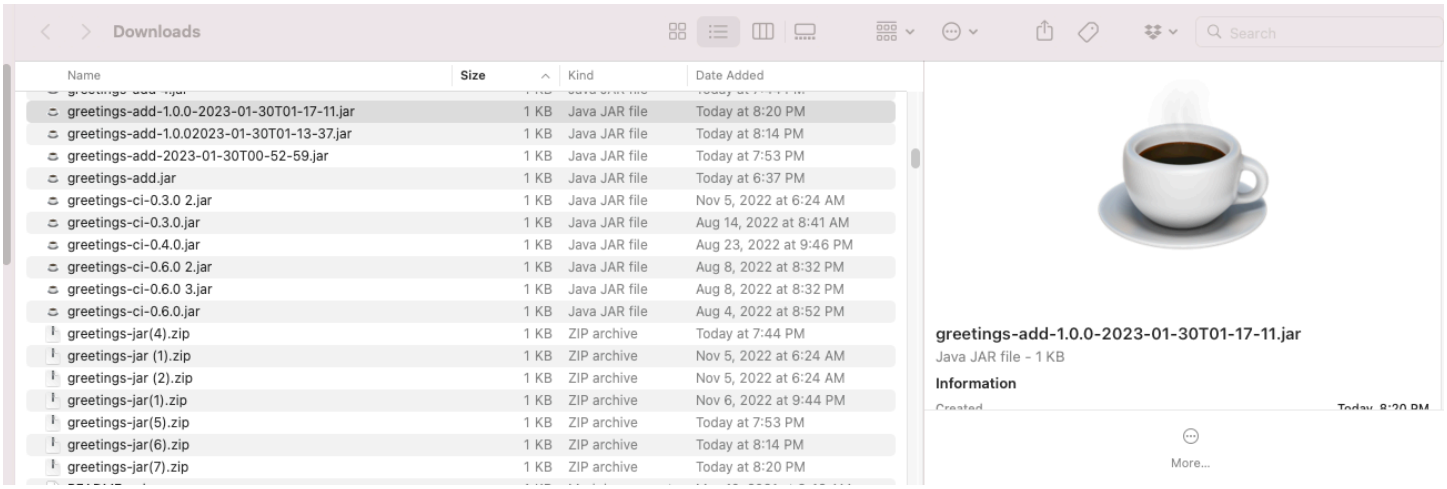
4. Go ahead and commit the changes. After this runs, you can look at the logs by clicking on the Actions menu, then in the workflow runs list, click on the commit message for the particular run and then on the job itself. On the right-hand side, click on the downward pointing arrows next to the "Tag artifact" and "Upload Artifact" and expand them to see the individual steps. Notice the *TDS* variable we defined as part of the environment.



5. The workflow_dispatch code we added to the event trigger sections created a way to manually run the workflow and also pass in values for the parameters we defined. To see how to run the workflow manually, click on the main Actions menu (if not already there), then select the "Simple pipe" workflow on the lefthand side. At that point you should see a "Run workflow" button on the far right at the top of the runs list. Click on that button and enter any numeric value you want for the Input version and then any Input values you want to supply. Then click on "Run workflow".



6. After this run, you can select the run from the runs list, scroll down and find the greetings.jar artifact and click on it to download it. Once you have it downloaded, you can uncompress the artifact and you should see a jar file with the version you entered for "Input Version" and the time-date stamp.

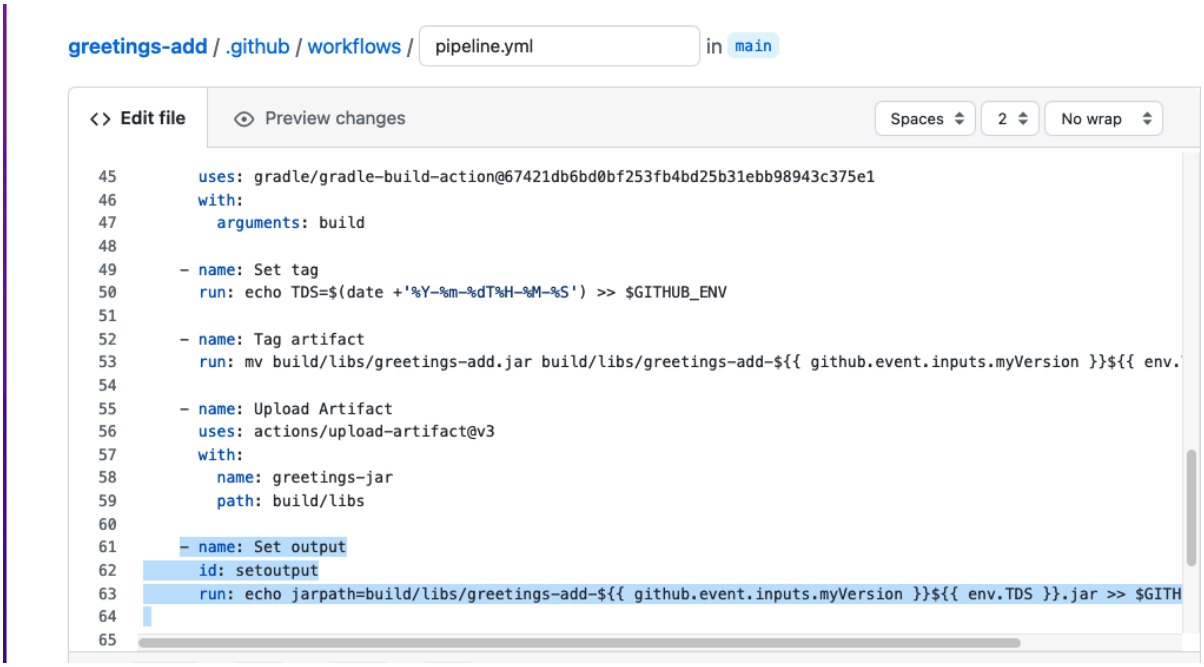


Lab 4 – Sharing output between jobs

Purpose: In this lab, we'll see how to capture output from one job and share it with another one

1. Let's add one more piece to this job so we can have the path of the jar file available for other jobs. To do this, edit the workflow file and add a step at the end of the job to set the output value.

```
- name: Set output
    id: setoutput
    run: echo jarpath=build/libs/greetings-add-${{ github.event.inputs.myVersion
}}${{ env.TDS }}.jar >> $GITHUB_OUTPUT
```



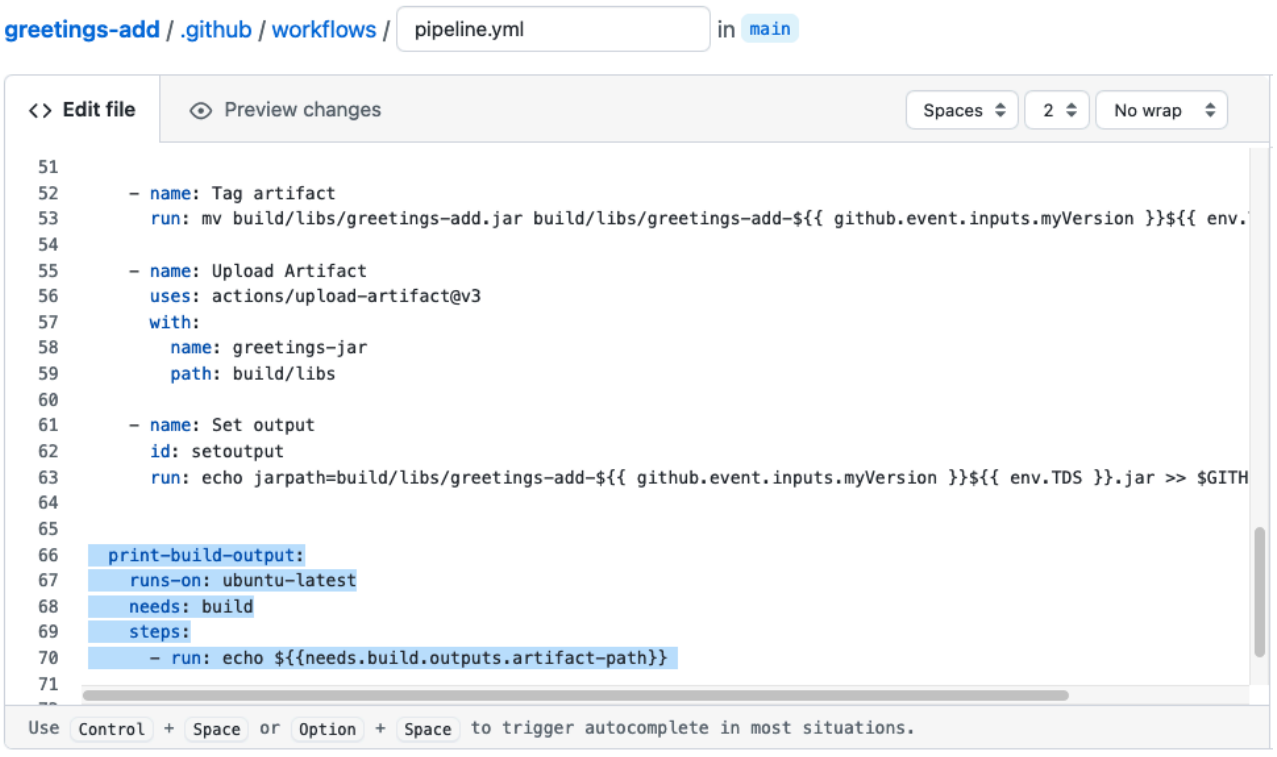
2. Now we need to add an "outputs" section BETWEEN the "runs-on:" and the "steps:" section near the top of the "build" job. This will map the variable "artifact-path" to the outputs of the previous step.

```
# Map a step output to a job output
outputs:
  artifact-path: ${{ steps.setoutput.outputs.jarpath }}
```

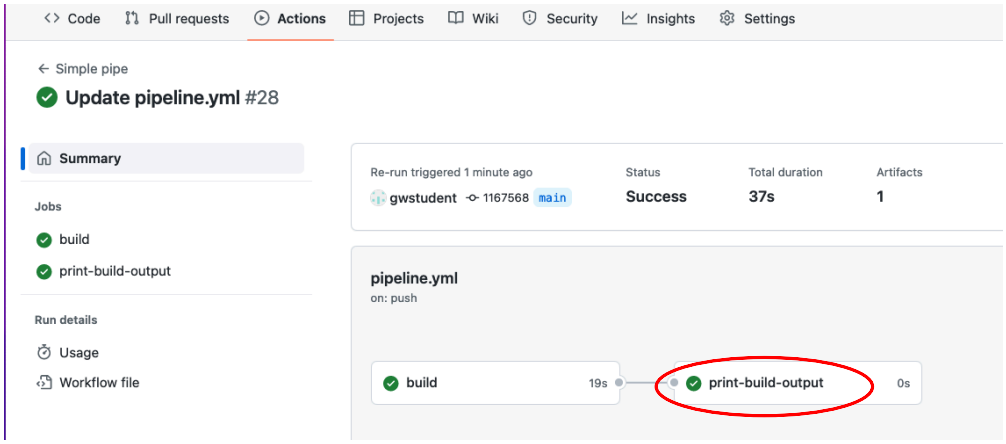
```
26
27  jobs:
28    build:
29
30      runs-on: ubuntu-latest
31
32      # Map a step output to a job output
33      outputs:
34        artifact-path: ${{ steps.setoutput.outputs.jarpath }}
35
36      steps:
37      - uses: actions/checkout@v3
38      - name: Set up JDK 11
39        uses: actions/setup-java@v3
40
```

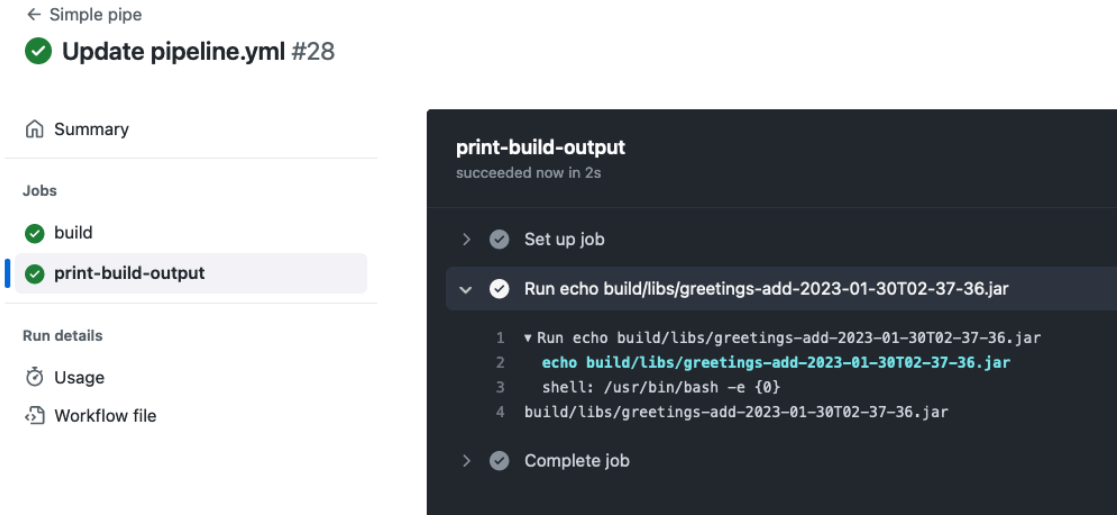
3. In order to verify that we can see the output from the build job, add a new, second job in the workflow file. Copy and paste the simple job below that echoes out the output value from the build job. Note that "print-build-output" should line up with the "build" title of the first job.

```
print-build-output:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - run: echo ${{needs.build.outputs.artifact-path}}
```



4. Commit the changes. After the workflow runs, you should see two jobs in the graph for the workflow run - one for "build" and one for "print-build-output". Click on the "print-build-output" one and you can see from the logs that it was able to print the output value created from the "build" job.

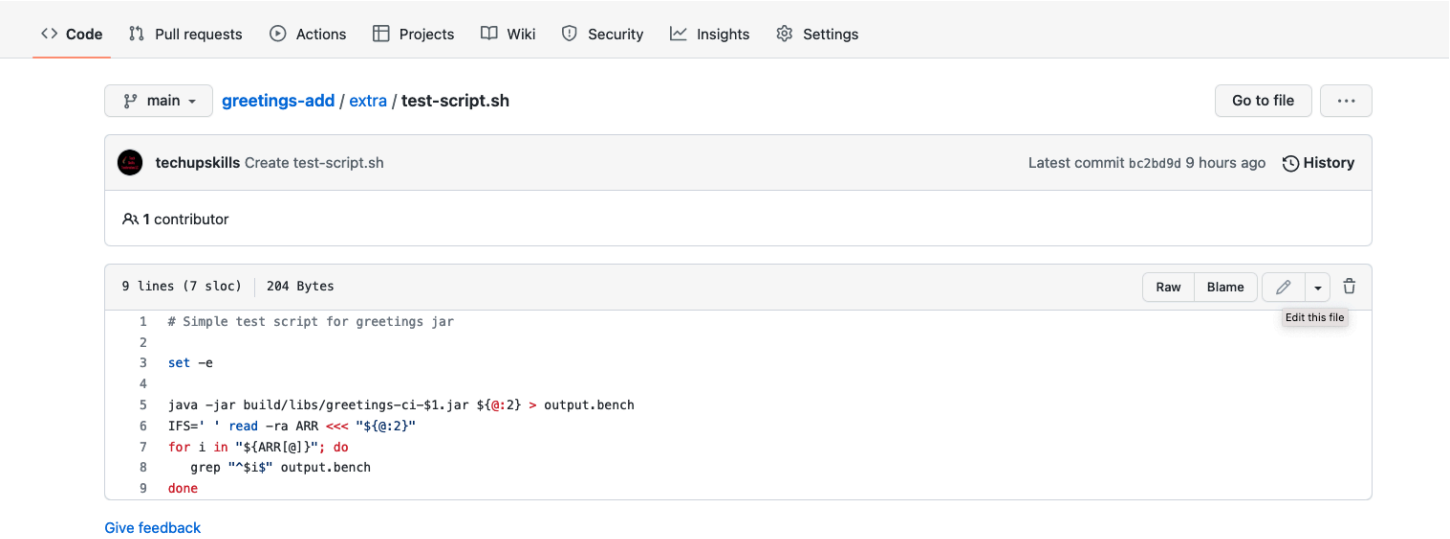




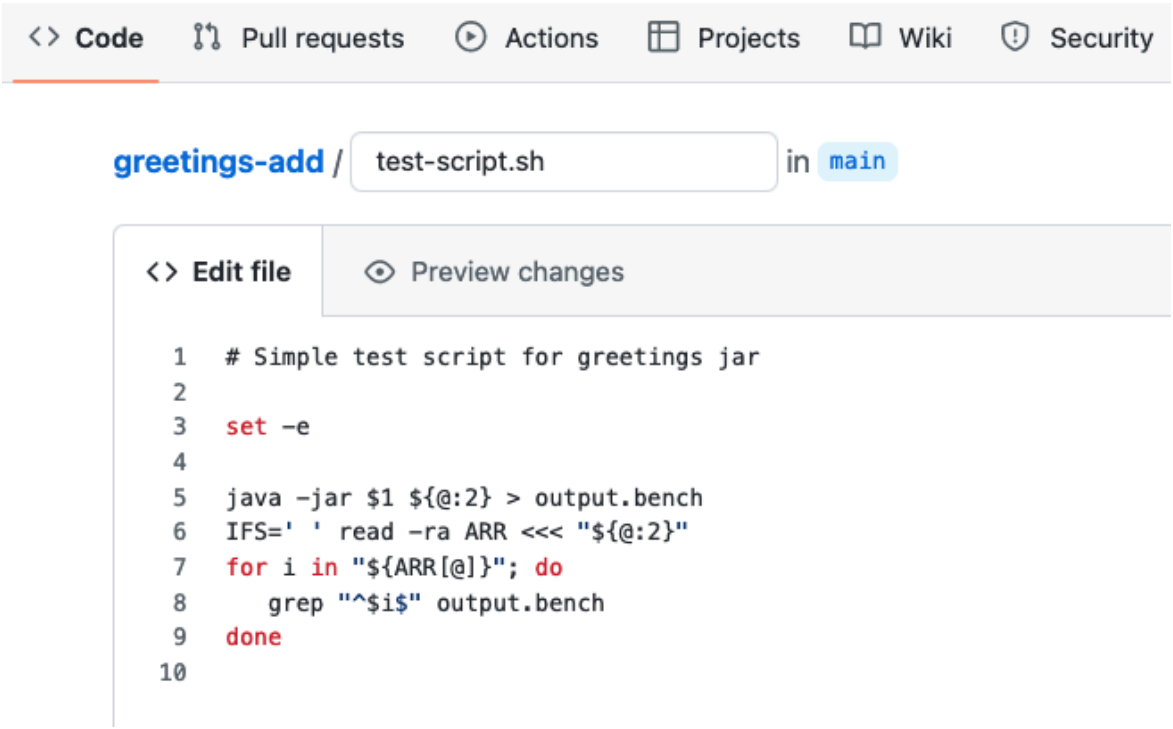
Lab 5: Adding in a test case

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

1. First, let's create a script to test our code. The code for the test script we'll use is already in the "extra/test-script.sh" file. Open up that file (in the "extra" subdirectory) and click the pencil icon to edit it.



2. In the editor, all you need to do is change the path of the file. Click in the text entry area for "test-script.sh" and backspace over the "extra" path so that the file is directly in the "greetings-add" directory (root repo directory).



3. This script takes the path to 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.

4. Go ahead and commit this file into the repository for the path change.

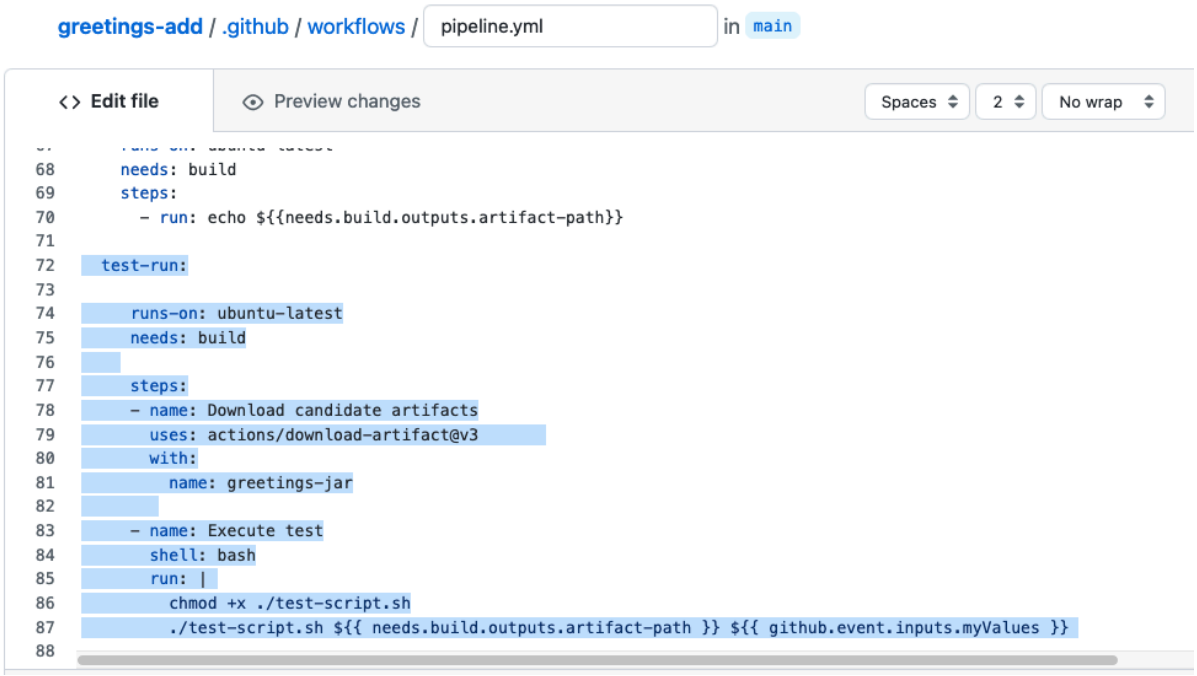
5. Now let's add a second job to our workflow (in pipeline.yml) to do a simple "test". As you've done before, edit the *pipeline.yml* file.

6. Add the job definition for a job called "test-run" that runs on ubuntu-latest. You can copy and paste this code from **extra/test-run.txt** or grab it from the next page.

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.)

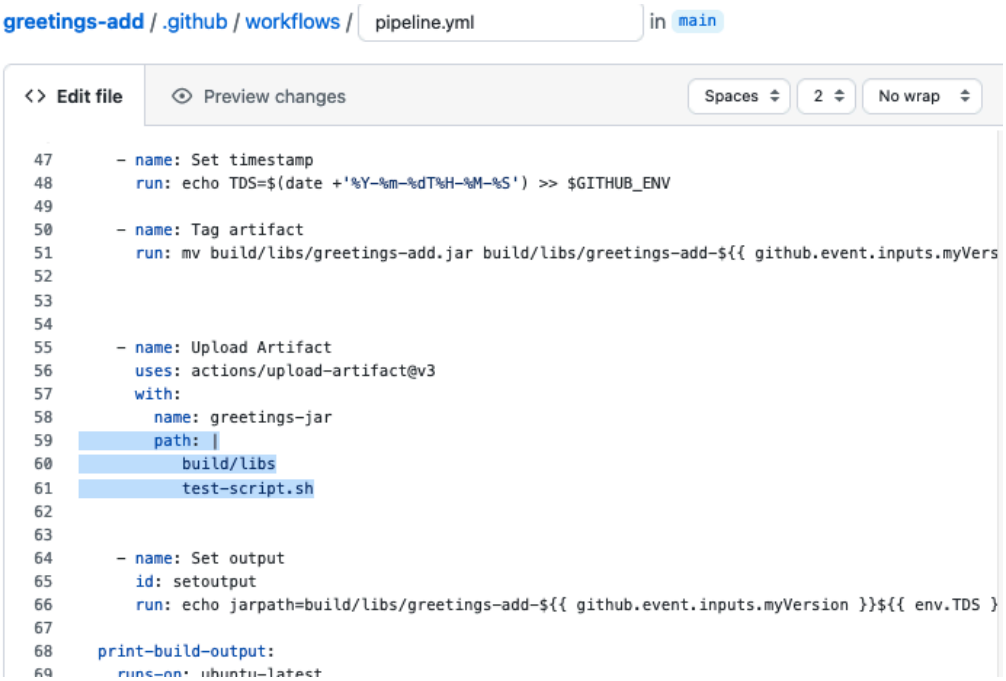
```
test-run:
  runs-on: ubuntu-latest
  needs: build

  steps:
  - name: Download candidate artifacts
    uses: actions/download-artifact@v3
    with:
      name: greetings-jar

  - name: Execute test
    shell: bash
    run: |
      chmod +x ./test-script.sh
      ./test-script.sh ${{ needs.build.outputs.artifact-path }} ${{ github.event.inputs.myValues }}
```



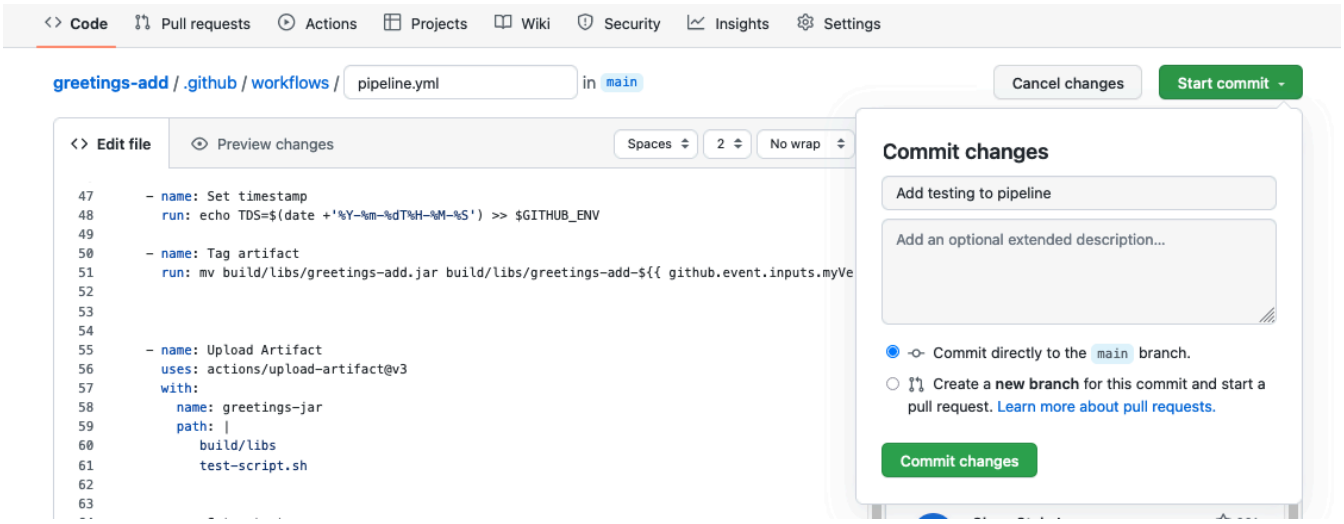
7. Since each job executes on a separate runner system, we need to make sure our new test script is available on the runner that will be executing the tests. For simplicity, we can just add it to the list of items that are included

in the uploading of artifacts. Scroll back up, find the "Upload Artifact" step in the "build" job. Modify the **path** section of the "Upload Artifact" step to change from "path: build/libs" to look like below.

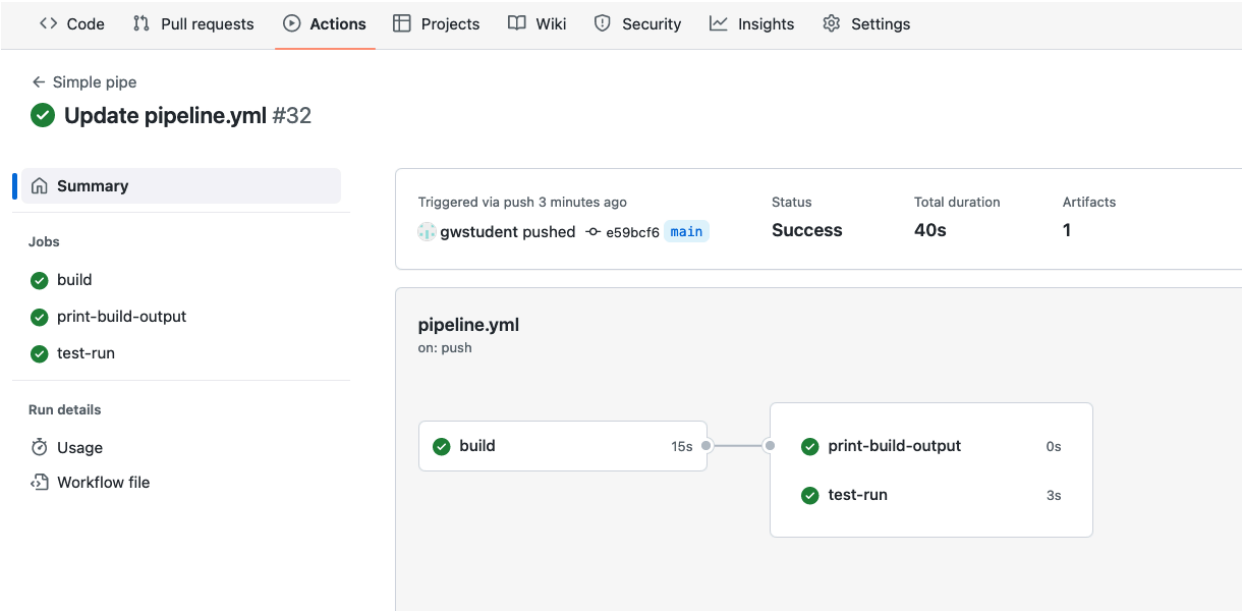
```
path: |
  build/libs
  test-script.sh
```



8. Now, you can just commit the pipeline changes with a simple message like "Add testing to pipeline".



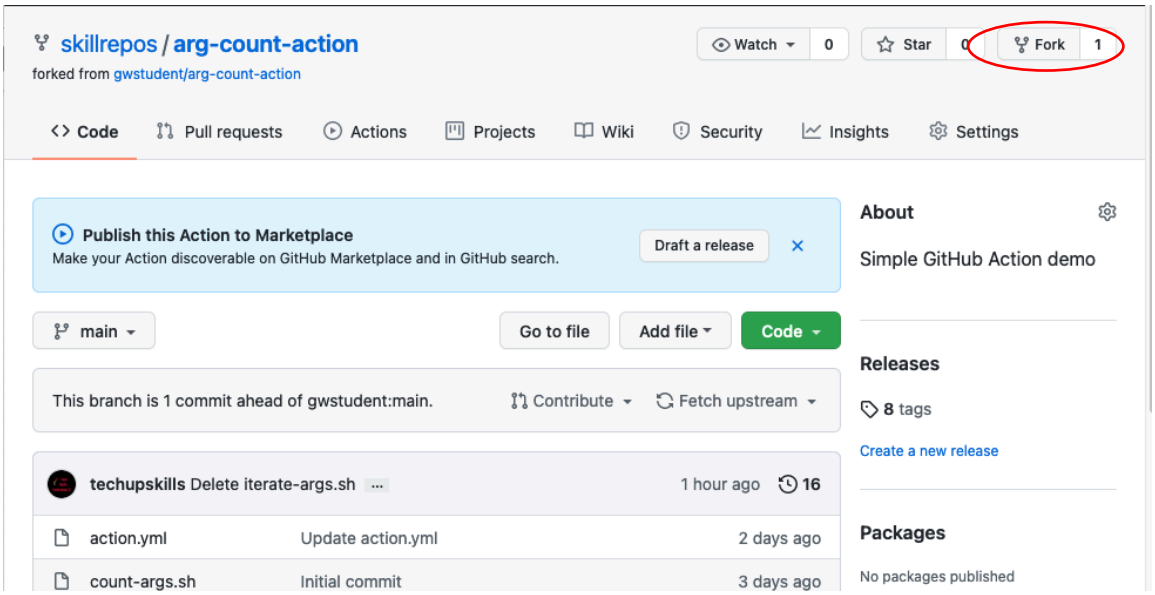
9. Afterwards, you should see a new run of the action showing multiple jobs in the action run detail. Notice that we can select and drill into each job separately.



Lab 6: Adding your own action

Purpose: in this lab, we'll see how to create and use a custom GitHub Action.

1. First, we'll fork the repo for a simple action that displays a count of the arguments passed into a function. Go to https://github.com/skillrepos/arg-count-action and then Fork that repository into your own GitHub space. (You can just accept the default selections on the page.)

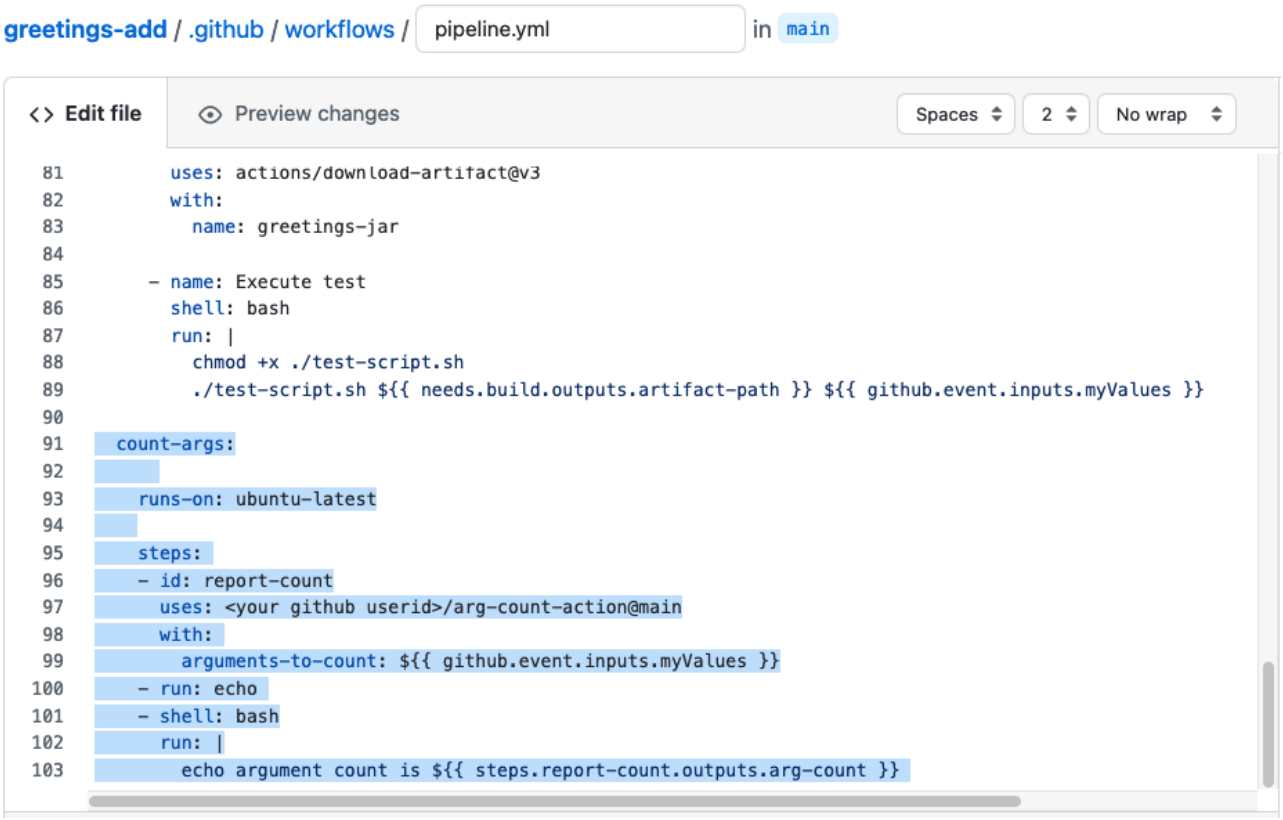


2. In your fork of the repository, look at the files here. We have a one-line shell script (for illustration) to return the count of the arguments - "count-args.sh." And we have the primary logic for the action in the "action.yml" file.

Take a look at the action.yml file and see if you can understand what its doing. The syntax is similar to what we've seen in our workflow up to this point.

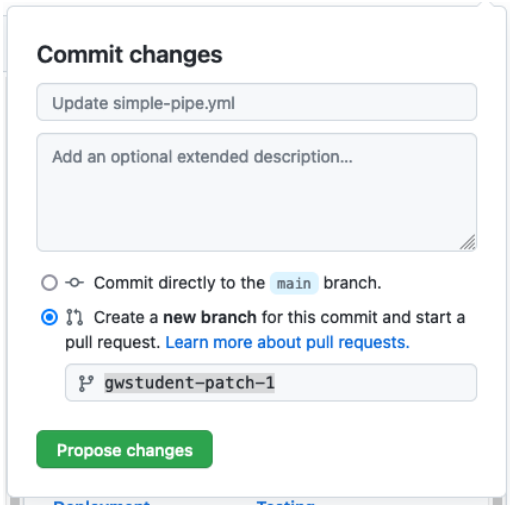
3. Switch back to the file for your original workflow (go back to the greetings-add project and edit the *pipeline.yaml* file in *.github/workflows*. Let's add the code to use this custom action to report the number of arguments passed in. Edit the file and add the code shown below (again indenting the first line to align with the other job names). (For convenience, this code is also in "greetings-add/extra/count-args.txt".) **For now, just leave the text exactly as is so we can see what errors look like.**

```
count-args:

  runs-on: ubuntu-latest

  steps:
  - id: report-count
    uses: <your github userid>/arg-count-action@main
    with:
      arguments-to-count: ${{ github.event.inputs.myValues }}
  - run: echo
  - shell: bash
    run: |
      echo argument count is ${{ steps.report-count.outputs.arg-count }}
```

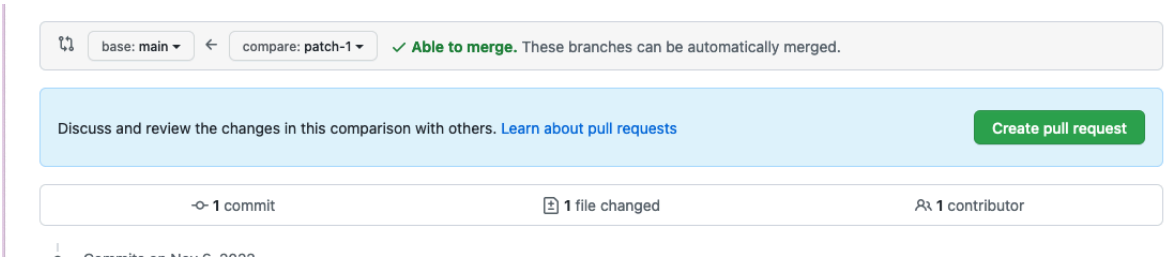


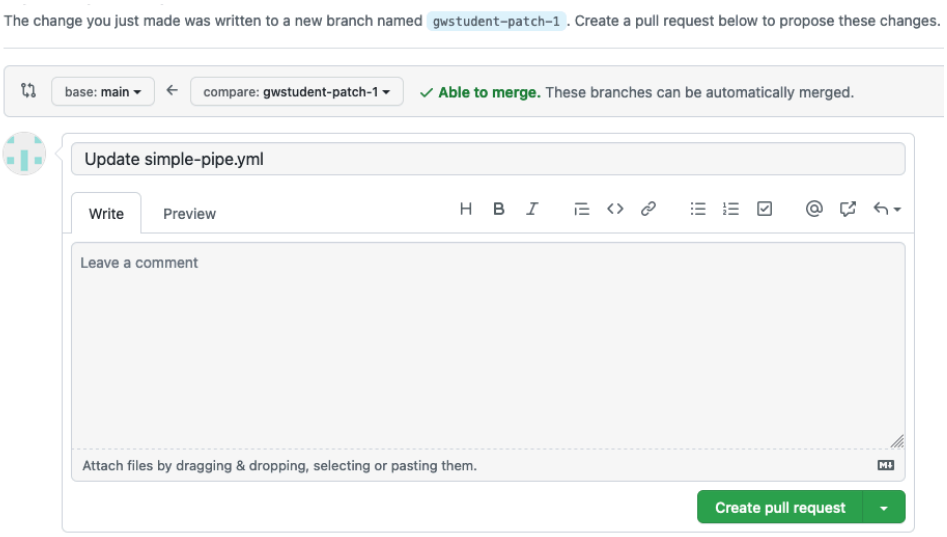
In this case, we call our custom action (<your github repo/arg-count-action>), using the latest from the main branch.

4. Let's use a pull request to merge this change. Click on the green "Commit changes…" button, but in the "Commit changes" dialog, click on the bottom option to "Create a new branch for this commit and start a pull request." Change the proposed branch name if you want and then click on "Propose changes".

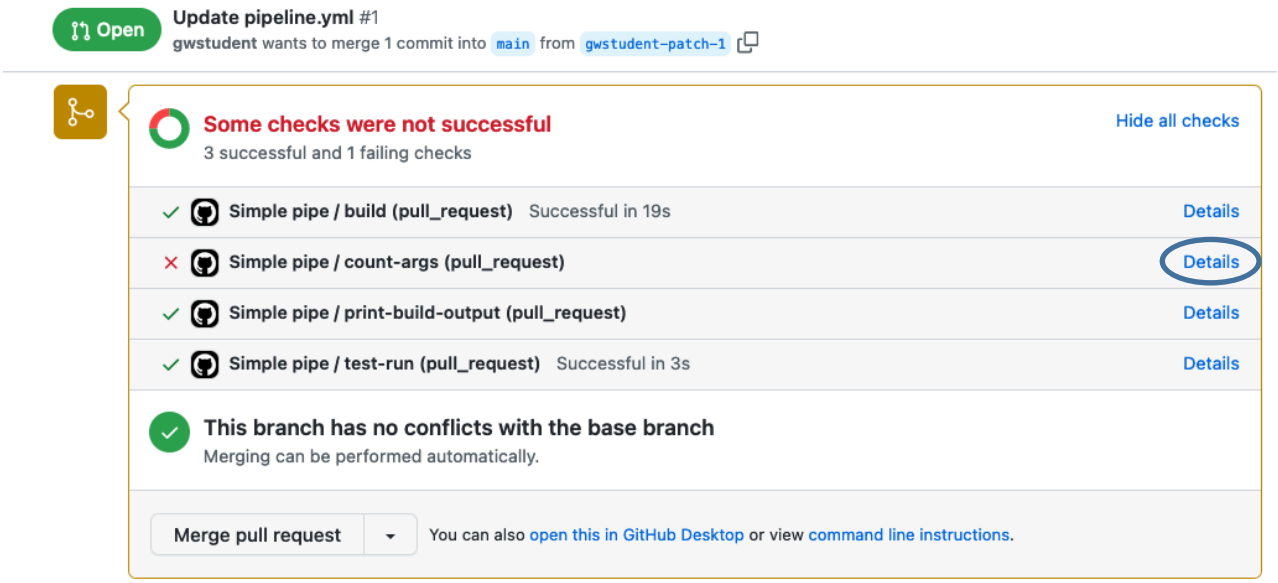


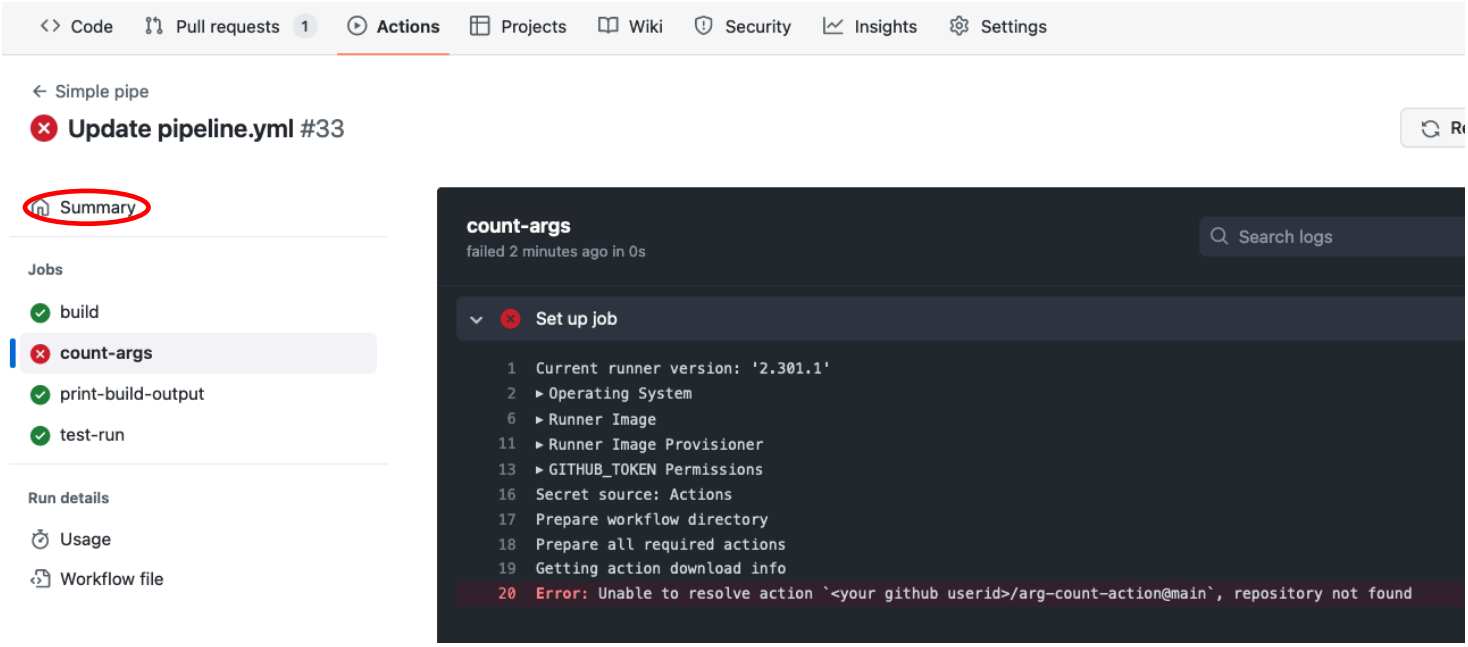
5. In the next screen, click on the "Create pull request" button. In the following screen, update the comment if you want and then click on the "Create pull request" button. You'll then see it run through the jobs in our workflow as prechecks for merging.



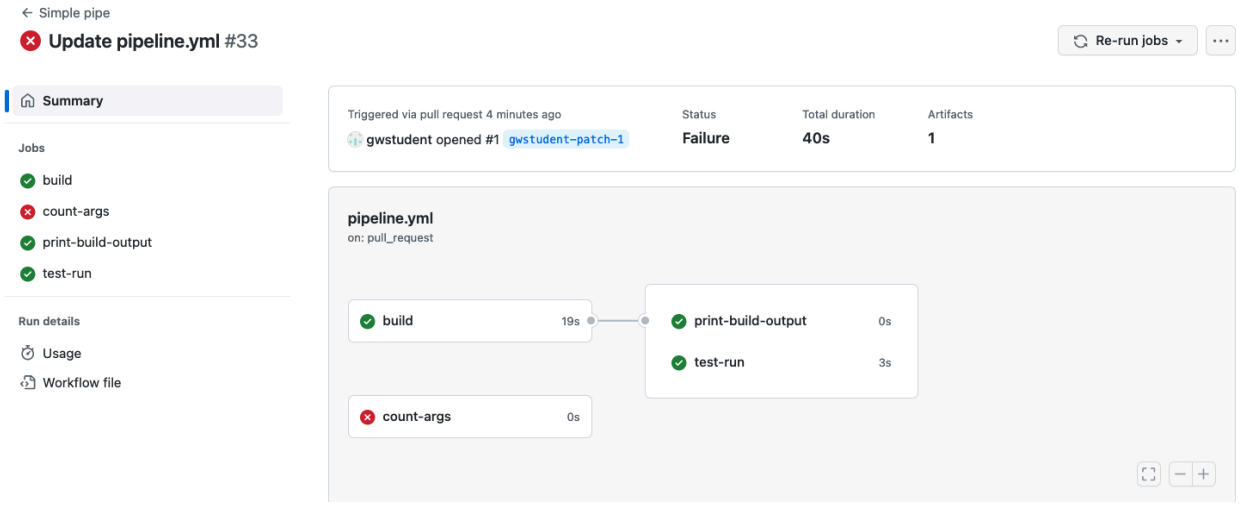


6. When the checks are done running, you'll see one with a failure. Click on the link for "Details" on the right of the line with the failure to see the logs that are available. You can then see the error at the bottom of the log.

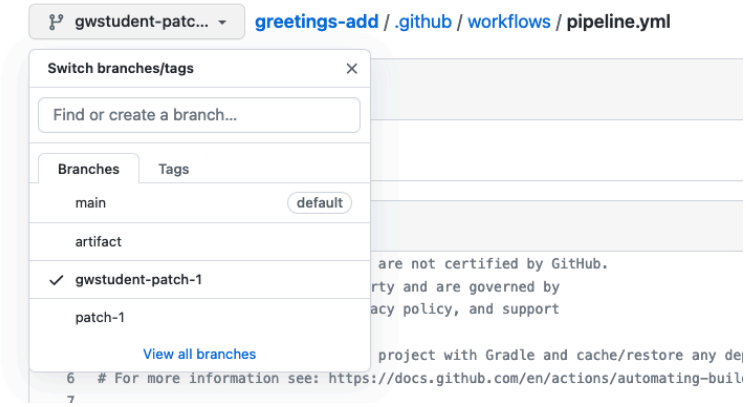




7. In the left column, click on the "Summary" link. This will take you back to the main graph page where you can also see the error.



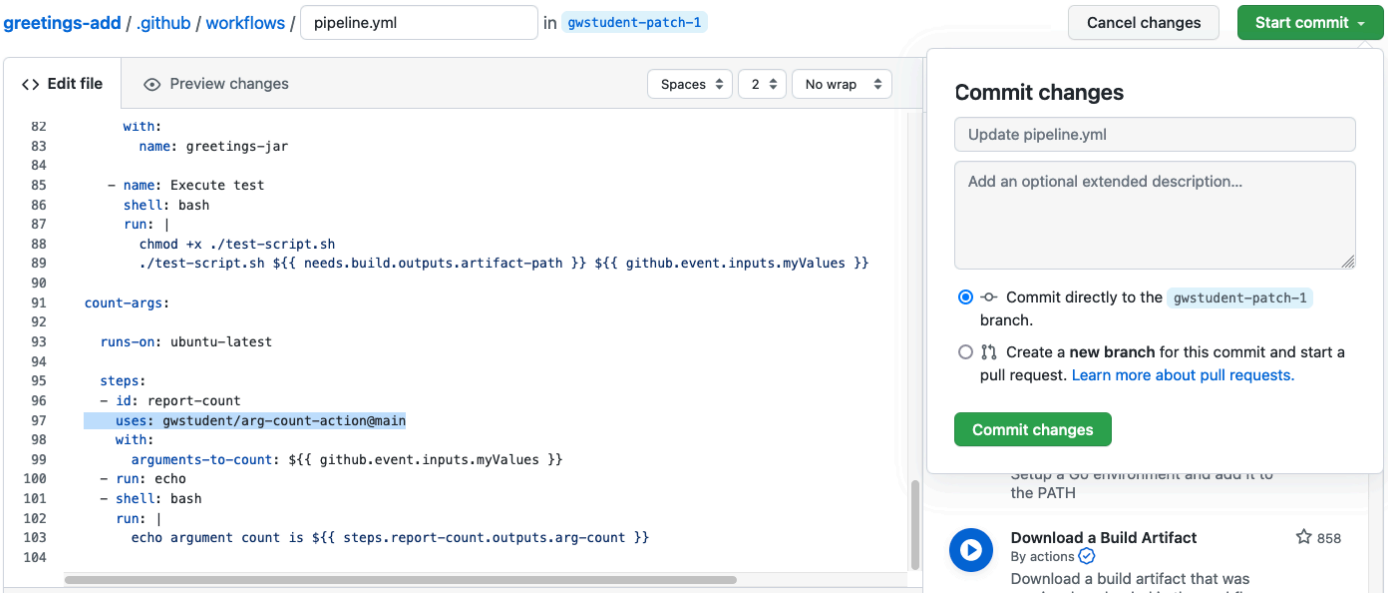
8. So, before we can merge the PR, we need to fix the code. Go back to the "Actions" tab at the top, select the "Simple Pipe" workflow on the left, and then select the **pipeline.yaml** file (link above the list of workflow runs - if not already there) and **switch to the patch branch that you created for the pull request.** (Alternatively, you can select the file via the Code tab.)



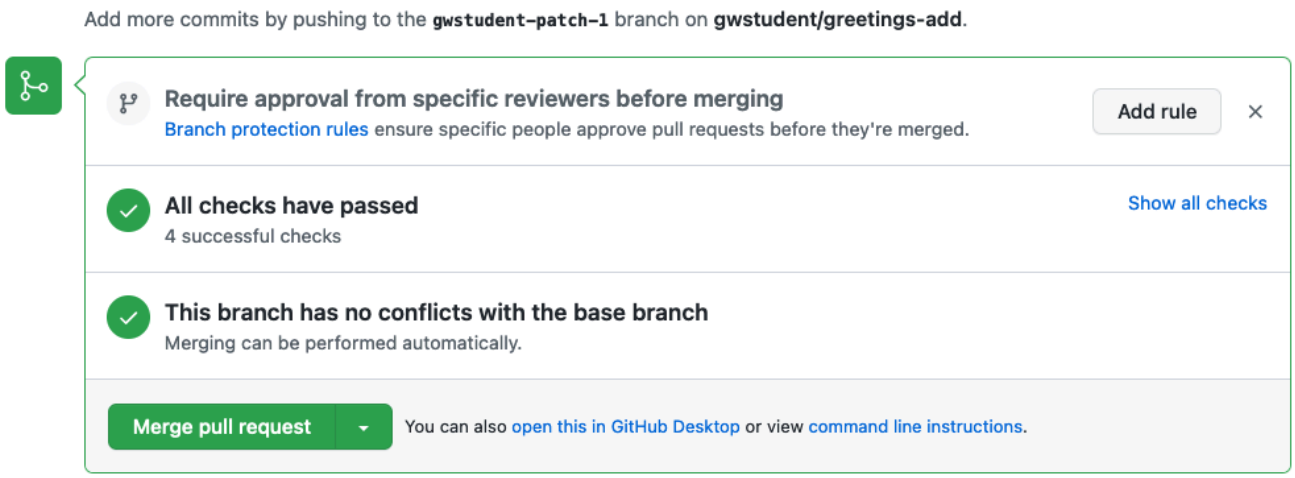
9. Edit the pipeline.yml file (use the pencil icon). Then update the line that has "uses : <your github userid>/arg-count-action@main" to actually have your GitHub userid in it.

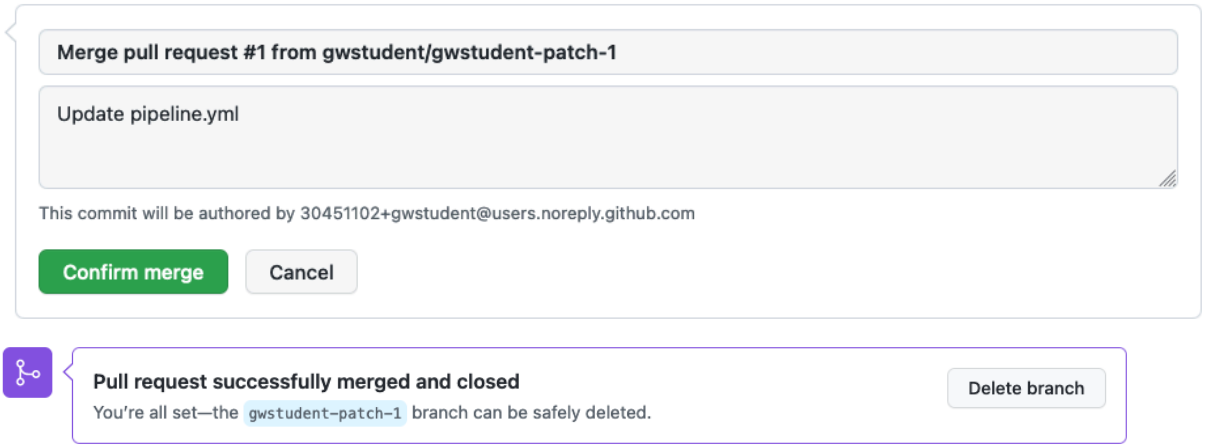
```
90
91    count-args:
92
93      runs-on: ubuntu-latest
94
95      steps:
96      - id: report-count
97        uses: gwstudent/arg-count-action@main
98        with:
99          arguments-to-count: ${{ github.event.inputs.myValues }}
100     - run: echo
101     - shell: bash
102       run: |
103         echo argument count is ${{ steps.report-count.outputs.arg-count }}
104
```

10. When you're done, click on the green "Commit changes…" button, add in a comment if you want, leave the selection set to "Commit directory to the … branch" so it will go into the same patch branch as before. Then select "Commit changes".

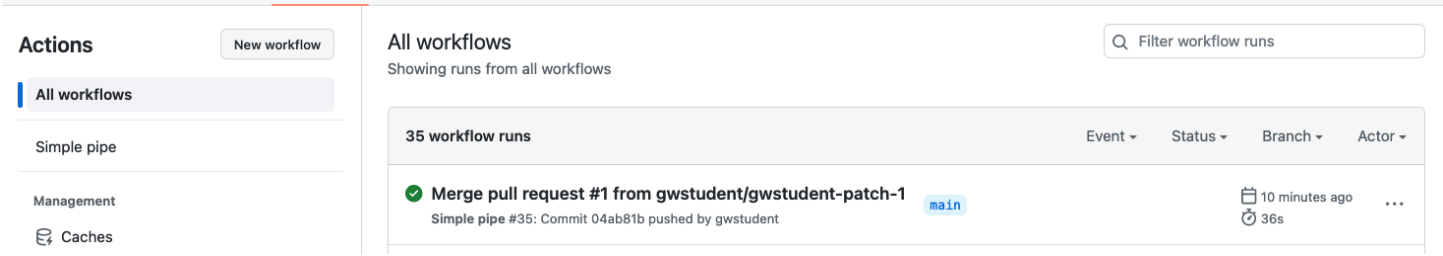


11. Now click on the "Pull requests" link at the top of the page and select the Pull Request again. Eventually all the checks should complete. You can now choose to "Merge pull request", confirm the merge and delete the branch.





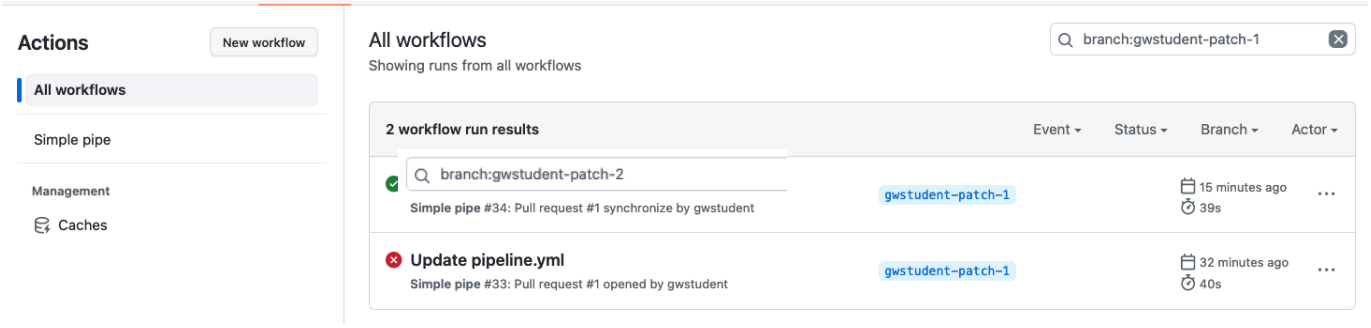
12. Afterwards, you should see that a new run of the workflows in main has been kicked off and will eventually complete.



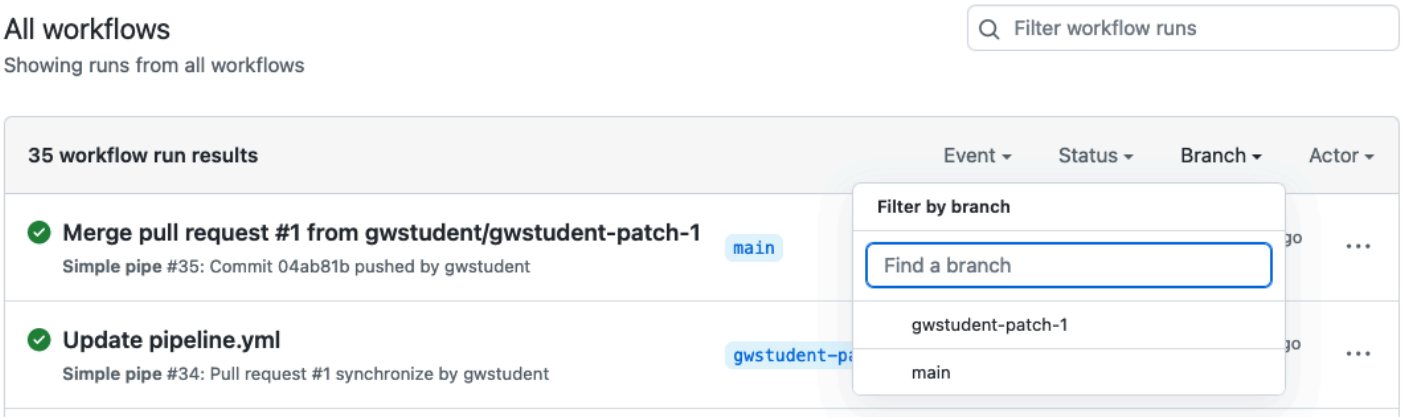
Lab 7: Exploring logs

Purpose: In this lab, we'll take a closer look at the different options for getting information from logs.

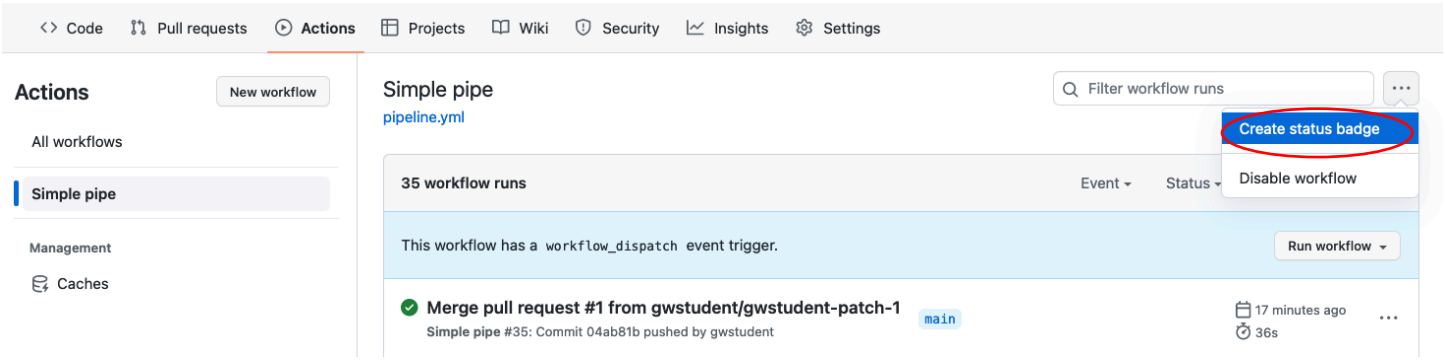
1. If not already there, switch back to the Actions tab. To the right of the list of workflows is a search box. Let's execute a simple search - note that only certain keywords are provided and not a complete search. Let's search for the workflow runs that were done for the branch that you used for the Pull Request in the last lab. Enter "**branch:<patch-branch-name>"** (no spaces) in the search box and hit enter.



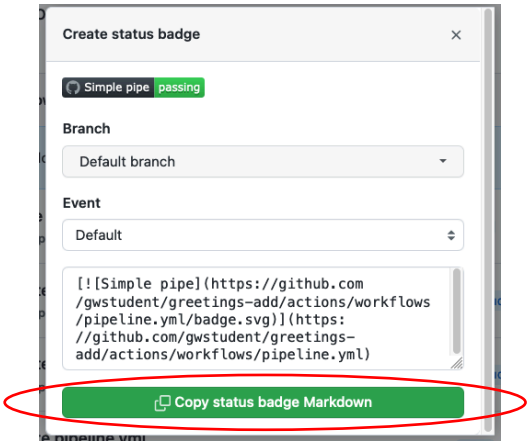
2. Click on the "X" at the right end of the search box to clear the entry. You can also accomplish the same thing by clicking on the items in the "workflow run results" bar. Clicking on one of the arrows next to them will bring up a list of values to select from that will also filter the list. Try clicking on some of them. Click on the "X" again when done.



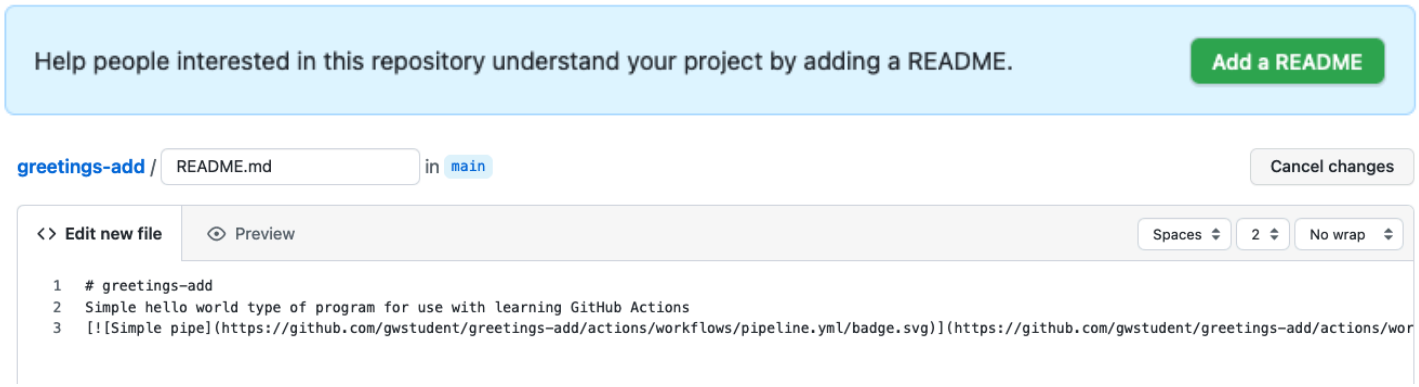
3. Make sure you are on the branch *main*. (Switch back if you need to.) Select the "Simple Pipe" workflow. You'll now have a box with "…" beside the search box that has some additional options. Click on the "…" beside the search box to see some of them. They include disabling the workflow and setting up a "badge" for the workflow. Let's go ahead and set up a badge now to show success/failure for running the workflow. Click on the entry for "Create status badge".



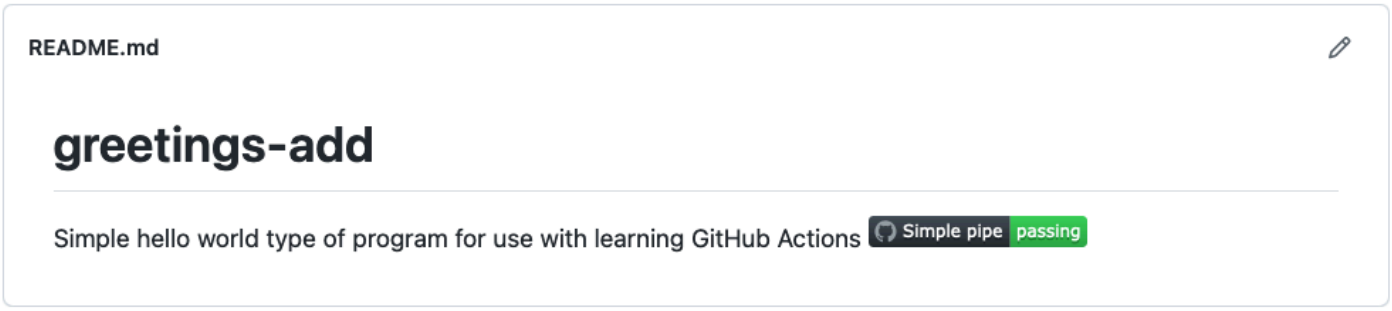
4. In the dialog that pops up, click on the entry for "Copy status badge Markdown". Then close the dialog.



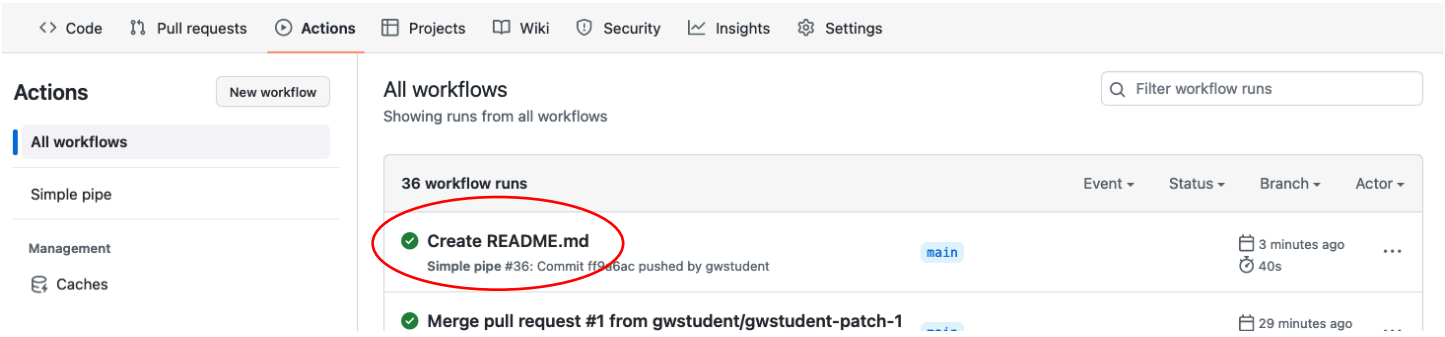
5. Click on the "<> Code" tab at the top of the project. At the bottom of the file list, click on the green button to "Add a README" (or edit the README if you already have one). Paste the code you copied in the previous step into the README.md text edit window.



6. Scroll down and commit your changes. Then you should see the badge showing up as part of the README.md content.



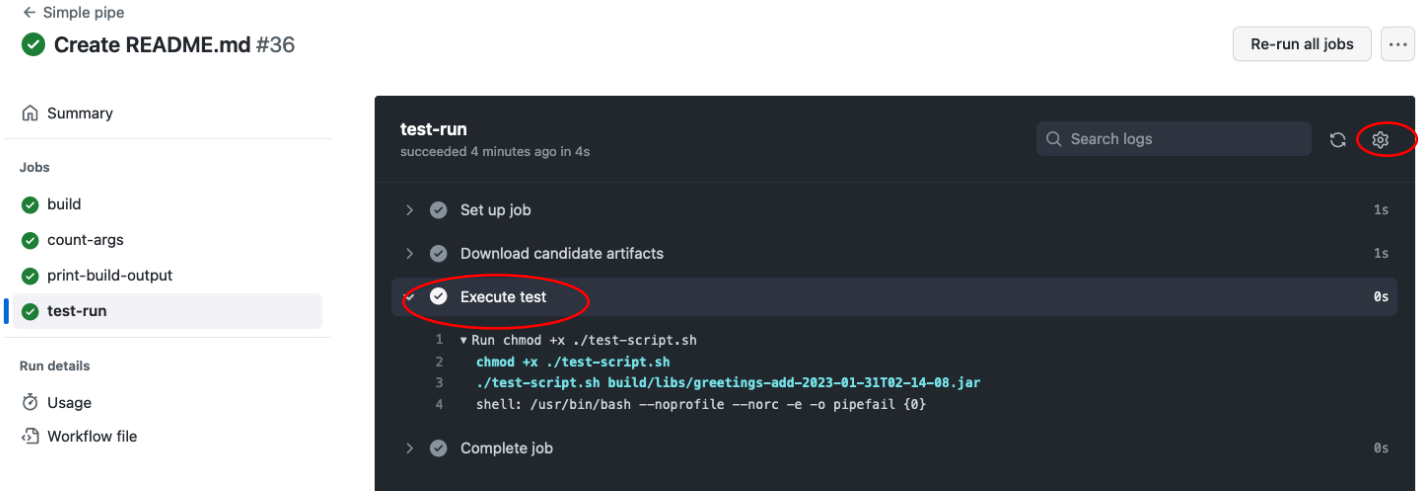
7. Click back on the Actions tab. Click on the name of the top run in the Workflow runs list. Notice that we have information in the large bar at the top about who initiated the change, the SHA1, the status, duration, and number of artifacts.



8. In the main part of the window, we have the job graph, showing the status and relationships between jobs. **Click on the "test-run" job**. In the screen that pops up, we can get more information about what occurred on the runner for that job.

First, let's turn on timestamps. **Click on the "gear" icon and select the "Show timestamps" entry.**

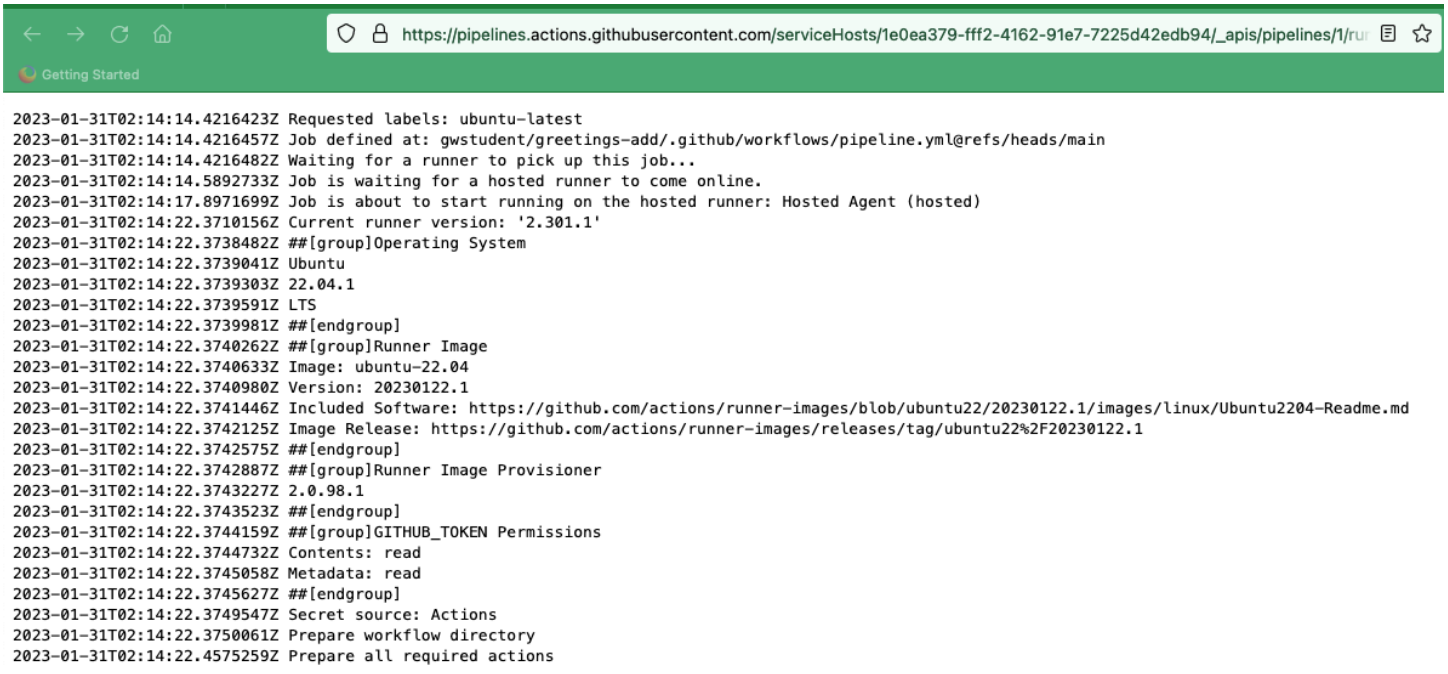
In the list of steps **click on the third item "Execute test"** to expand it. Then, in line 1 of that part, **click on the arrowhead after the timestamp** to expand the list and see all the steps executed in between.



9. We can get links to share to any line. Hover over any of the line numbers and then right-click to Copy Link, Open in a New Tab, or whatever you would like to do.

10. Click on the gear icon again. Notice there is an option to "Download log archive" if we want to get a copy of the logs locally. Or we can get a full view of the raw logs by clicking on the last entry.

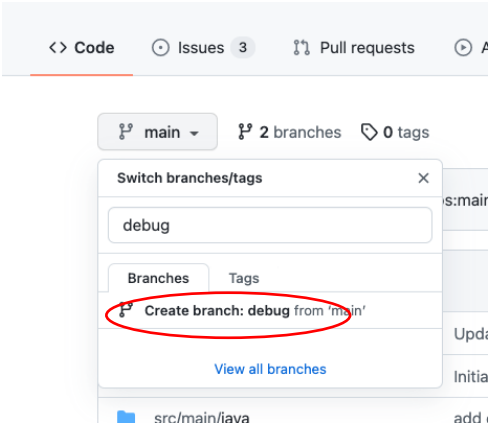
    Click on "View raw logs". When you are done looking at them, switch back to the workflow screen.



Lab 8: Looking at debug info

Purpose: In this lab, we'll look at some ways to get more debugging info from our workflows.

1. First, let's create a new branch in GitHub for the debug instances of our workflows. On the repository's Code page, click on the drop-down under "main", and enter "debug" in the "Find or create a branch…" field. Then click on the "Create branch: debug from 'main' link in the dialog.



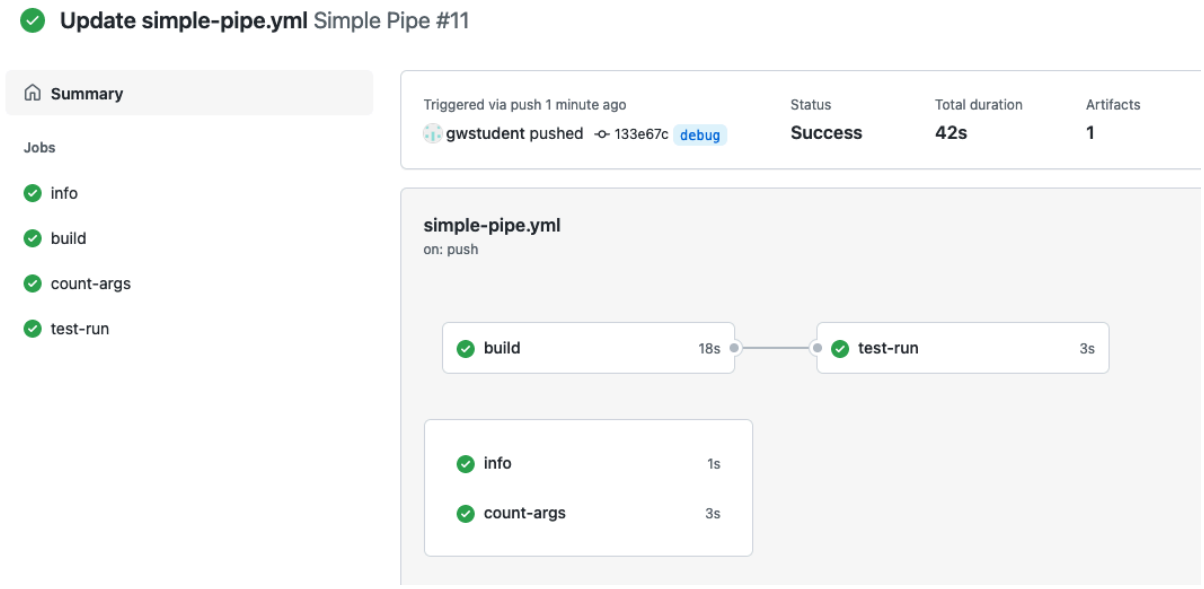
2. At this point you should be in the new branch - the "debug" branch. Go to the workflow file in .github/workflows and edit the pipeline.yaml file. **Change the references to "main" in the "on" section at the top to "debug".** Also, add a new job to surface some debug context.
Add in the lines below after the "jobs:" line. Pay attention to indenting again. A screenshot of how everything should look and lines up is further down. (For convenience, the text for the info job is also in a file in extra/info.txt.)

```
info:
  runs-on: ubuntu-latest

  steps:
  - name: Print warning message
    run: |
      echo "::warning::This version is for debugging only."
  - name: Dump context for runner
    env:
      RUNNER_CONTEXT: ${{ toJSON(runner) }}
    run:
      echo "::debug::Runner context is above."
```

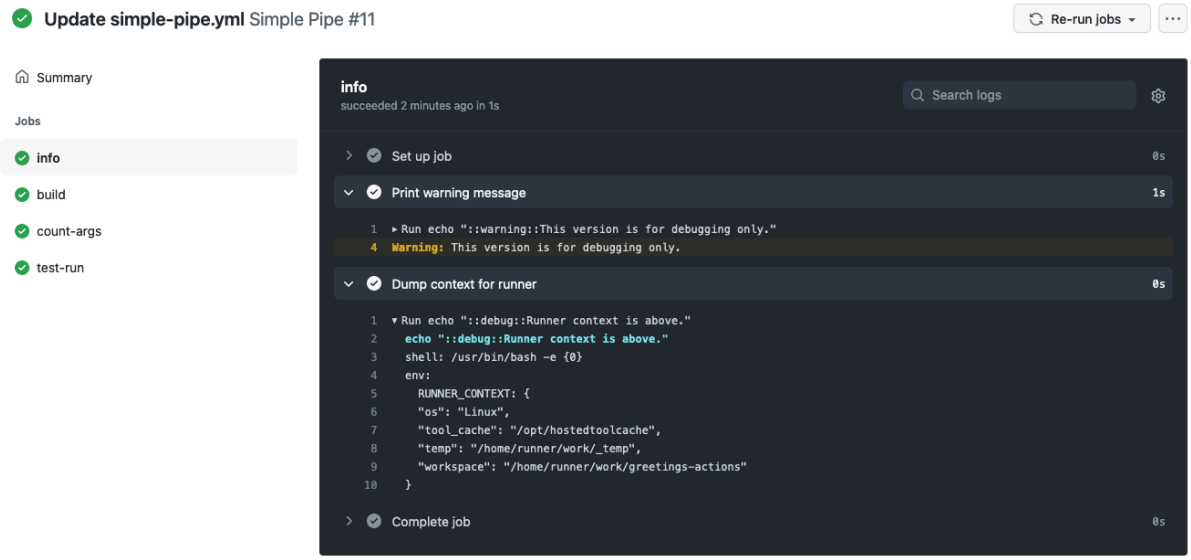
Edit file | Preview changes

```
on:
  push:
    branches: [ debug ]
  pull_request:
    branches: [ debug ]
  workflow_dispatch:
    inputs:
      myValues:
        description: 'Input Values'

jobs:

  info:
    runs-on: ubuntu-latest

    steps:
    - name: Print warning message
      run: |
        echo "::warning::This version is for debugging only."
    - name: Dump context for runner
      env:
        RUNNER_CONTEXT: ${{ toJSON(runner) }}
      run:
        echo "::debug::Runner context is above."
```

3. When you are done making the changes, commit as usual. Switch back to the Actions tab and click on the currently running workflow. Then click on the "info" job in the graph and look at the logs.

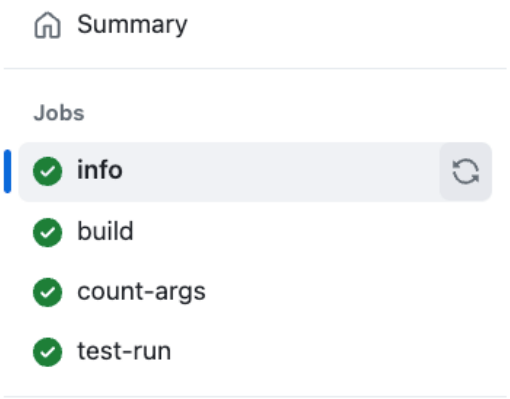


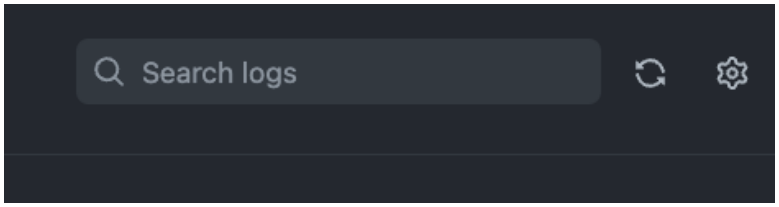
4. Expand the entries for "Print warning message" and "Dump context for runner" to see the outputs for those.



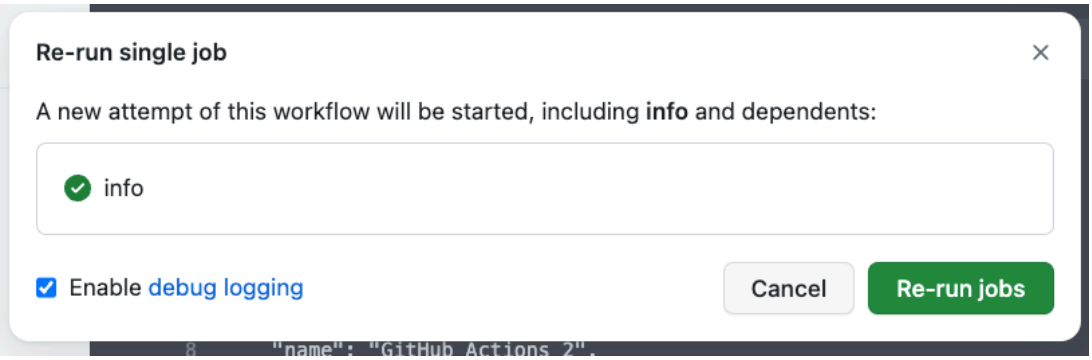
5. Notice that while we can see both commands that echo our custom "warning" and "debug" messages - only the output of the warning message actually is displayed, not the output of the debug message. There are a couple of ways to get the debugging info.

6. The first way is to simply rerun the job. If you hover over the job name under the Summary section, you can see two curved arrows appear. Click on those. (The same arrows are also available in the upper right of the logs window - next to the gear icon.)

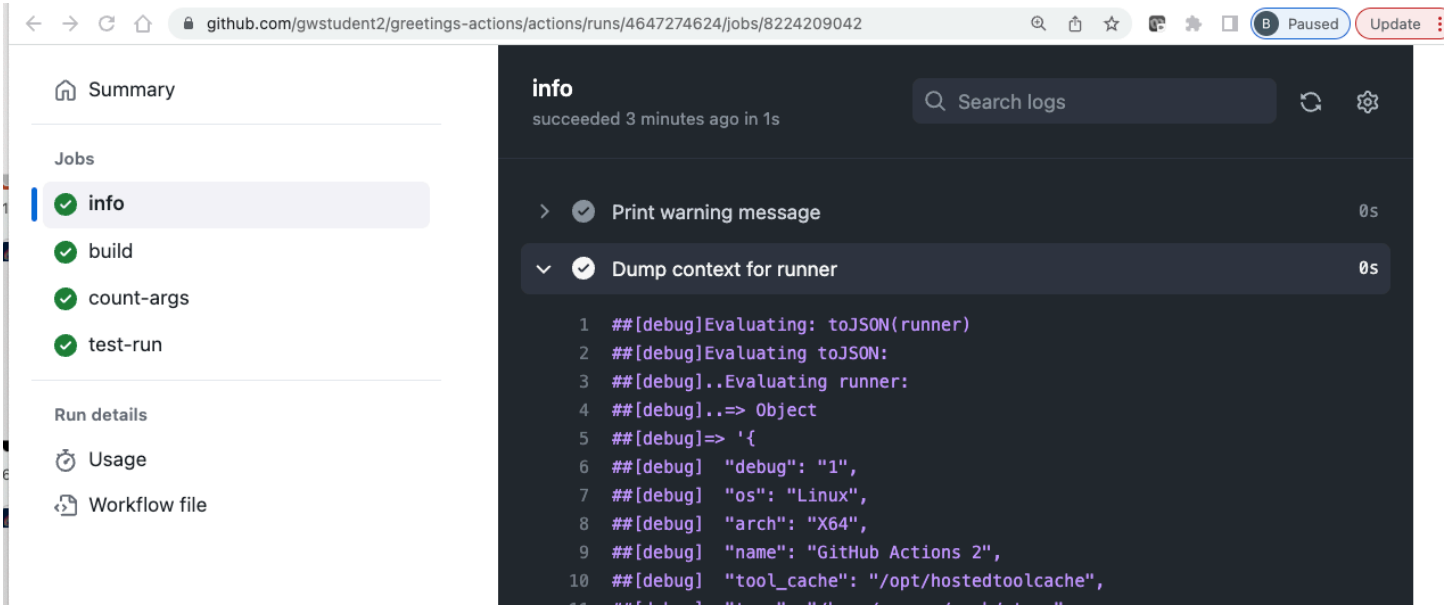




7. This will bring up a dialog to re-run that job (and any dependent jobs) - with a checkbox to click to *Enable debug logging*. Click that box and and then click the "Re-run jobs" button.

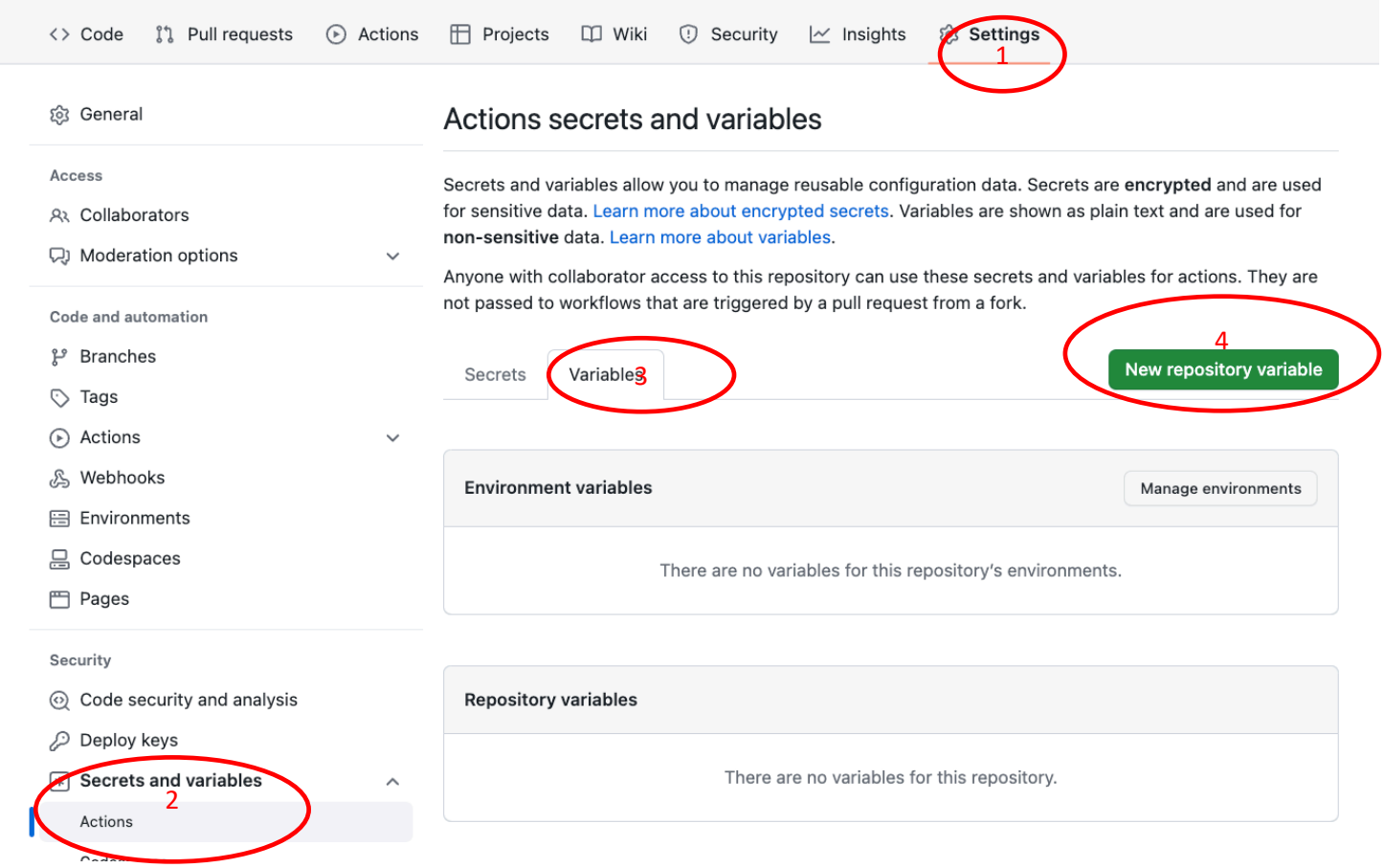


8. After the job is re-run, if you look at the latest output, and expand the "Dump context for runner" step, you'll see the actual debug output.

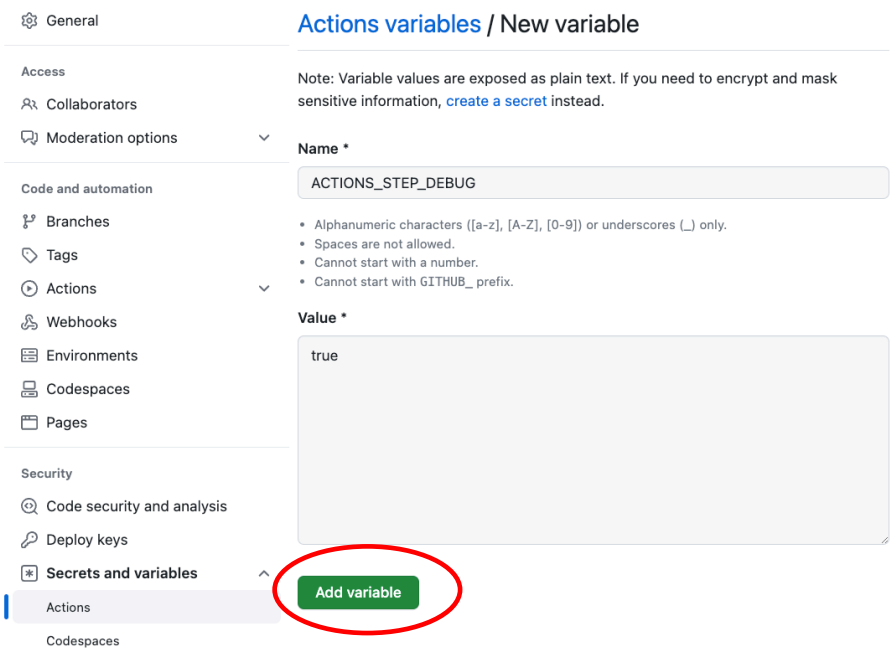


9. However, that doesn't cause debug info to show up for commits. We do that by enabling a secret or a variable for ACTIONS_STEP_DEBUG. Since this setting is not sensitive information, we'll use a variable.

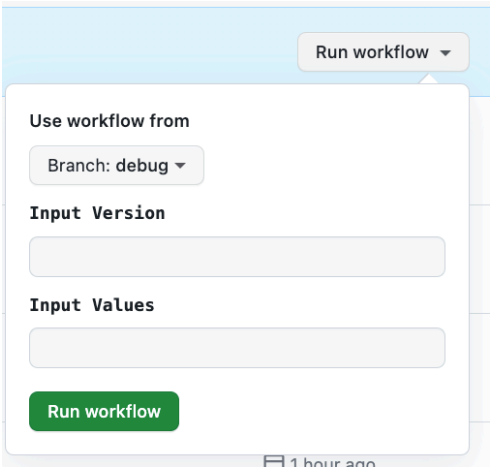
10. To do this, go to the repository's top menu and select "Settings". Then on the left-hand side, under "Security", select "Secrets and variables", and then "Actions" under that. Select the "Variables" tab and "New repository variable".



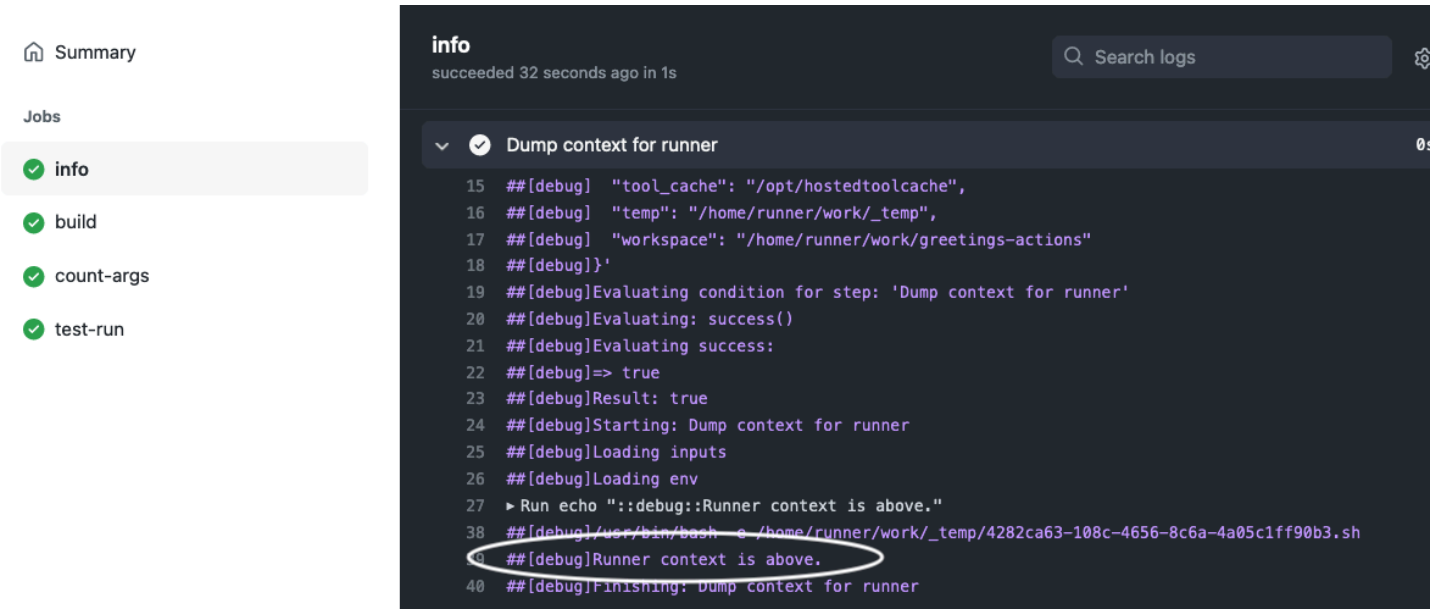
11. On the next screen, enter **ACTIONS_STEP_DEBUG** for the name, set the value to *true* and click on the *Add variable* button.



12. Now, switch back to the "Actions" tab, select the "Simple Pipe" workflow, and click on the "Run workflow" button. **Select "debug" from the list for the branch.** Enter in any desired arguments. Then click the green "Run workflow" button to execute the workflow.



13. A new run will be started. Go into it and select the "info" job. In the output now, if you expand the sections, you should be able to see a lot of "##[debug]" messages including the one you added in the "Dump context for runner" section.



14. Note that the debug log info will be turned on for all runs from here on - as long as the repository variable exists and is set to "true".

Lab 9 – Securing inputs

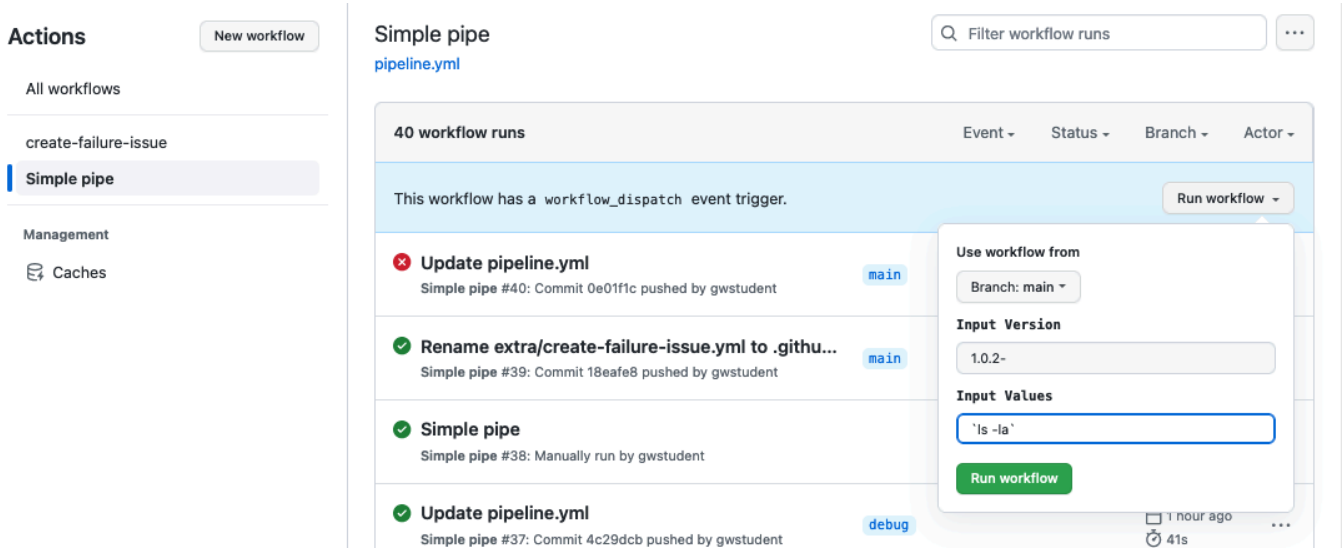
Purpose: In this lab, we'll look at how to plug a potential security hole with our inputs.

1. Make sure you're in the "main" branch. Switch to the pipeline.yml file in the .github/workflows directory and look at the "test-run" job and in particular, this line in the "Execute test" step:

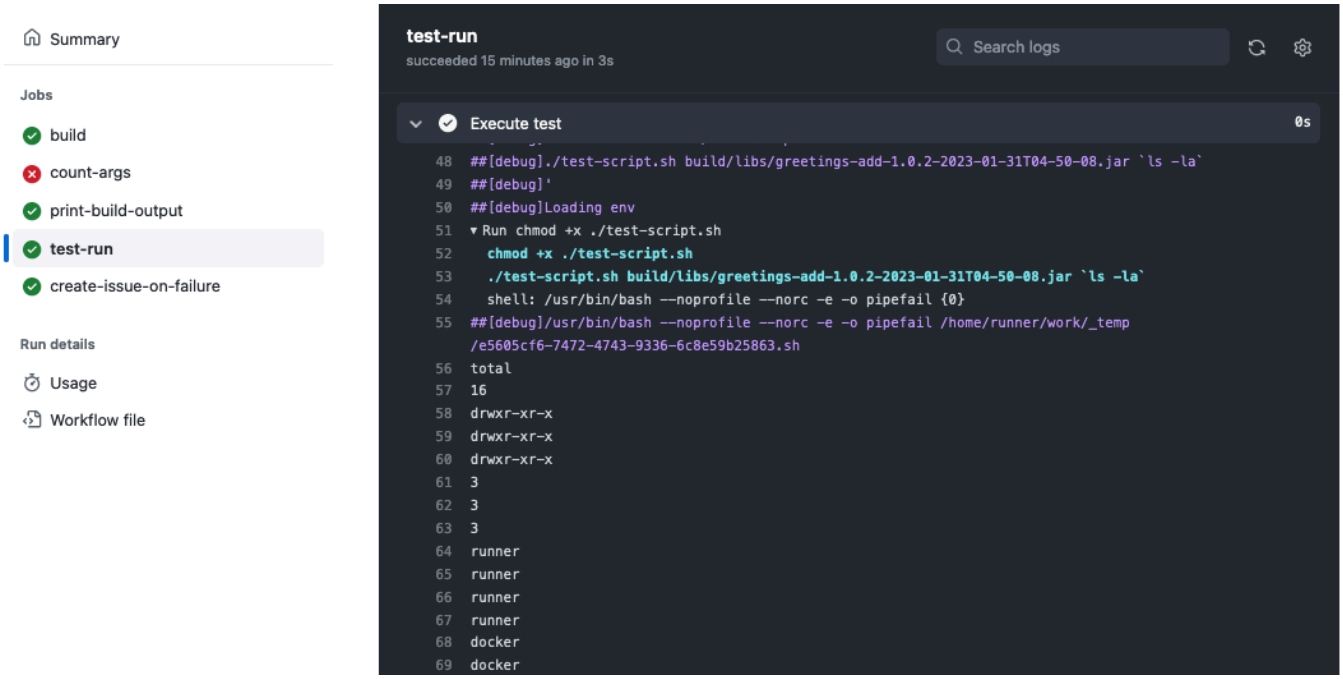
```
./test-script.sh ${{ needs.build.outputs.artifact-path }} ${{ github.event.inputs.myValues }}
```

```
84
85      - name: Execute test
86        shell: bash
87        run: |
88          chmod +x ./test-script.sh
89          ./test-script.sh ${{ needs.build.outputs.artifact-path }} ${{ github.event.inputs.myValues }}
90
91    count-args:
```

2. When we create our pipelines that execute code based on generic inputs, we must be cognizant of potential security vulnerabilities such as injection attacks. This code is subject to such an attack. To demonstrate this, use the workflow_dispatch event for the workflow in the Actions menu, put in a version and pass in the following as the arguments in the arguments field (NOTE: That is two backquotes around ls -la) `` `ls -la` `` Then hit "Run workflow".

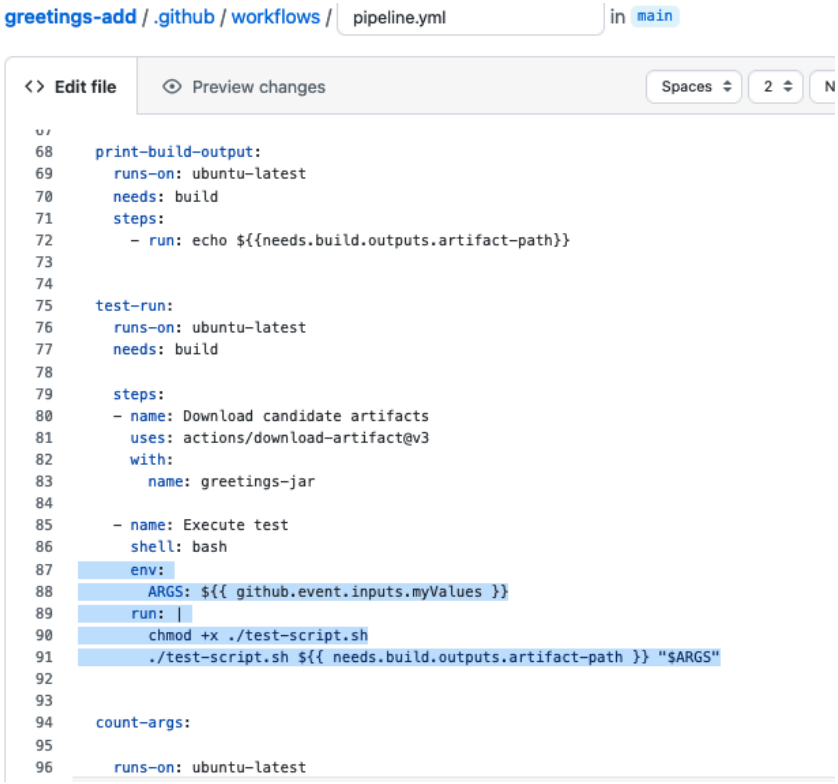


3. After the run completes, look at the output of the "test-run" job. Select the "Execute test" step and expand the logs. Notice that the step itself ran successfully, but it has actually run the `ls -la` command directly on the runner system. (Scroll down past the initial debug info to see it - around line 60.) The command was innocuous in this case, but this could have been a more destructive command.



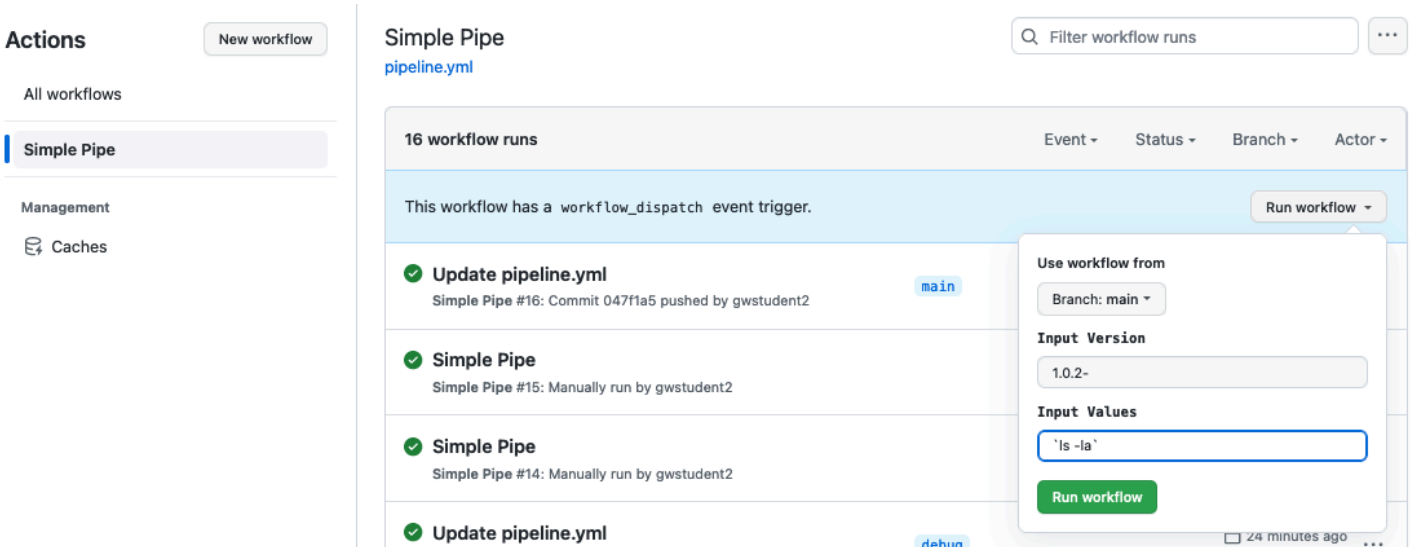
4. Let's fix the command to not be able to execute the code in this way. We can do that by placing the output into an environment variable first and then passing that to the step. Edit the *pipeline.yaml* file and change the code for the "Execute test" step to look like the following (pay attention to how things line up):

```
env:
  ARGS: ${{ github.event.inputs.myValues }}
run: |
  chmod +x ./test-script.sh
  ./test-script.sh ${{ needs.build.outputs.artifact-path }} "$ARGS"
```



5. Commit back the changes and wait till the action run for the push completes.

6. Now, you can execute the code again with the same arguments as before.



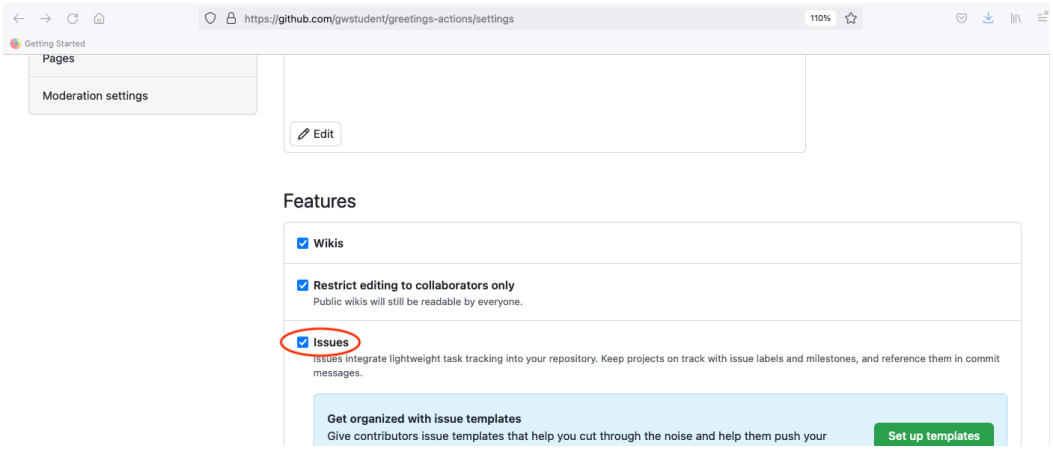
7. Notice that this time, the output did not run the commands, but just echoed them back out as desired.

END OF LAB

Lab 10: (Bonus/Optional) Chaining workflows, using conditionals, and working with REST APIs in workflows.

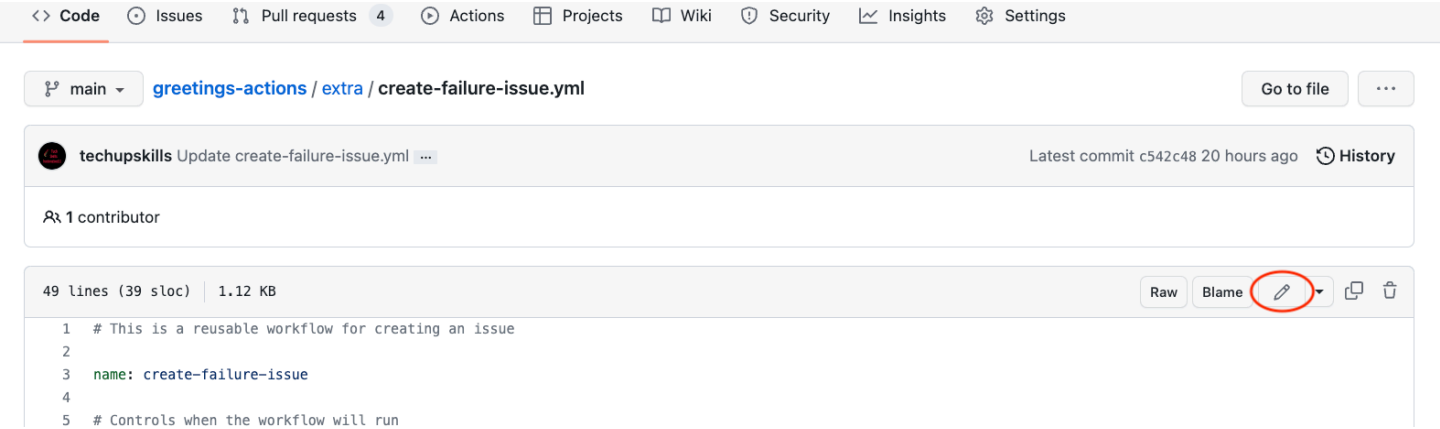
Purpose: Learning one way to drive one workflow from another.

1. 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. But first, we need to ensure that the "Issues" functionality is turned on for this repository. Go to the project's Settings main page, scroll down and under "Features", make sure the "Issues" selection is checked.

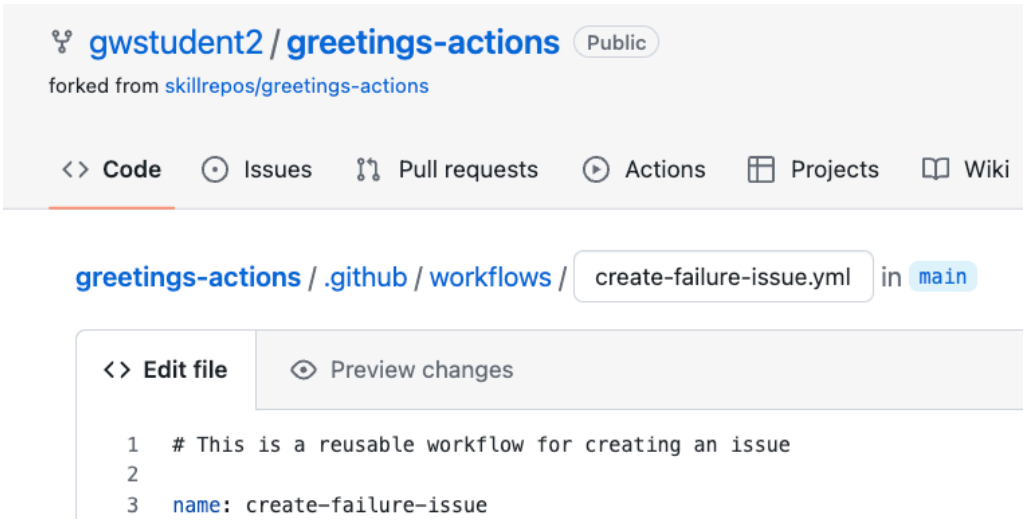


2. The workflow to create the issue using a REST API call is already written to save time. It is in the main project under "extra/create-failure-issue.yml". You need to get this file in the .github/workflows directory. You can just move it via GitHub with the following steps.

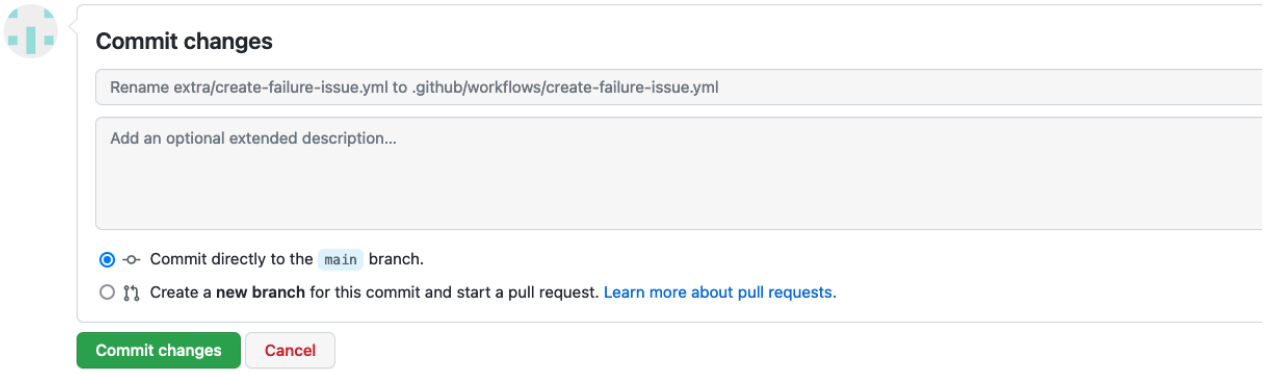
    a. In the repository, browse to the "extra" folder and to the "create-failure-issue.yml" file.
    b. Take a few moments to look over the file and see what it does. Notice that:
        i. It has a workflow_call section in the "on" area, which means it can be run from another workflow.
        ii. It 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.



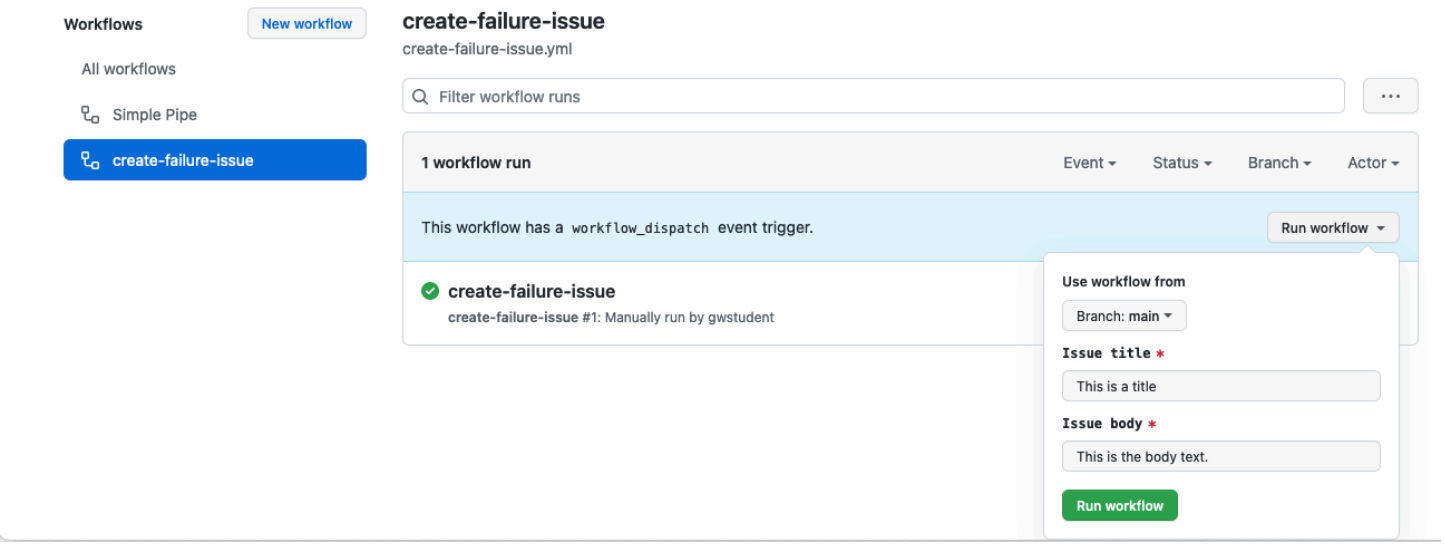
    d. In the filename field at the top, change the name of file. Use the backspace key to backspace over "extra/" making sure to backspace over the word. Then type in the path to put it in the workflows ".github/workflows/create-failure-issue.yml".



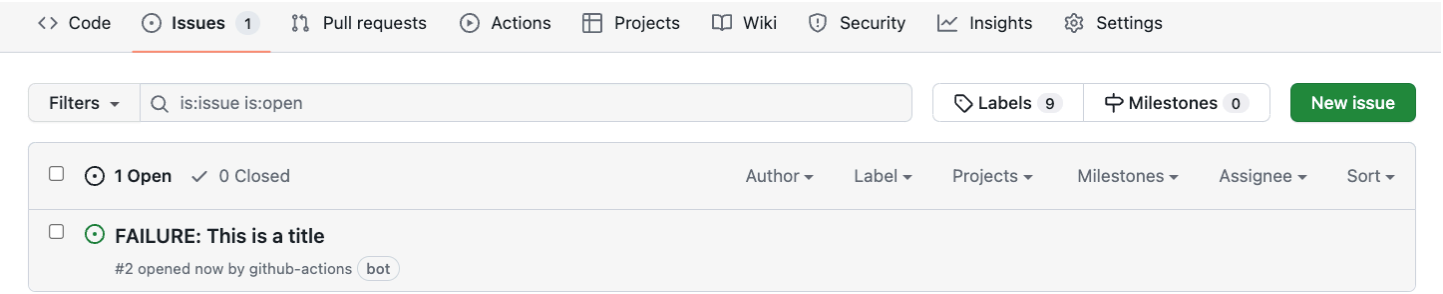
e. To complete the change, scroll to the bottom of the page, and click on the green "Commit changes" button.



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.



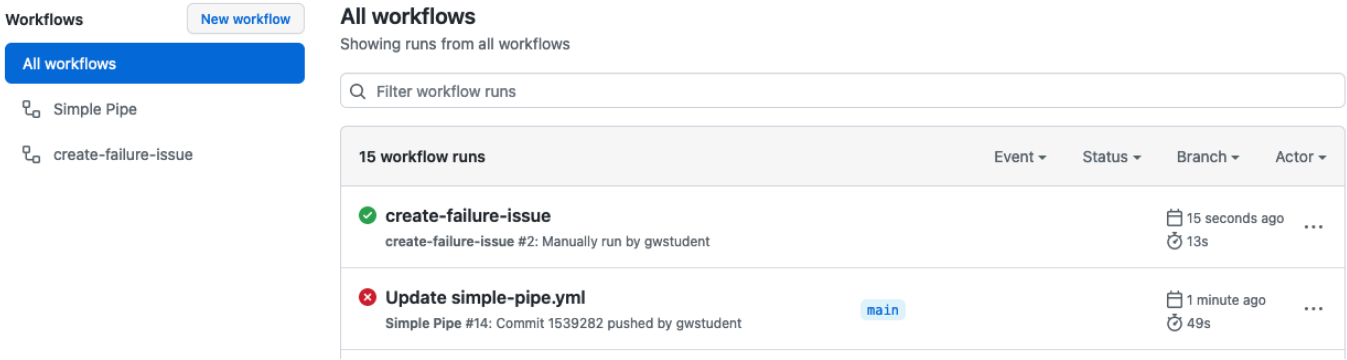
5. Now that we know that our new workflow works as expected, we can make the changes to the previous workflow to "call" this if we fail. Edit the pipeline.yml file and add the following lines as a new job and set of steps at the end of the workflow. (For convenience, these lines are also in the file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

```
  create-issue-on-failure:

    permissions:
      issues: write
    needs: [test-run, count-args]
    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 GitHub Action workflow ** ${{
github.workflow }} **"
```

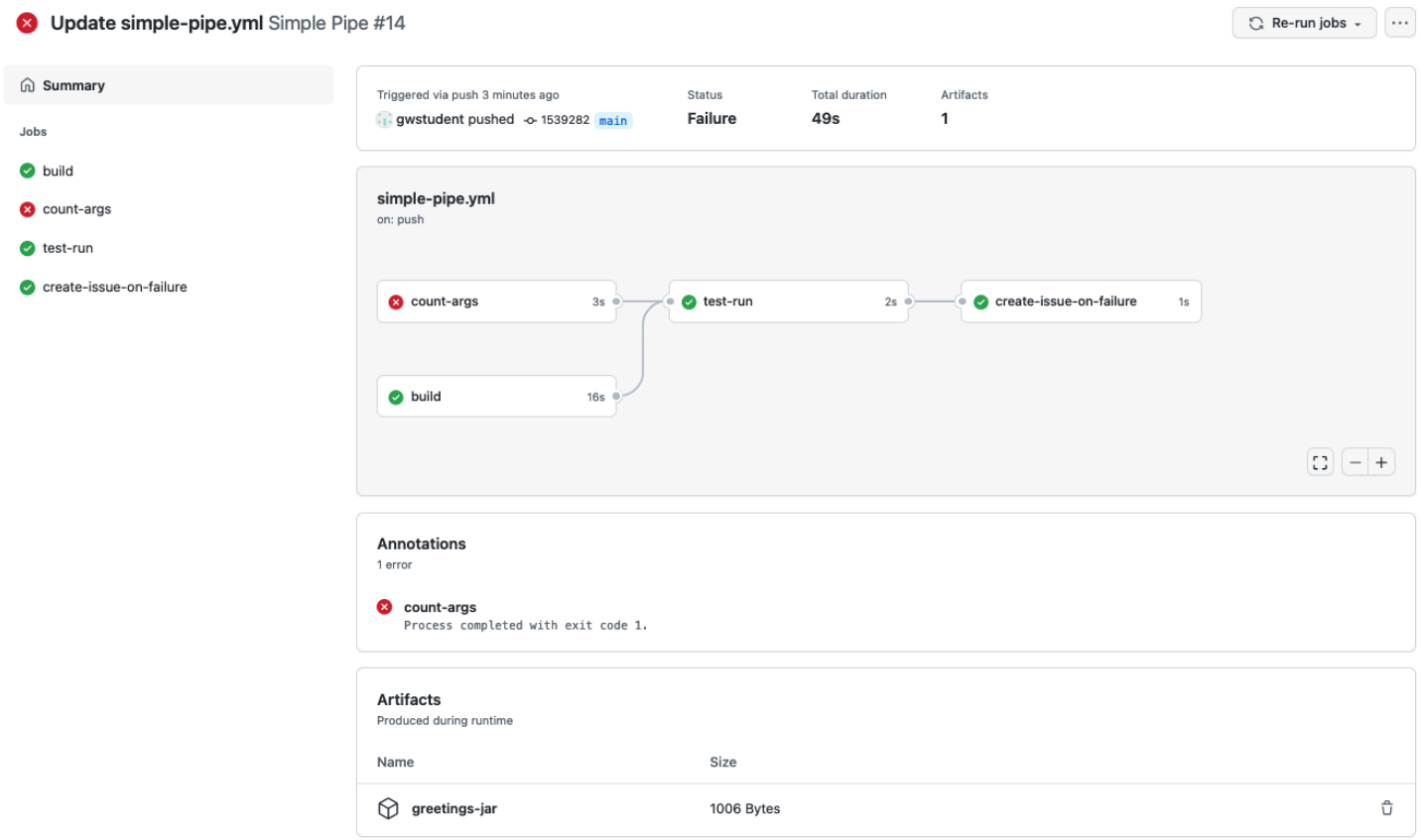
6. In order to have this executed via the "if" statement, we need to force a failure. We can do that by simply adding an "exit 1" line at the end of the "count-args" job (right above the job you just added).

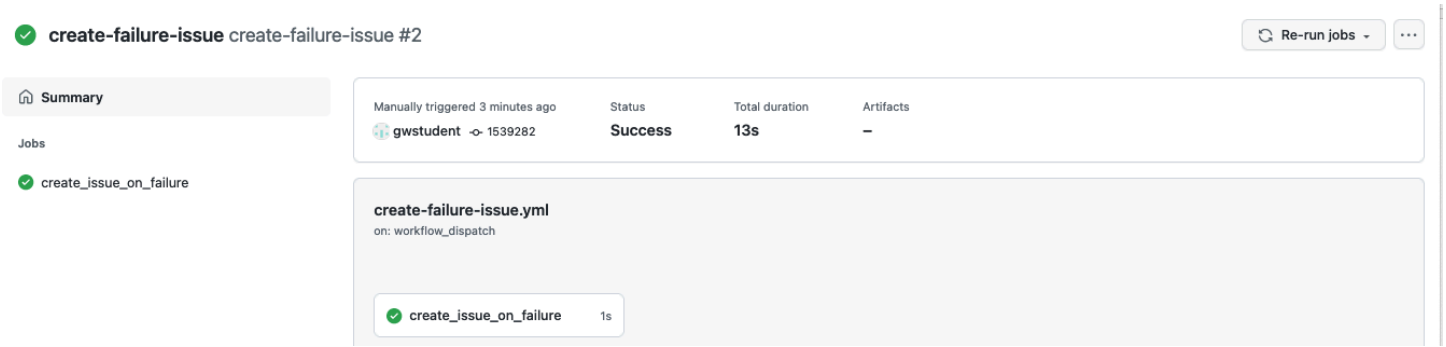
Make that change too. (A screenshot is below showing what the changes should look like. The "exit 1" is line 65 in the figure.)

7. After you've made the changes, commit them. At that point, you should get a run of the workflow. Click back to the Actions tab to watch it. After a few minutes, it will complete, and the "count-args" job will fail. This is expected because of the "exit 1" we added. But in a few moments, the create-issue-on-failure job should kick in and invoke the other workflow and produce a new ticket.

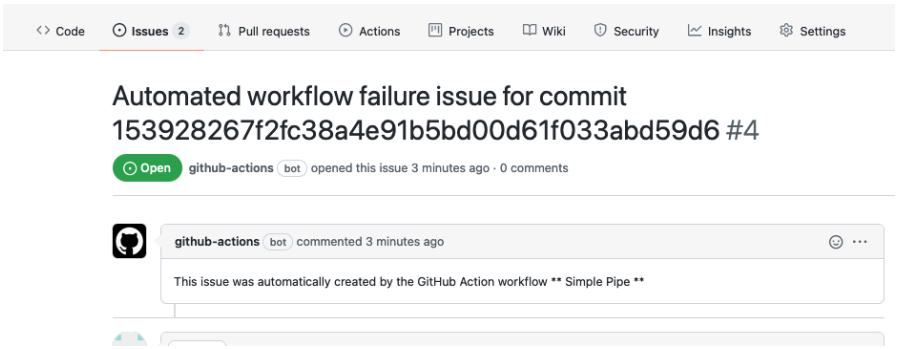


8. You can look at the graphs from the runs of the two workflows if you want.





9. Under "Issues", you can also see the new ticket that was opened with the text sent to it.



THE END - THANKS!