

CI/CD and DevOps in 3 Weeks

Week 3

Revision 1.3 – 08/22/22

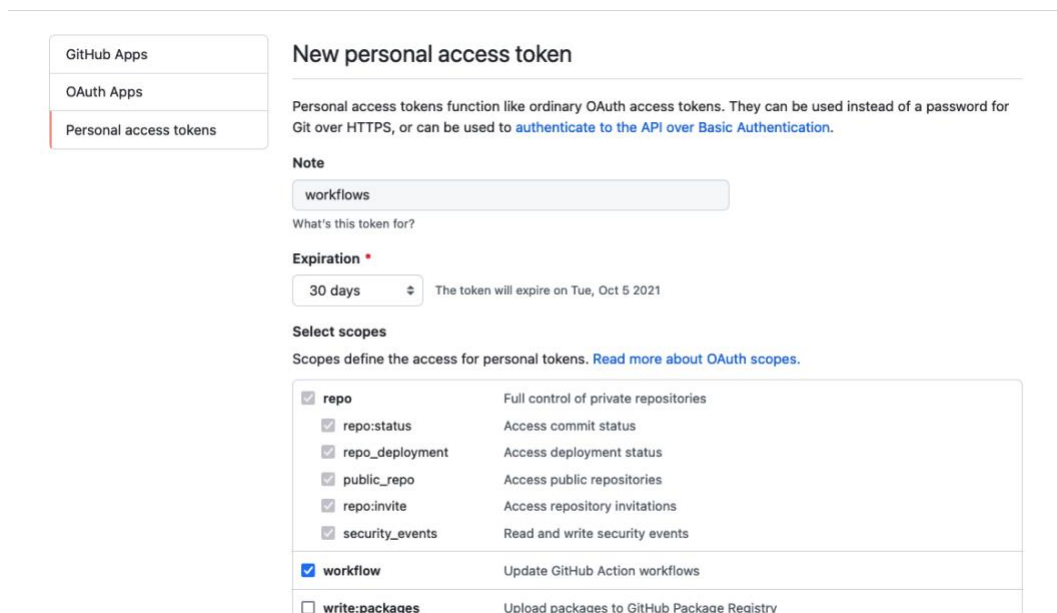
Tech Skills Transformations LLC / Brent Laster

Lab 10: Getting a Personal Access Token and enhancing the workflow-dispatch code

1. For this lab, we need to prepare a Personal Access Token (PAT) and add it to a secret that our workflow can reference. If you already have a PAT, you may be able to use it if it has access to the project. If not, you'll need to create a new one. Go to <https://github.com/settings/tokens>.

(Alternatively, on the GitHub repo screen, click on your profile picture in the upper right, then select "Settings" from the drop-down menu. You should be on the <https://github.com/settings/profile> screen. On this page on the left-hand side, select "Developer settings" near the bottom. On the next page, select "Personal access tokens".)

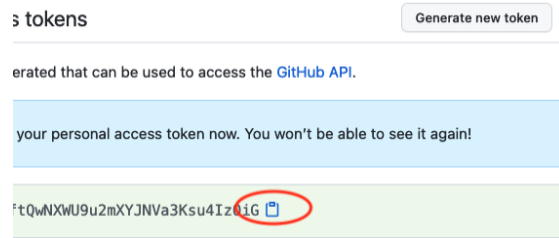
2. Click on "Generate new token". Confirm your password if asked. In the "Note" section enter some text, such as "workflows". You can set the "Expiration" time as desired or leave it as-is. Under "Select scopes", assuming your repository is public, you can just check the boxes for "repo" and "workflow". Then click on the green "Generate token" at the bottom.



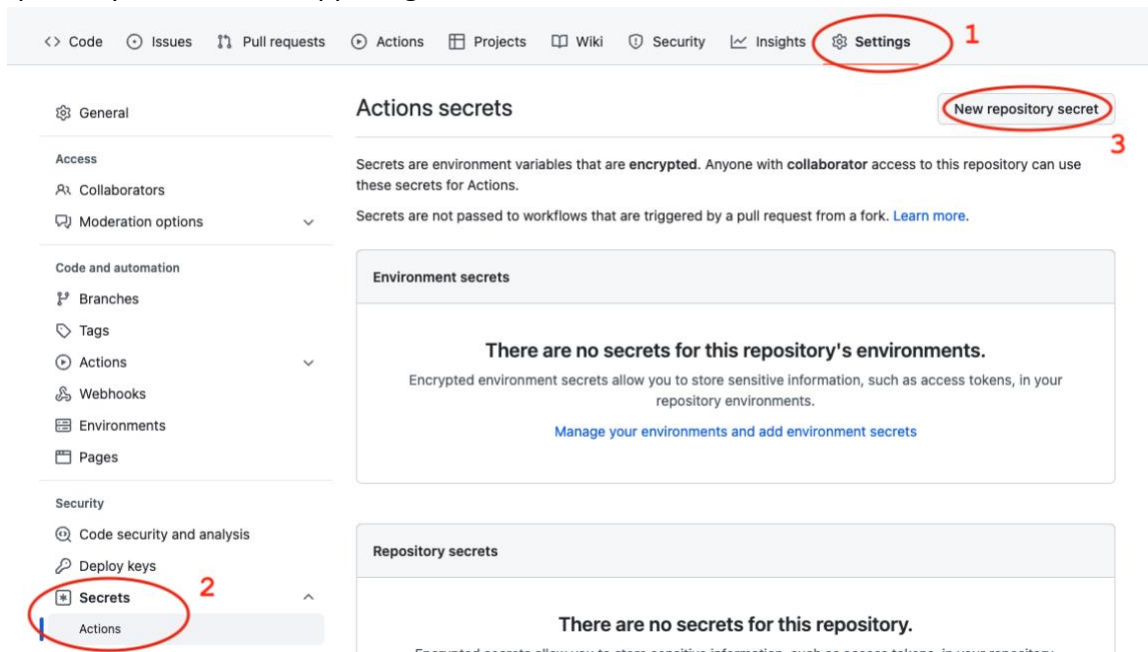
The screenshot shows the GitHub 'New personal access token' interface. On the left, a sidebar contains links for 'GitHub Apps', 'OAuth Apps', and 'Personal access tokens', with the last one highlighted. The main content area is titled 'New personal access token' and includes a descriptive paragraph about PATs. Below this is a 'Note' section with a text input field containing 'workflows'. The 'Expiration' section shows a dropdown set to '30 days' and a note that the token will expire on Tuesday, October 5, 2021. The 'Select scopes' section lists various permissions with checkboxes: 'repo' (checked), 'repo:status', 'repo_deployment', 'public_repo', 'repo:invite', 'security_events', 'workflow' (checked), and 'write:packages' (unchecked). Each scope has a brief description of its permissions.

Scope	Description
<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events
<input checked="" type="checkbox"/> workflow	Update GitHub Action workflows
<input type="checkbox"/> write:packages	Upload packages to GitHub Package Registry

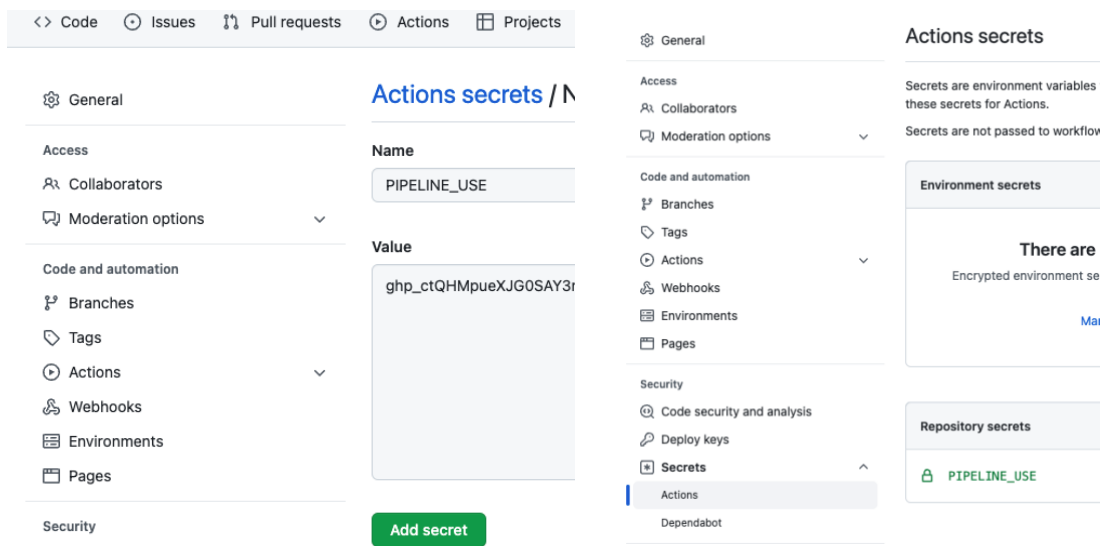
3. After the screen comes up that shows your new token, make sure to copy it and store it somewhere you can get to it.



- Now we'll create a new secret and store the PAT value in it. Go to the repository and in the top menu select "Settings". Then on the left-hand side, select "Secrets" and select "Actions". Now, click on the "New repository secret" in the upper right to create a new secret for the action to use.



- For the Name of the new secret, use PIPELINE_USE. Paste the value from the PAT into the Value section. Then click on the "Add secret" button at the bottom. After this, the new secret should show up at the bottom.



6. Now let's update the workflow to allow providing a version input for the workflow_dispatch code. First, add an additional parameter to allow you to put in a version for the artifact in the workflow_dispatch code in the "on:" section. Add the two lines for a new input value as shown below.

```
myVersion:
  description: 'Input Version'
```

```
8  name: Java CI with Gradle
9
10 on:
11   push:
12     branches: [ "main" ]
13   pull_request:
14     branches: [ "main" ]
15   workflow_dispatch:
16     inputs:
17       myVersion:
18         description: 'Input Version'
19     myValues:
20       description: 'Input Values'
21
```

7. Next, add the version input as an alternative to use in the tagging step if the changelog.outputs.version is empty. Make the update highlighted below.

```
- name: Tag artifact
  run: mv build/libs/greetings-ci.jar build/libs/greetings-ci-`${ steps.changelog.outputs.version ||
github.event.inputs.myVersion }`.jar
```

```
48 - name: Build with Gradle
49   uses: gradle/gradle-build-action@v2.2.1
50   with:
51     arguments: build
52
53 - name: Tag artifact
54   run: mv build/libs/greetings-ci.jar build/libs/greetings-ci-`${ steps.changelog.outputs.version || github.event.inputs.myVersion }`.jar
55
56 - name: Upload Artifact
```

8. Next, we'll update the call to the testing script with a similar approach. If the changelog.outputs.version is empty, we'll try to use the myVersion input. Make the highlighted change below.

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

```
75
76 - name: Execute test
77   shell: bash
78   run: |
79     chmod +x ./test-script.sh
80     ./test-script.sh `${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }` `${ github.event.inputs.myValues }`
81
```

9. Go ahead and commit the changes with a "fix: <message>" commit message.
10. You can now launch the workflow with the updated workflow_dispatch event if you want to try it out.

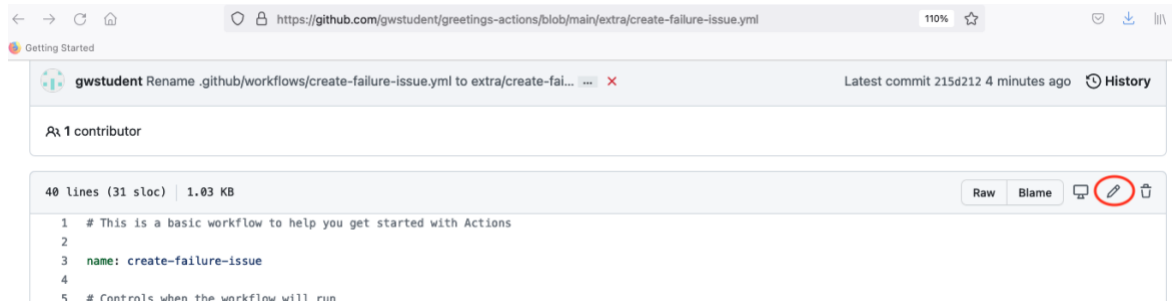
The screenshot shows the GitHub Actions interface for the 'Java CI with Gradle' workflow. The left sidebar shows the 'Workflows' section with a 'New workflow' button and a list of workflows including 'Java CI with Gradle' and 'create-failure-issue'. The main area shows the 'Java CI with Gradle' workflow with a search bar and a list of 72 workflow runs. The runs are filtered by 'Event', 'Status', 'Branch', and 'Actor'. A 'Run workflow' button is visible in the top right. A modal is open showing options to run the workflow from a specific branch and with specific input values.

END OF LAB

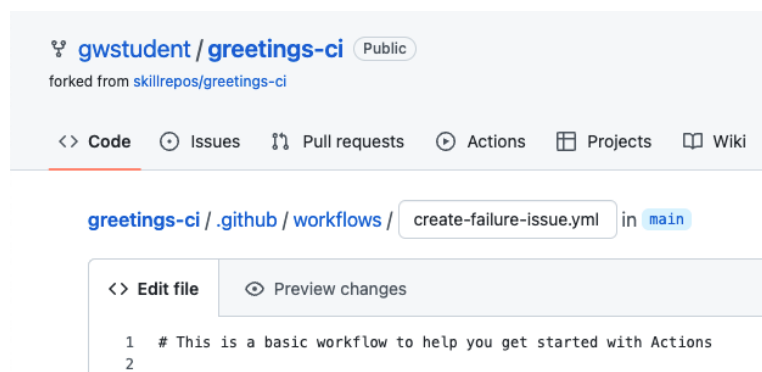
Lab 11: Working with fast feedback and automatically reporting issues

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

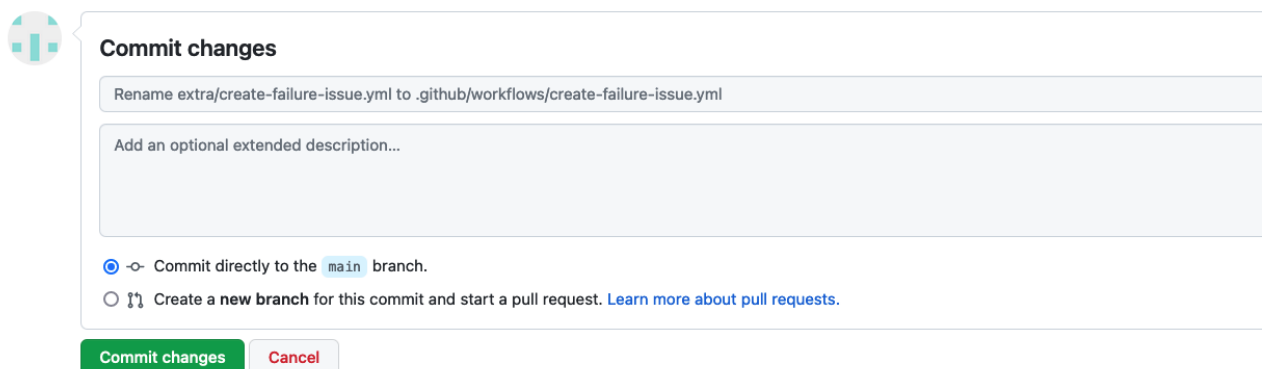
1. We're going to create a new workflow that will be able to automatically create a GitHub issue in our repository. And then we will invoke that workflow from our current workflow. The workflow to create the issue using a REST API call is already written to save time. It is in the main project under "extra/create-failure-issue.yml". You need to get this file in the .github/workflows directory. To do that, you can clone and move it. Or you can just do 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_dispatch section in the "on" area, which means it can be run manually.
 - ii. It has two inputs - a title and body for the issue.
 - iii. 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.



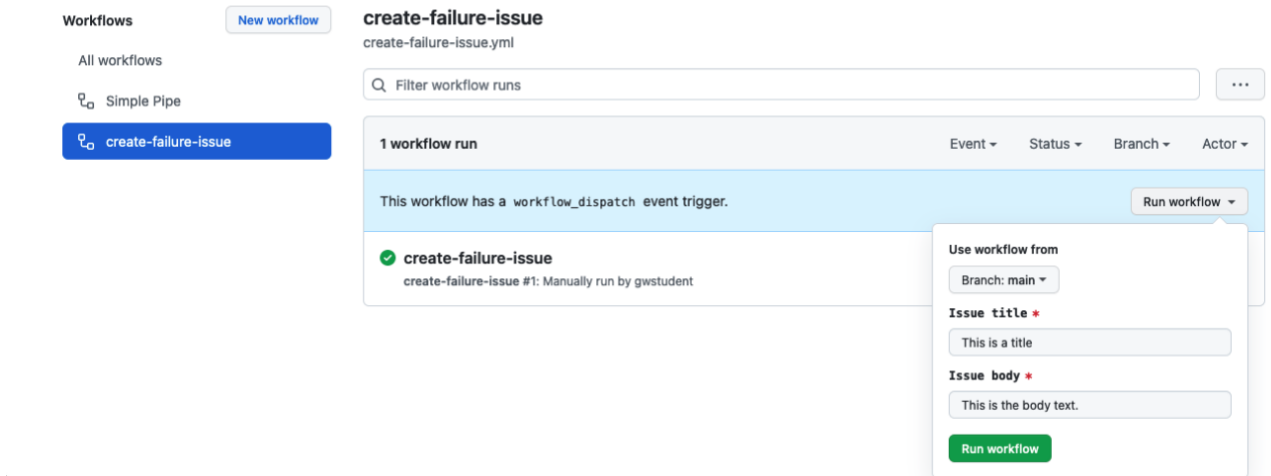
- d. In the filename field, change the name of the file. Use the backspace key to backspace over "extra/" making sure to backspace over the word. Then type in the path to put it in the workflows ".github/workflows/create-failure-issue.yml".



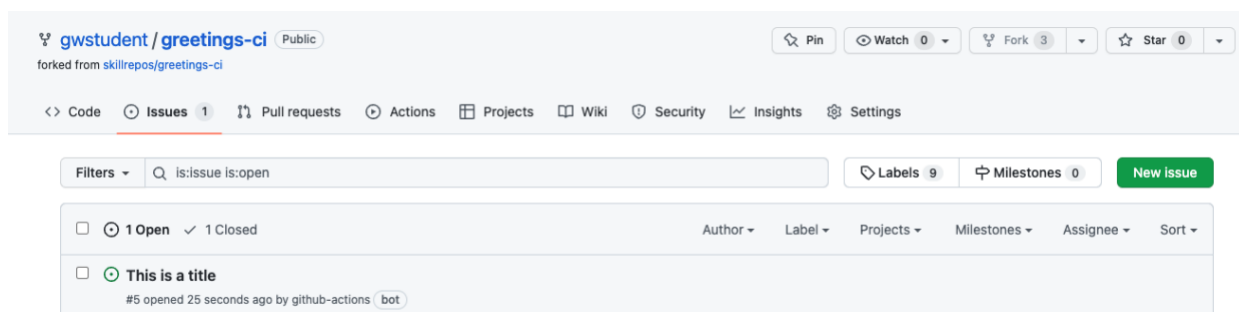
- e. To complete the change, scroll to the bottom of the page, and click on the green "Commit changes" button.



2. 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".



- 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.



- 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.) The "create-issue-on-failure" job name should align with the "test-run" job name. See screenshot further down.

create-issue-on-failure:

```
runs-on: ubuntu-latest
needs: test-run
if: always() && failure()
steps:
  - name: invoke workflow to create issue
    run: >
      curl -X POST
      -H "authorization: Bearer ${ secrets.PIPELINE_USE }"
      -H "Accept: application/vnd.github.v3+json"
      "https://api.github.com/repos/${ github.repository }/actions/workflows/create-
failure-issue.yml/dispatches"
      -d '{"ref":"main",
        "inputs":
```

```

        {"title":"Automated workflow failure issue for commit ${github.sha}"},"
        "body":"This issue was automatically created by the GitHub Action workflow
** ${github.workflow} **"}
    }'

```

```

57 |         name: greetings-jar
58 |         path: |
59 |             build/libs
60 |             test-script.sh
61 |
62 |
63 |     test-run:
64 |
65 |         runs-on: ubuntu-latest
66 |         needs: build
67 |
68 |         steps:
69 |         - name: Download candidate artifacts
70 |           uses: actions/download-artifact@v3
71 |           with:
72 |             name: greetings-jar
73 |
74 |         - name: Execute test
75 |           shell: bash
76 |           run: |
77 |             chmod +x ./test-script.sh
78 |             ./test-script.sh ${needs.build.outputs.artifact-tag} ${github.
79 |
80 |     create-issue-on-failure:
81 |
82 |         runs-on: ubuntu-latest
83 |         needs: test-run
84 |         if: always() && failure()
85 |         steps:
86 |         - name: invoke workflow to create issue
87 |           run: >
88 |             curl -X POST
89 |             -H "authorization: Bearer ${secrets.PIPELINE_USE}"
90 |             -H "Accept: application/vnd.github.v3+json"
91 |             "https://api.github.com/repos/${github.repository}/actions/workflows/${github.workflow}/dis
92 |             -d '{"ref":"main",
93 |               "inputs":
94 |                 {"title":"Automated workflow failure issue for commit ${github.sha}","body":"This issue was automatically created by the GitHub Action workflow
95 |                 ** ${github.workflow} **"}
96 |             }'
97 |

```

- After this is committed and the workflow runs, you can look at the output for the run and you'll see that the "create-issue-on-failure" job was skipped. That makes sense because we have the checks in the code and there was no failure on previous jobs.

feat: add create issue on failure Java CI with Gradle #38 Re-run all jobs ...

Summary

Triggered via push 7 minutes ago

Jobs	Status	Total duration	Artifacts
gwstudent pushed 6e0a6b4 main	Success	47s	1

pipeline.yml
on: push

```

graph LR
    build[build 25s] --> test-run[test-run 2s]
    test-run --> create-issue-on-failure[create-issue-on-failure 0s]

```

6. To have this executed via the "if" statement, we need to have a failure. Let's try some different input with special characters that may not print out as expected. Go to the Actions menu, and then select our main "Java CI with Gradle" workflow. Click on the "Run workflow" button and enter text like below: (that's two backslashes between the "de" and "f"). As long as you have two backslashes somewhere, this should fail.

```
" abc. de\\f ghi
```

The screenshot shows the GitHub Actions interface for the workflow "Java CI with Gradle". On the left, there's a sidebar with "Workflows" and a list of workflows including "Java CI with Gradle" and "create-failure-issue". The main area shows a list of workflow runs. The first run is "Java CI with Gradle #72: Manually run by gwstudent2" with a green status. The second run is "fix: version 3" with a green status. The third run is "fix: version2" with a red status, indicating a failure. A dropdown menu is open for the "Run workflow" button, showing options for "Use workflow from" (Branch: main), "Input Version" (1.0.1), and "Input Values" (abc de\\f ghi). A "Run workflow" button is visible at the bottom of the dropdown.

7. After the workflow run completes for this, there should be a failure in our testing. This will in turn, cause our other workflow to create an issue. You can verify the failure in testing by looking at the logs.

The screenshot shows the GitHub Actions interface for the workflow "Java CI with Gradle". The top navigation bar includes links for "Code", "Issues", "Pull requests", "Discussions", "Actions", "Projects", "Wiki", and "Security". The main area shows the workflow run "Java CI with Gradle #40" with a red status. A sidebar on the left shows a list of jobs: "build", "test-run", and "create-issue-on-failure". The "test-run" job is selected, and its logs are displayed. The logs show the following steps: "Set up job" (2s), "Download candidate artifacts" (3s), "Execute test" (1s), and "Complete job" (0s). The "Execute test" step is expanded, showing the command "Run chmod +x ./test-script.sh" and the output "abc". Below the output, there is a red error message: "Error: Process completed with exit code 1.".

<>

Code

Issues

5

Pull requests

Discussions

Actions

Projects

Wiki

Security

...

Java CI with Gradle

Java CI with Gradle #39

Re-run jobs

...

Summary

Jobs

build

test-run

create-issue-on-failure

create-issue-on-failure

succeeded 5 minutes ago in 1s

Search logs

Set up job

invoke workflow to create issue

```

1  Run curl -X POST -H "authorization: ***" -H "Accept: application/vnd.github.v3+json"
   "https://api.github.com/repos/gwstudent/greetings-ci/actions/workflows/create-failure-
   issue.yml/dispatches" -d '{"ref":"main",
2   curl -X POST -H "authorization: ***" -H "Accept: application/vnd.github.v3+json"
   "https://api.github.com/repos/gwstudent/greetings-ci/actions/workflows/create-failure-
   issue.yml/dispatches" -d '{"ref":"main",
3   "inputs":
4   {"title":"Automated workflow failure issue for commit
   3d1d1ca2a645955953cc2a6978aba577319dd2b7",
5   "body":"This issue was automatically created by the GitHub Action workflow **
   Java CI with Gradle **"}
6   },
7   shell: /usr/bin/bash -e {0}
8   % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
9   %           %         %       Dload  Upload  Total  Spent    Left     Speed
10
11    0     0    0     0     0      0     0      0  1024  --:--:-- --:--:-- --:--:--    0
12  100   251    0     0   100    251     0    1024  --:--:-- --:--:-- --:--:--  1020
13  100   251    0     0   100    251     0    1024  --:--:-- --:--:-- --:--:--  1020

```

END OF LAB

Lab 12 – Securing inputs

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

1. Switch to the pipeline.yml file in the .github/workflows directory and take a look at the "test-run" job and in particular, this line in the "Execute test" step:

```
./test-script.sh ${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion
```

```
}} ${ github.event.inputs.myValues }}
```

```
75
76 - name: Execute test
77   shell: bash
78   run: |
79     chmod +x ./test-script.sh
80     ./test-script.sh ${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }} ${ github.event.inputs.myValues }}
81
82   create-issue-on-failure:
```

2. When we create our pipelines that execute code based on generic inputs, we have to 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``

The screenshot shows the GitHub Actions interface for the 'Java CI with Gradle' workflow. The workflow is triggered by a workflow_dispatch event. The 'Run workflow' button is highlighted, and a dropdown menu is open showing the 'Input Values' field with the value '`ls -la`' entered. The workflow runs are listed below, showing a successful run for 'fix: version 3' and a failed run for 'fix: version2'.

3. After the run completes, take a look at the output of the step. Notice that it ran successfully but it has actually run the ``ls -la`` command directly on the runner system. The command was innocuous in this case, but this could have been a more destructive command.

Code Issues 6 Pull requests Actions Projects Wiki Security Insights

Java CI with Gradle Java CI with Gradle #46 Re-run all jobs

Summary

Jobs

- build
- test-run
- create-issue-on-failure

test-run

succeeded 1 hour ago in 2s

Search logs

- Set up job 1s
- Download candidate artifacts 0s
- Execute test 0s


```

1 ▶ Run chmod +x ./test-script.sh
5 total
6 16
7 drwxr-xr-x
8 drwxr-xr-x
9 drwxr-xr-x
10 3
11 3
12 3
13 runner
14 runner
15 runner
16 runner
          
```

- 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 to look like the following:

```

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

```

```

65 test-run:
66
67   runs-on: ubuntu-latest
68   needs: build
69
70   steps:
71   - name: Download candidate artifacts
72     uses: actions/download-artifact@v3
73     with:
74       name: greetings-jar
75
76   - name: Execute test
77     shell: bash
78     env:
79       ARGS: "${{ github.event.inputs.myValues }}"
80     run: |
81       chmod +x ./test-script.sh
82       ./test-script.sh "${{ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }}" "$ARGS"
83
84 create-issue-on-failure:
85
86   runs-on: ubuntu-latest
87   needs: test-run

```

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.

The screenshot shows the GitHub Actions interface for the 'Java CI with Gradle' workflow. On the left, there's a sidebar with 'Workflows' and 'All workflows'. The main area shows a list of workflow runs. The top run is 'fix: security update' (green checkmark). Below it is 'Java CI with Gradle' (green checkmark). The third run is 'Java CI with Gradle' (red X), which is the one being interacted with. A 'Run workflow' modal is open, showing the workflow name, branch (main), input version (1.0.3), and input values ('ls -la'). The 'Run workflow' button is highlighted.

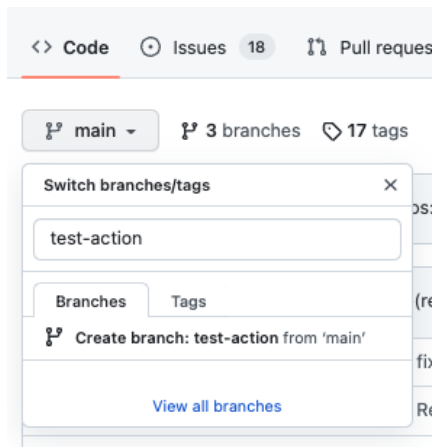
7. Notice that this time, the output did not run the commands, but just echoed them back out as desired.

END OF LAB

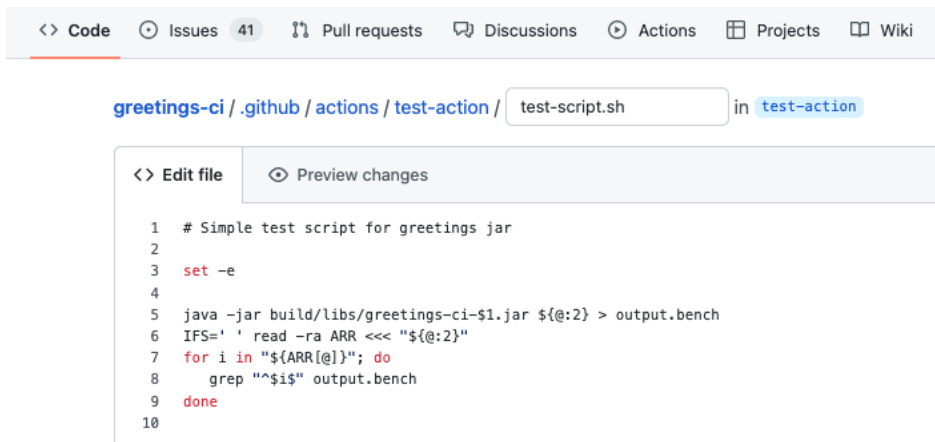
Lab 13 – Separating out jobs into a separate action

Purpose: In this lab, we'll look at how to separate our testing job into a separate action.

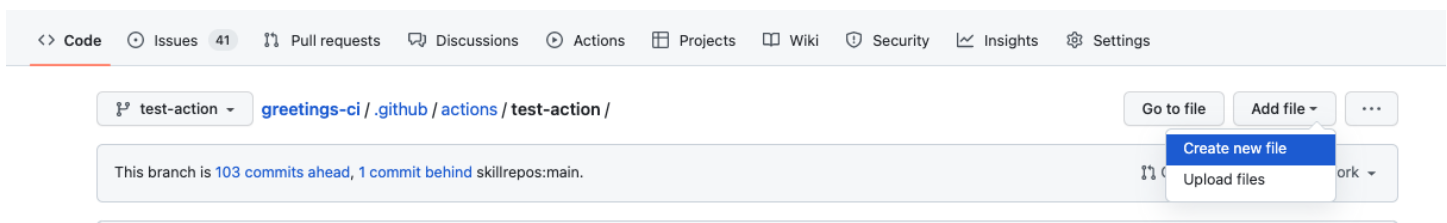
1. We're going to make our test script into a composite action. To do this, let's first create a new branch to work with called "test-action". In the "Code" tab, click on the branch dropdown that says "main". Then in the text area that says "Find or create a branch...", enter the text "test-action". Then click on the **"Create branch: test-action from 'main'"** link.



2. You should now be on the "test-action" branch. The "test-script.sh" file will be the basis for our new composite action. So, let's move it to a separate local area for this action. Select the test-script.sh file, edit it, and then add ".github/actions/test-action" to the path as shown below.



3. Click on the green button to commit your changes. Notice that no workflows were kicked off because we don't have events defined in our workflow related to the "test-action" branch.
4. Now, let's create the action.yml file for our test action. You will need to create a new file in the path "greetings-ci/.github/actions/test-action" directory by going there, clicking on "Add file" and then clicking on "Create new file"



5. Name the new file "action.yml". For the file contents, you can either copy and paste from below or from the file at <https://raw.githubusercontent.com/skillrepos/greetings-ci/main/extra/action.yml> **Commit the file when done.**

```

name: 'Test Action'
description: 'Runs a simple execution to validate compiled built deliverable'
author: 'attendee'
inputs:
  artifact-version: # semantic version of the artifact from build
    description: 'built version of artifact'
    required: true
    default: '1.0.0'
  arguments-to-print: # rest of arguments to echo out
    description: 'arguments to print out'
runs:
  using: "composite"
  steps:
    - name: Download candidate artifacts
      uses: actions/download-artifact@v3
      with:
        name: greetings-jar
        path: build/libs
    - id: test-run
      env:
        ARGS: ${{ inputs.arguments-to-print }}
      run: |
        chmod +x ${{ github.action_path }}/test-script.sh
        ${{ github.action_path }}/test-script.sh ${{ inputs.artifact-version }} "$ARGS"
      shell: bash

```

<> Code Issues 41 Pull requests Discussions Actions Projects Wiki Security ...

greetings-ci / .github / actions / test-action / action.yml

in test-action

Cancel changes

<> Edit file

Preview changes

Spaces

2

No wrap

```

1 name: 'Test Action'
2 description: 'Runs a simple execution to validate compiled built deliverable'
3 author: 'attendee'
4 inputs:
5   artifact-version: # semantic version of the artifact from build
6     description: 'built version of artifact'
7     required: true
8     default: '1.0.0'
9   arguments-to-print: # rest of arguments to echo out
10     description: 'arguments to print out'
11 runs:
12   using: "composite"
13   steps:
14     - name: Download candidate artifacts
15       id: download
16       uses: actions/download-artifact@v3
17       with:
18         name: greetings-jar
19         path: build/libs
20     - id: test-run
21       env:
22         ARGS: ${{ inputs.arguments-to-print }}
23       run: |
24         chmod +x ${{ github.action_path }}/test-script.sh
25         ${{ github.action_path }}/test-script.sh ${{ inputs.artifact-version }} "$ARGS"
26       shell: bash
27
28

```

- This is all we need for our basic composite action. Notice that we've essentially copied over a couple of steps into our composite action that were in the original workflow file. So, we can go back and modify the original workflow file to use our new action. Still in the "test-action" branch, edit the file "greetings-ci/.github/workflows/pipeline.yaml".

Replace the current steps of test-run, with the new set as shown below. Notice that we need to add a checkout action here to have the necessary pieces from our test-action directory present for the action to get to. Then we just call our new action passing in the parameters. **Commit the file when done.**

- uses: actions/checkout@v3

```

- name: run-test
  uses: ../github/actions/test-action
  with:
    artifact-version: ${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }
    arguments-to-print: ${ github.event.inputs.myValues }

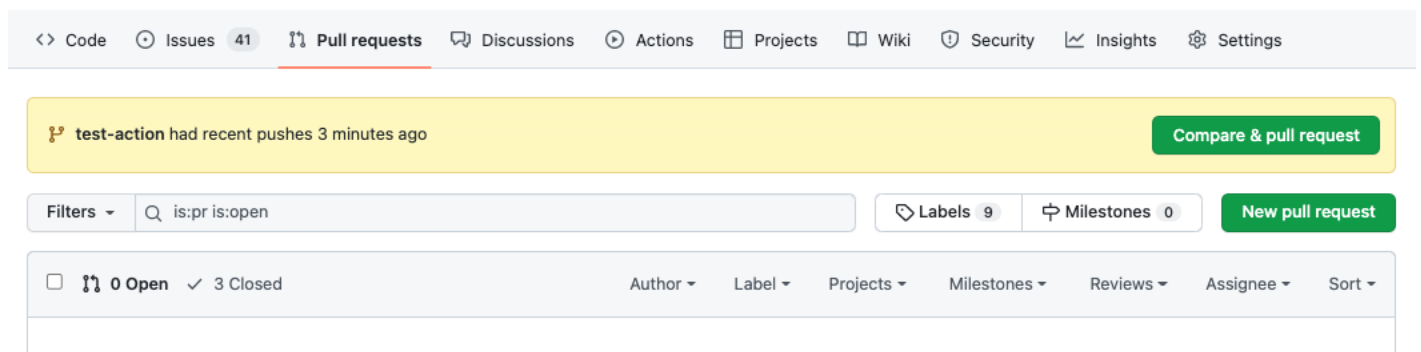
```

```

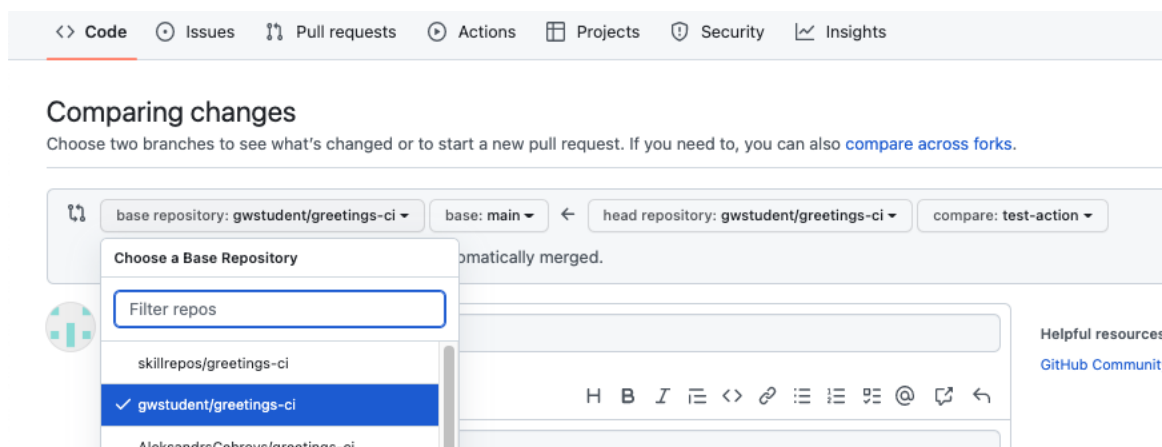
65
66   test-run:
67
68     runs-on: ubuntu-latest
69     needs: build
70
71     steps:
72
73     - uses: actions/checkout@v3
74
75     - name: run-test
76       uses: ../github/actions/test-action
77       with:
78         artifact-version: ${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }
79         arguments-to-print: ${ github.event.inputs.myValues }
80
81     create-issue-on-failure:

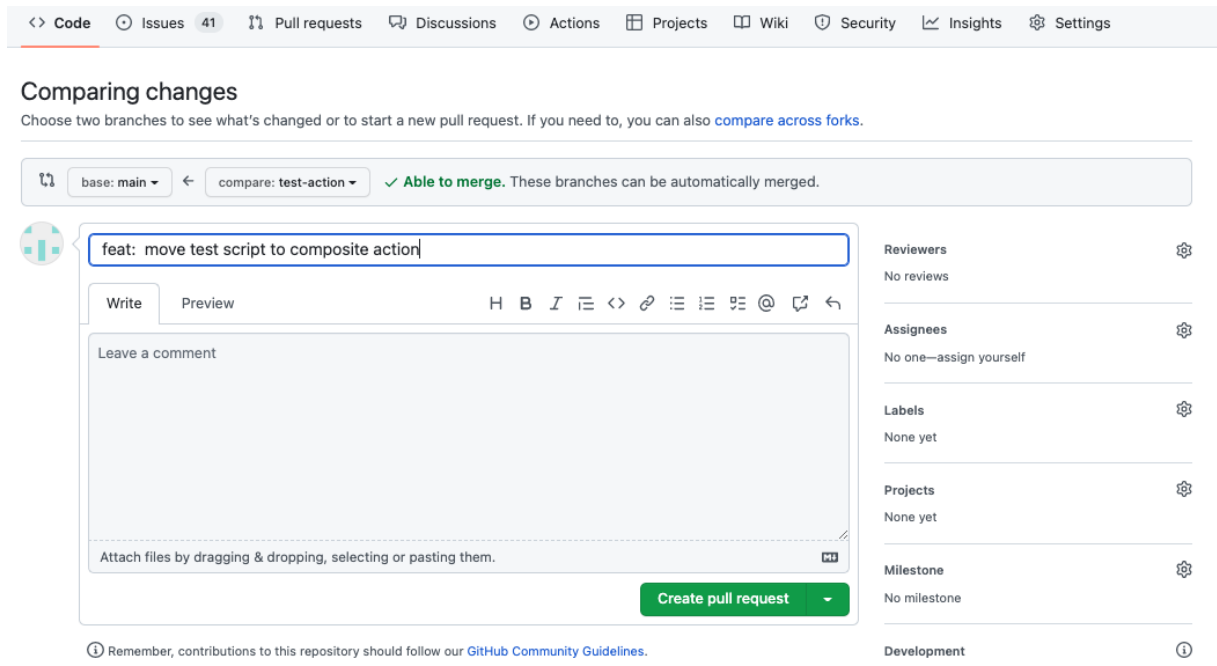
```

- Finally, let's merge in the "test-action" branch to the "main" branch. Click on the top-level "Pull requests" menu. You should see a yellow bar with text that indicates the "test-action" branch had recent pushes. Click on the green "Compare & pull request" button.

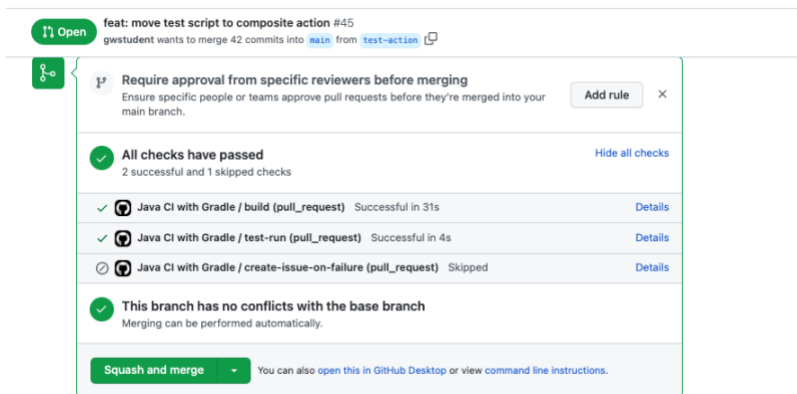


- As we've done before, change the "base" portion to be the current repo. After this, it should show that you can merge from the "test-action" branch to the "main" branch. Fill in an appropriate comment and then click the green "Create pull request" button.





9. With the Pull Request created, the automated merge checks should run and succeed. After that, you can click on the "Squash and merge" button to complete the merge. Confirm when asked. The merge should complete and the Pull Request should be closed.



10. A workflow run will have occurred as a result of the merge. But if you want to try out the merged code with the action more fully, you can do a manual workflow run as before.

END OF LAB

Lab 14 – Adding Environments and Releases

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

- 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.

- We're going to illustrate blue/green deployment with two new branches "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       myVersion:

```

4. Now, let's add the job for deploying a "stage" environment/release. This job can be inserted between the "test-run" job and the "create-issue-on-failure" job. The code for this job is already done for you and can be copied from the file <https://raw.githubusercontent.com/skillrepos/greetings-ci/main/extra/deploy-stage.txt> Just copy and paste.

This code essentially does the following:

- Waits for the build and test jobs to complete (line 79)
- Checks to see if the branch being pushed to is "blue" or "green" (line 80)
- Establishes an environment called "staging" (line 83)
- Sets the associated URL for the environment to the releases page (line 85)
- Checkouts the source code (line 87-90)
- Downloads the jar we built (line 92-95)
- Calls a GitHub Action to create a release that: (line 97-105)
 - is based on the tag we got from the build
 - is set as a draft and prerelease
 - includes the jar file we've built

```

66     steps:
67
68     - uses: actions/checkout@v3
69
70     - name: run-test
71       uses: ./github/actions/test-action
72       with:
73         artifact-version: ${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }
74         arguments-to-print: ${ github.event.inputs.myValues }
75
76
77     deploy-stage:
78
79     needs: [build, test-run]
80     if: github.ref == 'refs/heads/blue' || github.ref == 'refs/heads/green'
81
82     runs-on: ubuntu-latest
83     environment:
84       name: staging
85       url: https://github.com/${ github.repository }}/releases/tag/v${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }
86
87     steps:
88     - uses: actions/checkout@v3
89       with:
90         fetch-depth: 0
91
92     - name: Download candidate artifacts
93       uses: actions/download-artifact@v3
94       with:
95         name: greetings-jar
96
97     - name: GH Release
98       uses: softprops/action-gh-release@v0.1.14
99       with:
100         tag_name: v${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }
101         prerelease: true
102         draft: true
103
104         name: ${ github.ref_name }
105
106         files: |
107           greetings-ci-${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }.jar
108

```

- Now, let's add the job for deploying a "prod" (production) environment/release from a pull-request being merged into "main". This job can be inserted between the "deploy-stage" job and the "create-issue-on-failure" job. The code for this job is already done for you and can be copied from the file

<https://raw.githubusercontent.com/skillrepos/greetings-ci/main/extra/deploy-prod.txt> Just copy and paste.

This code essentially does the following:

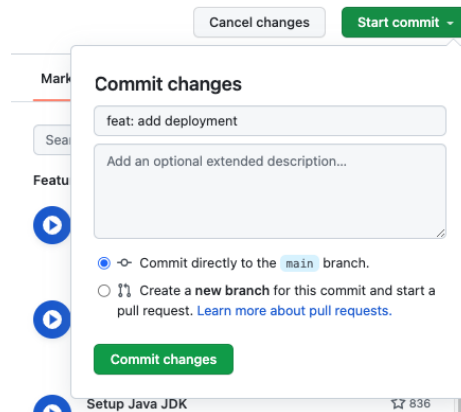
- Waits for the build and test jobs to complete (line 114)
- Checks to see if we got here on the main branch (line 115)
- Establishes an environment called "production" (line 119)
- Sets the associated URL for the environment to the releases page (line 120)
- Checkouts the source code (line 123-125)
- Downloads the jar we built (line 127-130)
- Calls a GitHub Action to create a release that: (line 132-140)
 - is based on the tag we got from the build
 - is named as "Production"
 - includes the jar file we've built and the CHANGELOG

```

105
106     files: |
107         greetings-ci-`${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }}.jar
108
109
110
111
112     deploy-prod:
113
114         needs: [build, test-run]
115         if: github.ref == 'refs/heads/main'
116
117         runs-on: ubuntu-latest
118         environment:
119             name: production
120             url: https://github.com/${ github.repository }}/releases/tag/v${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }}
121         steps:
122
123             - uses: actions/checkout@v3
124               with:
125                 fetch-depth: 0
126
127             - name: Download candidate artifacts
128               uses: actions/download-artifact@v3
129               with:
130                 name: greetings-jar
131
132             - name: GH Release
133               uses: softprops/action-gh-release@v0.1.14
134               with:
135                 tag_name: v${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }}
136                 generate_release_notes: true
137                 name: Production
138                 files: |
139                     CHANGELOG.md
140                     greetings-ci-`${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }}.jar
141
142
143     create-issue-on-failure:

```

- Go ahead and commit your changes to the main branch. You can include a "feat" conventional commit message.



- This will kick off a new run of the workflow, but it will **not** cause a deployment to happen because of our conditionals and the environments. You can see this by looking at the workflow run job graph by clicking on the most recent workflow run.

<> Code Issues 41 Pull requests Discussions Actions Projects Wiki Security Insights Settings

✓ feat: add deployment Java CI with Gradle #167

Summary

Jobs

- ✓ build
- ✓ test-run
- ⌚ deploy-stage
- ⌚ deploy-prod
- ⌚ create-issue-on-failure

Triggered via push 1 minute ago

gwstudent pushed 240b39b main

Status Success

Total duration 1m 11s

Artifacts 1

pipeline.yml

on: push

```

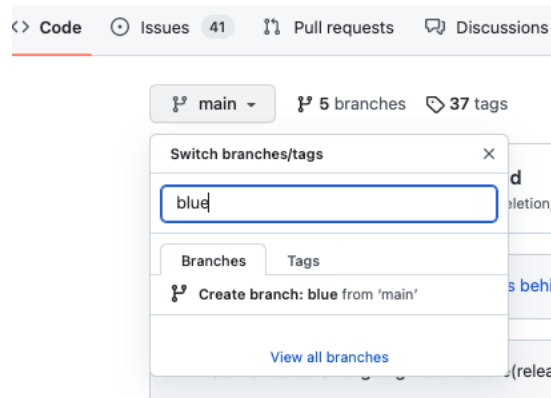
graph LR
    build[build 44s] --> test-run[test-run 4s]
    test-run --> deploy-prod[deploy-prod 0s]
    test-run --> deploy-stage[deploy-stage 0s]
    test-run --> create-issue-on-failure[create-issue-on-failure 0s]
  
```

END OF LAB

Lab 15 – 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. In 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.



2. In the "blue" branch, 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!");
}
  
```

```

1  /* Echo Message Java Program */
2  /* 8-31-21 */
3  public class echoMsg {
4      public static void main(String[] args) {
5          String value = System.getenv("USERNAME");
6          if (value != null) {
7              System.out.format("Greetings blue %s!\n", value);
8          } else {
9              System.out.println("Greetings blue!");
10         }
11     }
12     if (args.length > 0) {

```

3. Commit the changes with an appropriate "fix: " conventional commit message.

Commit changes

☒ Commit directly to the blue branch.

☐ Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

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.

fix: update to greeting for blue
 Java CI with Gradle #170

Summary

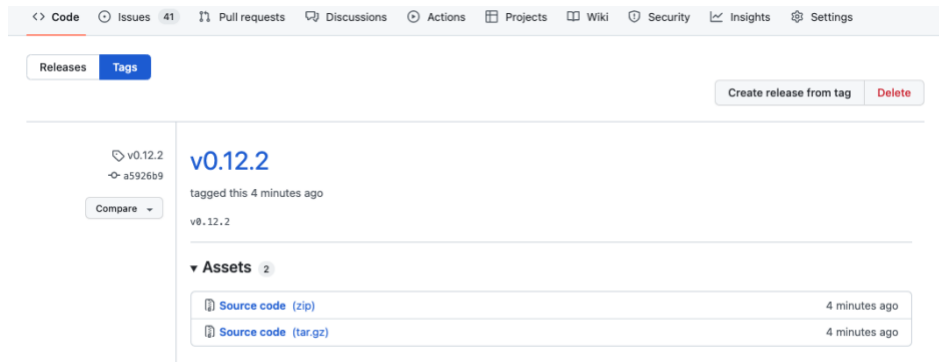
Triggered via push 2 minutes ago
 Status **Success**
 Total duration **52s**
 Artifacts **1**

gwstudent pushed -> c9a2ff0 blue

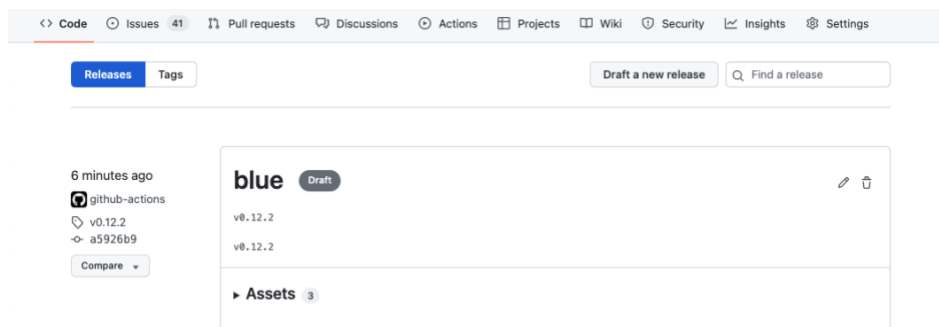
build
 test-run
 deploy-stage
 deploy-prod
 create-issue-on-failure

build 15s
 test-run 4s
 deploy-prod 0s
 deploy-stage 4s
 create-issue-on-failure 0s

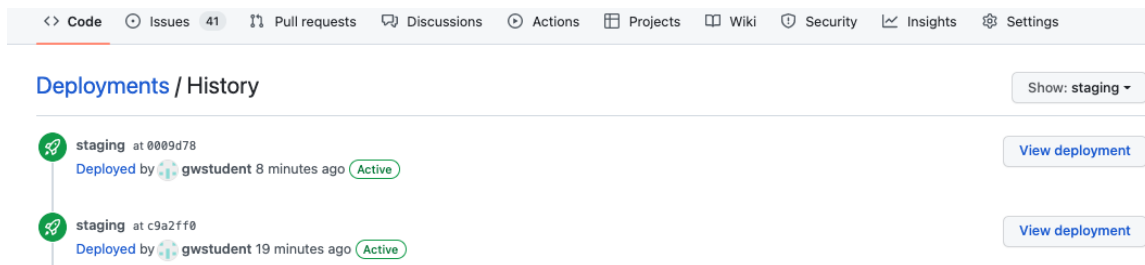
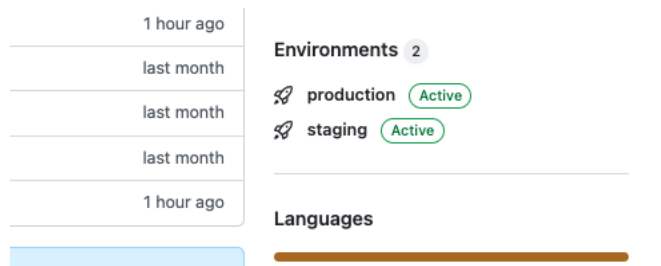
5. Now, click on the link in the "deploy-stage" box. This will take you to the tagged version of the source repo.



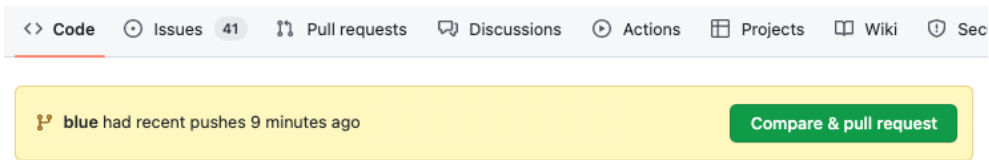
6. If you click on the "Releases" item next to "Tags", you can see the draft release that was created.



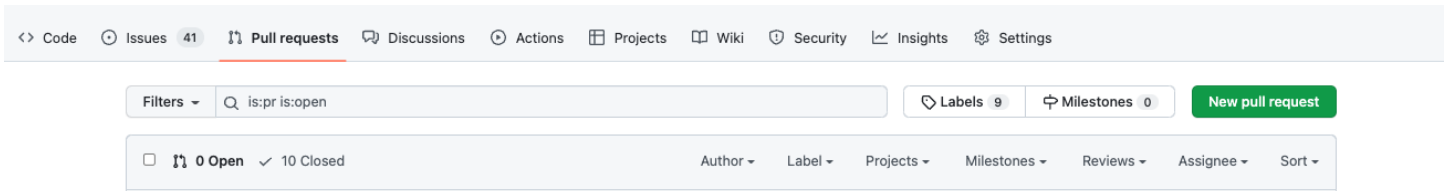
7. And, if you click on the main code page, in the lower right, you'll be able to see a new "Staging" environment. You can click on that to see a list of recent deployments there.



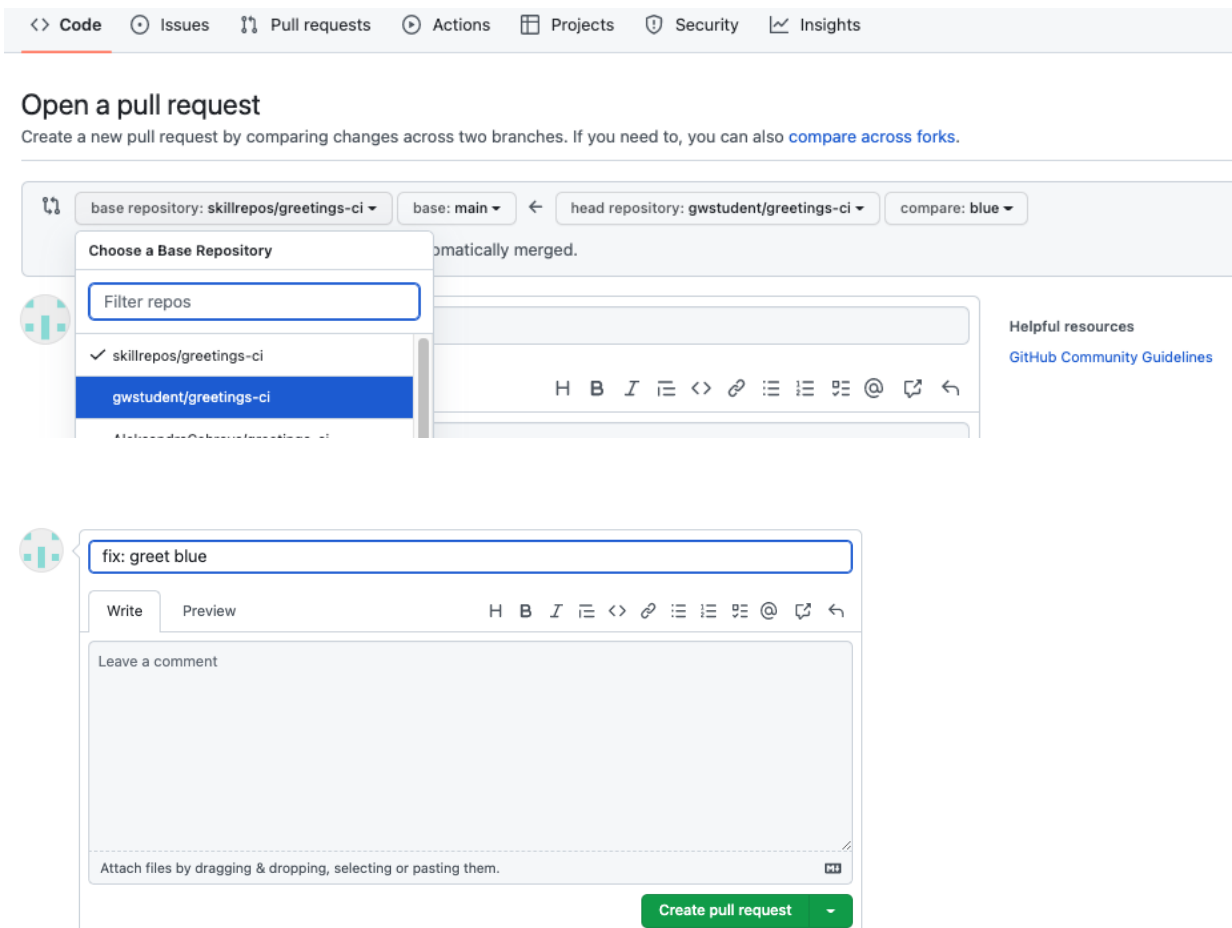
8. Since everything built ok, we can deploy this change to the production environment. To merge the changes, we can just create a pull request to main and merge it. In the "Code" page for your repository, there may be a yellow bar that says "blue had recent pushes..." If so, click on the big green "Compare & pull request" button.



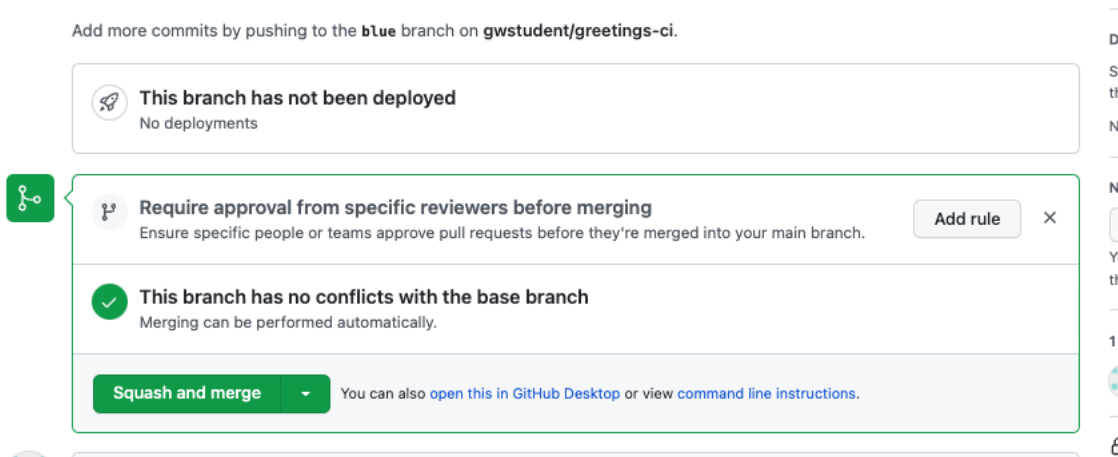
If not, got to "Pull requests" and then click on "New pull request" and "Create pull request".



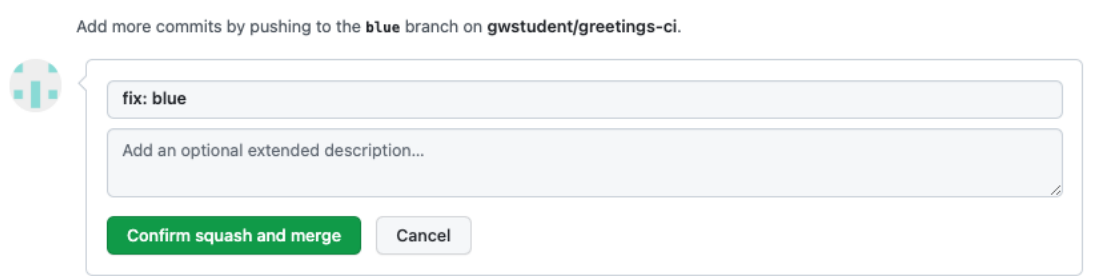
9. Change the dropdown at the top to select the same repository so you are merging the "blue" branch into the "main" branch. Add a conventional commit message like "fix: greet blue". Then proceed to "Create pull request" by clicking the other green button.



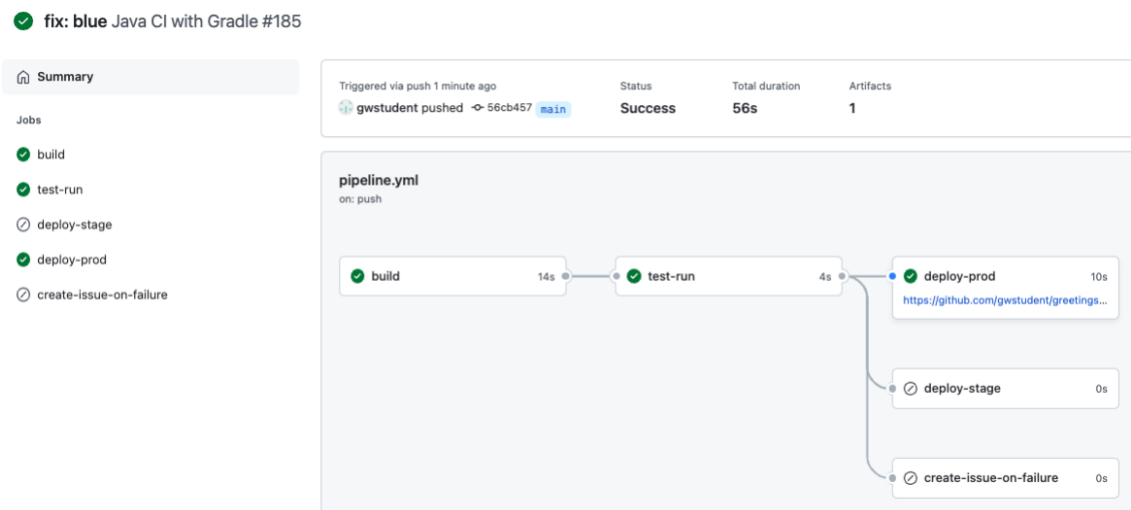
10. At this point, it will run the build and test jobs. You can go ahead and click the "Squash and merge" button when available and confirm.



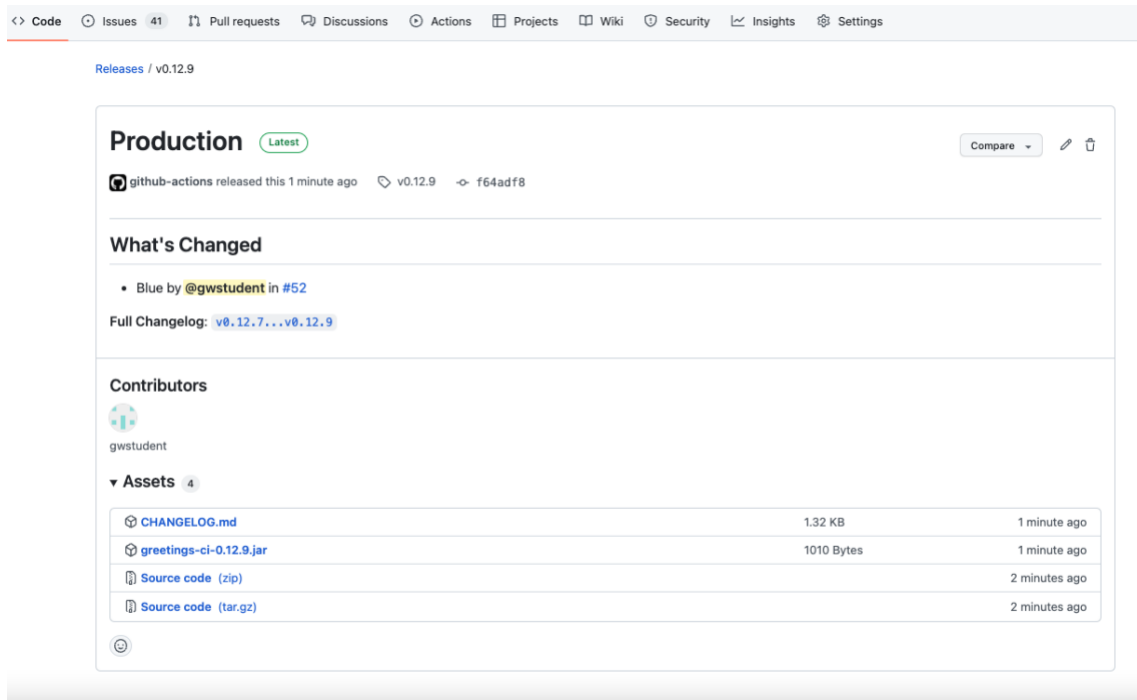
11. You can edit the main comment to have something like "fix: blue" in it and do what you want with the other commit messages. Then go ahead and click the "Confirm squash and merge" button.



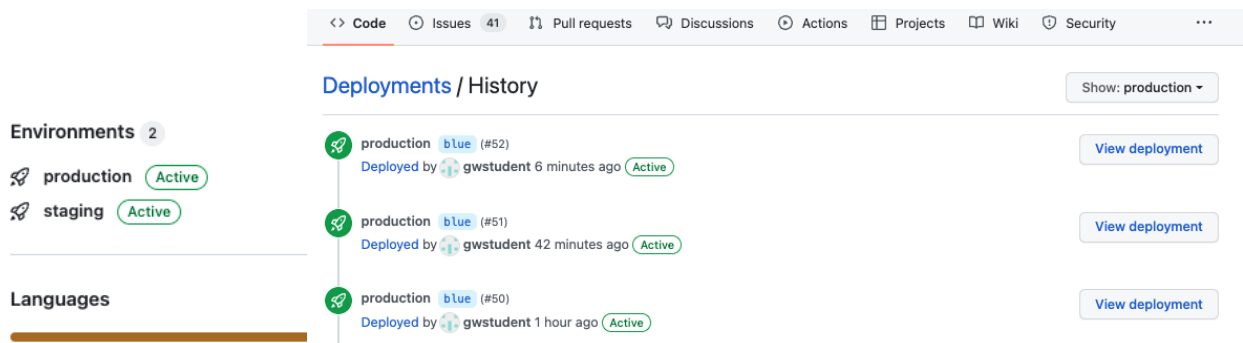
12. This should kick off another run of the action workflow in main. Because it runs in main, it should kick off the deploy-prod job.



13. After this completes, you can click on the link in the "deploy-prod" box to see the release it created.



14. You can also now see a Production environment available from the main repo page. You can click on it and see the deployments to production. Clicking on "View deployment" will take you to the same kind of page as the previous step did.



15. If you want, you can repeat the same exercise with a "green" branch to see how it works the same.

END OF LAB