

CI/CD and DevOps in 3 Weeks

Week 2

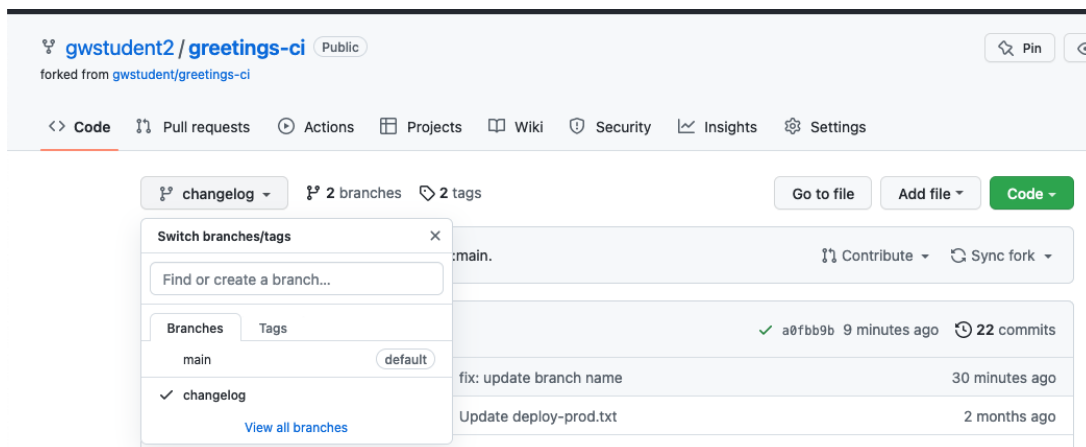
Revision 1.4 – 11/05/22

Tech Skills Transformations LLC / Brent Laster

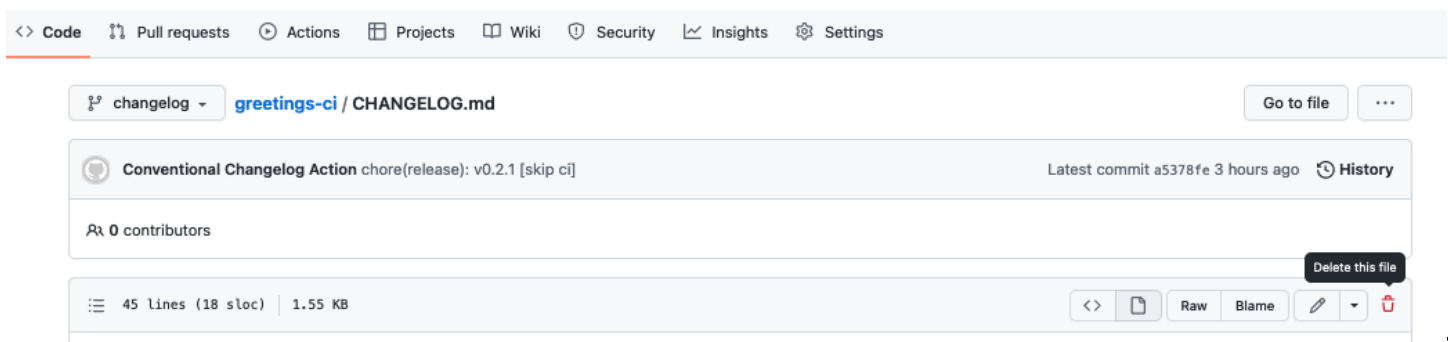
Lab 5 – Pull requests from other users

Purpose: In this lab, we'll see how to take our new code and execute a pull request from another user.

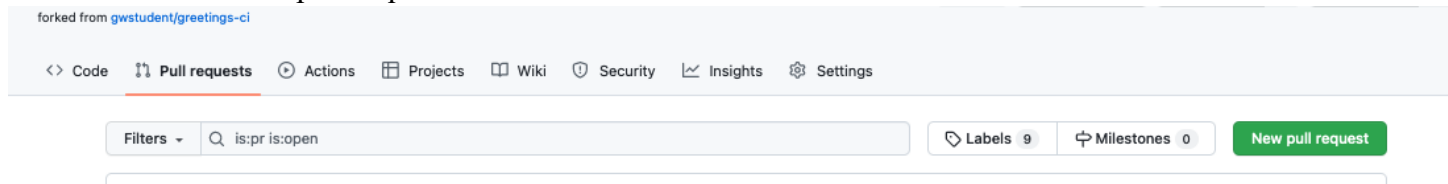
1. Starting this lab, you should be in the repository for your secondary GitHub user id - the one you added the extra actions call to in Lab 4. You also need to be on the "changelog" branch that you created in Lab 4.



2. Now that we have our code changes for the changelog in our forked repository (under the secondary userid), let's see how to get them merged back into the original repository (primary userid) via another pull request – this time between two separate repositories (meaning between the GitHub repository for our primary and secondary userids).
3. Still in the forked repository under your secondary GitHub userid and in the “changelog” branch, in preparation for the pull request, let's delete the CHANGELOG.md file so the target repo can create its own new one. Go back to the “Code” tab at the top of the repository and select the “CHANGELOG.md” file. Click on the trashcan icon on the far right in the gray bar above the text of the file. Then go ahead and commit those changes – just leave the commit message as-is.



- Also delete the package.json file via the same process.
- Still in the forked repository for your secondary GitHub userid, select the “Pull requests” tab and then click on the “New pull request” button.



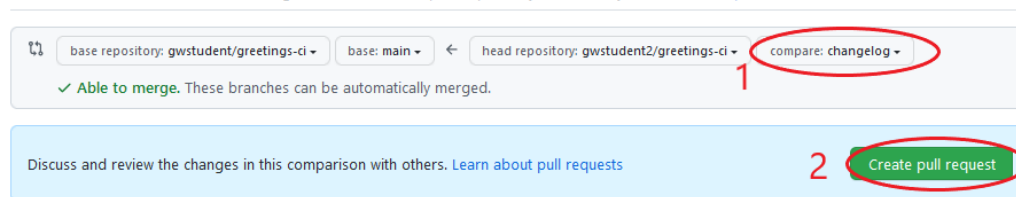
- Note that this automatically drops you into a screen where you are comparing the main branch of your forked project from the secondary GitHub userid (referred to as the “head repository”) to the original project’s (primary userid) main branch (referred to as the “base repository”).

In this case, since we made our changes in the “changelog” branch, we want to **switch the branch** in our forked project for the secondary GitHub user id (“head repository”) to be the “changelog” branch.

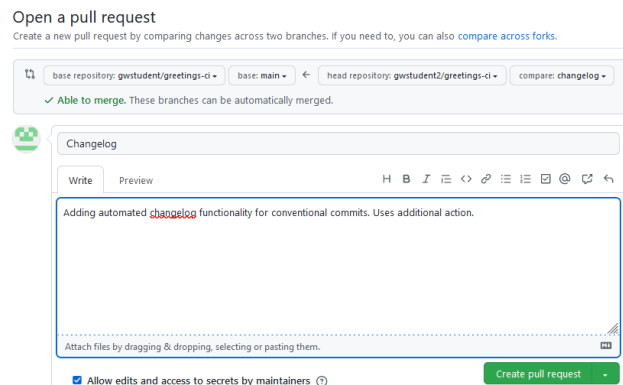
You can click on the dropdown on the far right that says “compare: main” and select “changelog” in there. Then click on the “Create pull request” button.

Comparing changes

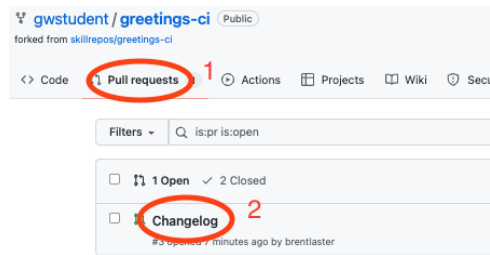
Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).



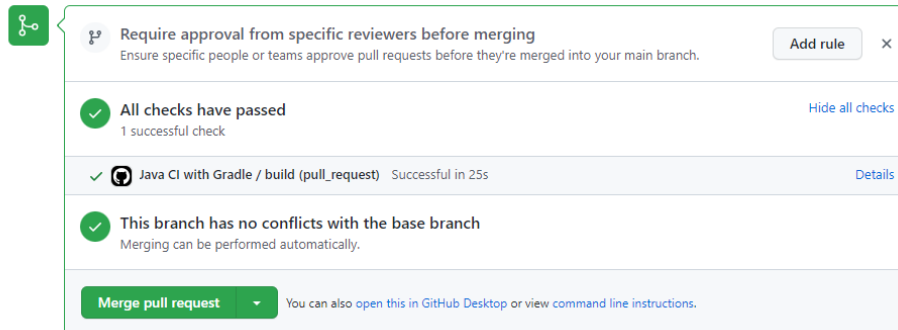
- On the next screen, you can just enter an appropriate comment and then click the next “Create pull request” button. After this, you’ll see a screen that summarizes the pull request with tabs across the top to look at the commits, checks, and files that were changed.



- Now, go back to a session under your **primary GitHub** userid (log out and log back in if needed or switch browsers, etc.). Then go to the original repository for greetings-ci under that userid. Click on the “Pull requests” tab at the top and you should see 1 open pull request from your secondary userid. Click on the link for that one.



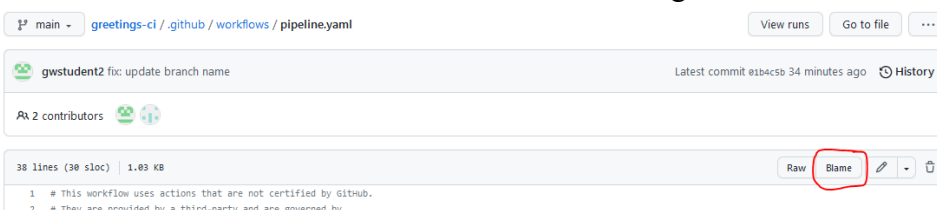
9. Now you can scroll down, and eventually you should see a message that “All checks have passed”. You can click on the “See all checks” link to the right and see that the “check” here was a run of our workflow. Then you can click on the “Merge pull request” button and then click the “Confirm merge” button and the merge request should be completed.

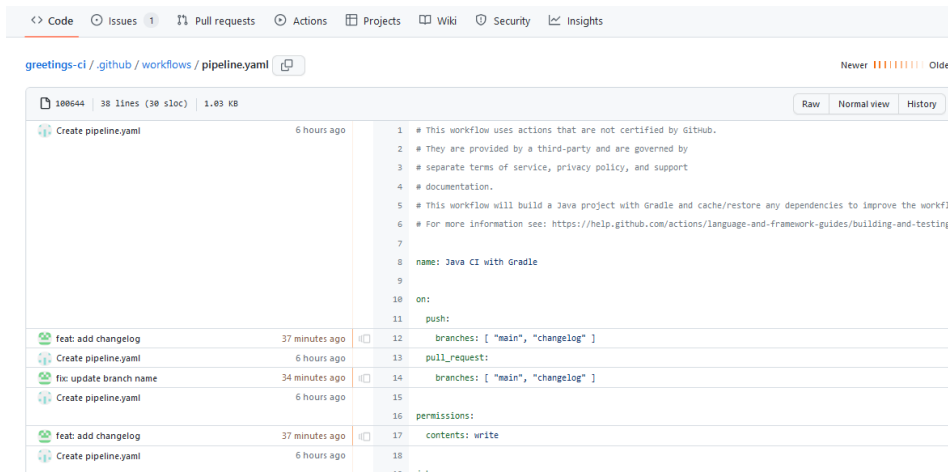


10. After the operation finishes, the workflow should have run and you should now see a CHANGELOG.md file and a package.json file in the list of files on the main branch of your original repo.

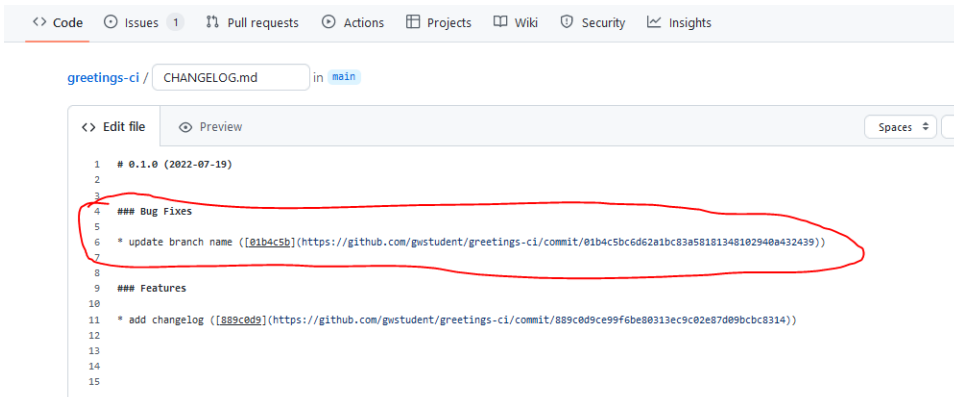
Conventional Changelog Action chore(release): v0.4.0	
.github/workflows	Update pipeline
extra	add extra dir
gradle/wrapper	Initial add
src/main/java	Initial add
CHANGELOG.md	chore(release):
build.gradle	Initial add
gradlew	Initial add
gradlew.bat	Initial add
package.json	chore(release):

11. Now, let’s take a look at who made what changes in the workflow file. Go to the workflow file (pipeline.yaml) page. Instead of clicking the pencil icon to edit, click the “Blame” button. You should see a screen like the second screenshot below showing who made what changes.

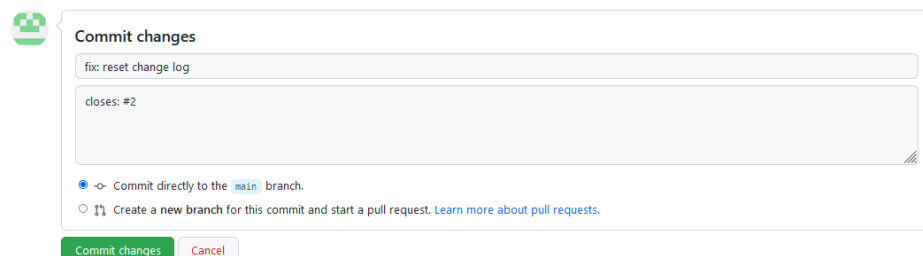




- Let's do one more fix for the repo. Click on the **Code** tab at the top in the repo. Open the CHANGELOG.md file and edit it. This is in the main branch. (Select file, pencil icon) To make it a bit cleaner for this repository, let's remove the reference to the bug fix we made in the other repository. Delete the lines shown in the red circle below.



- At the bottom of the page, in the commit message area for the “Commit changes” box, enter “fix: reset change log”. And in the “Add an optional extended description...” box, add the text “closes: #x” where “x” is the number of the issue that we opened earlier in the labs.

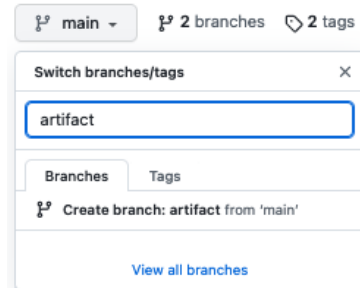


- After that, click on the “Commit changes” button. This should run the action again and close the issue.
- After this is done, look at the CHANGELOG.md file to see the fix increment and the issue (under the “Issues” tab) to see that it is closed.

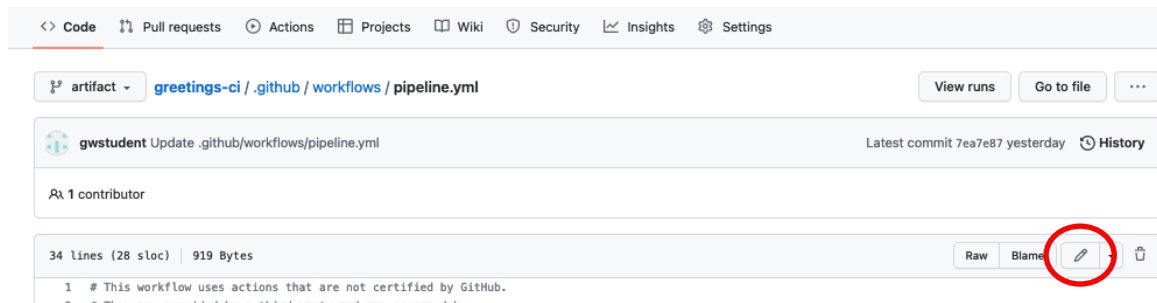
Lab 6 – Managing Artifacts

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

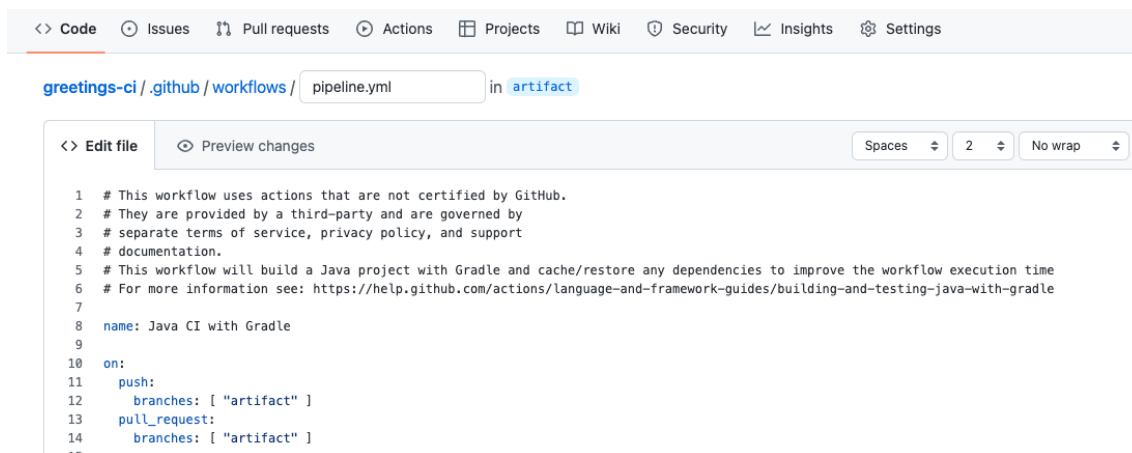
1. Log in to GitHub with your primary GitHub id and go to your “greetings-ci” project that we used in week 1.
2. As a best practice for building out the pipeline as a larger project, let's create a separate branch to work in for managing the versioning and storage of the artifact. We'll call it “artifact”. In the “Code” tab, click on the branch dropdown that says “main”. Then in the text area that says “Find or create a branch...”, enter the text “artifact”. Then click on the **“Create branch: artifact from ‘main’”** link.



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



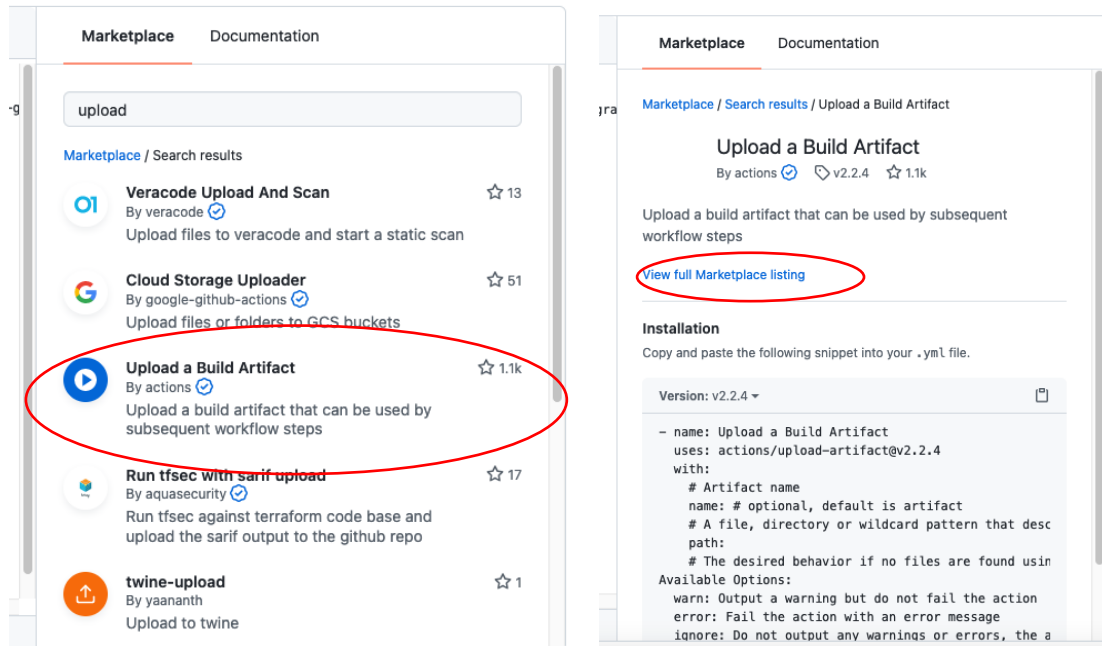
4. Change the references in the “on:” clause to be just the “artifact” branch so we don't trigger action runs on the other branches while we are working on this one. Make sure you are on the *artifact* branch before you proceed.



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

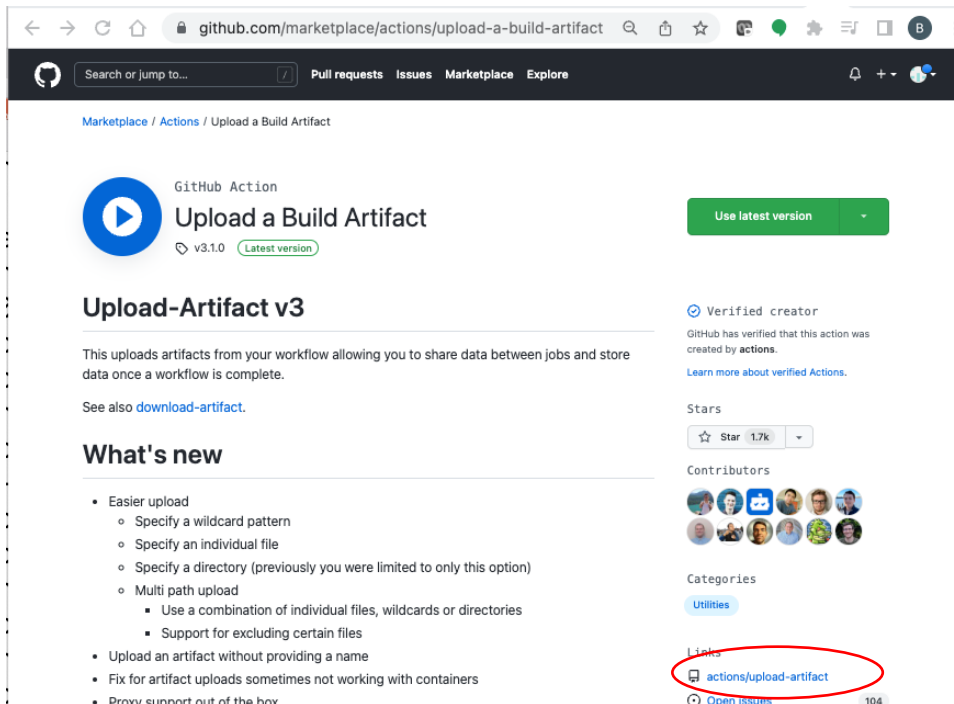
In the "Search Marketplace for Actions" box on the upper right, enter "Upload" and see what's returned.

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

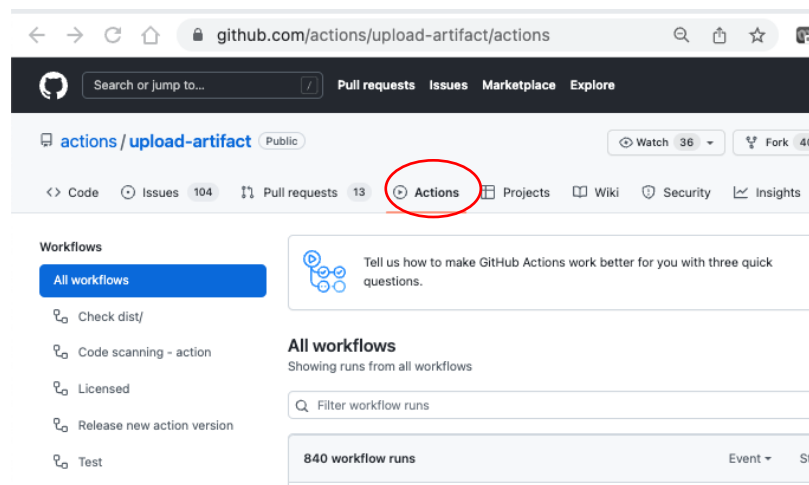


6. This should open up the full GitHub Actions Marketplace listing for this action. Notice the URL at the top - <https://github.com/marketplace/actions/upload-a-build-artifact>.

Then click on the "actions/upload-artifact" link under "Links" in the lower right.



7. 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 CI/CD themselves via GitHub Actions. Click on the Actions button to see the workflows that are in use/available



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

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

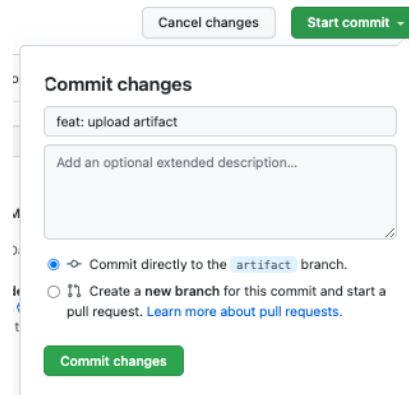
```
- name: Upload Artifact
  uses: actions/upload-artifact@v3
  with:
```

```
name: greetings-jar
path: build/libs
```

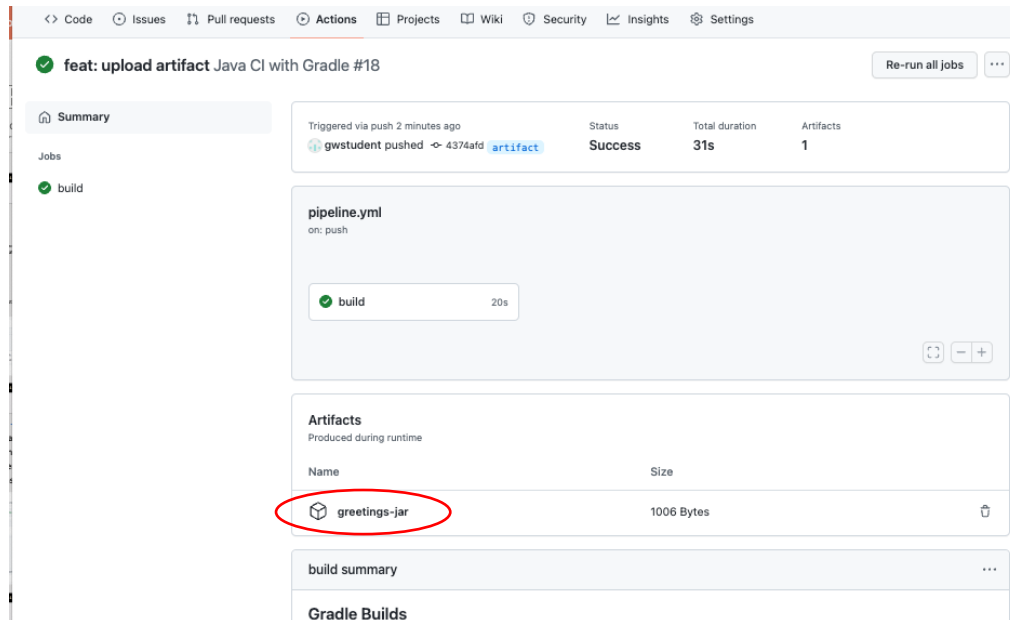
```
29
30   - name: Set up JDK 11
31     uses: actions/setup-java@v3
32     with:
33       java-version: '11'
34       distribution: 'temurin'
35   - name: Build with Gradle
36     uses: gradle/gradle-build-action@v2.2.1
37     with:
38       arguments: build
39
40   - name: Upload Artifact
41     uses: actions/upload-artifact@v3
42     with:
43       name: greetings-jar
44       path: build/libs
45
```

Use **Control** + **Space** or **Option** + **Space** to trigger auto

- Click on the green "Start commit" button in the upper right. In the dialog that comes up, add a conventional commit message like "feat: upload artifact", then click the green "Commit changes" button to make the commit.



- 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 "feat: upload artifact". On the next screen, in addition to the graph, there will be a new section called "Artifacts" around the middle of the page. You can download the artifact from there. Click on the name of the artifact to try this.



Lab 7 – Versioning Artifacts

Purpose: In this lab, we'll look at how to do simple artifact versioning to better keep track of what we produce and can use in our CI/CD processes.

1. Our artifact is being uploaded now, but we need to have each instance from a run of our pipeline clearly versioned so we can easily track changes and get back to specific versions if we need. For simplicity, we'll use the same semantic versioning scheme and value provided by our changelog generation process - specifically we'll use the "version" output from the changelog step.
 2. To reference the output from a step, we need to assign the step an "id". We can't just use the name. So, switch to the *artifact* branch and edit the pipeline.yml file again in the usual way and add the line in **bold** below in the "Conventional Changelog Action" step. See the screen capture below for a reference of where to add this. The middle line is the only one you need to add. Make sure you are on the "artifact" branch again!
- name: Conventional Changelog Action
id: changelog
 uses: [TriPSs/conventional-changelog-action@v3.14.0](#)

```

19 jobs:
20   build:
21
22     runs-on: ubuntu-latest
23
24     steps:
25       - uses: actions/checkout@v3
26
27       - name: Conventional Changelog Action
28         id: changelog
29         uses: TriPSs/conventional-changelog-action@v3.14.0
30
31       - name: Set up JDK 11
32         uses: actions/setup-java@v3

```

3. Now, we need to construct the reference to get the version output from the changelog step. This is pretty straightforward. The reference looks like this (where *changelog* is the id we added in the previous step): (We're just looking at code here, we'll make the actual changes in the next step)

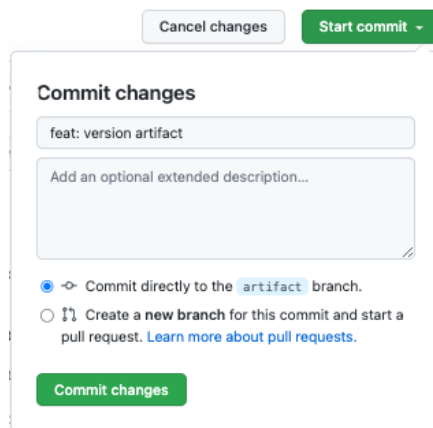
`${{ steps.changelog.outputs.version }}.jar`

4. To make this simple, we'll just add a step to tag the artifact with the version by renaming it to include the version number. We'll want to add this after the build has produced the artifact and before we upload it. Add the two lines for a new step in the code after the build step and before the upload step. (Note the second line does not need to be split in your code - it just displays that way because of the length.)

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

```
44 - name: Build with Gradle
45   uses: gradle/gradle-build-action@v2.2.1
46   with:
47     arguments: build
48
49 - name: Tag artifact
50   run: mv build/libs/greetings-ci.jar build/libs/greetings-ci-${{ steps.changelog.outputs.version || github.event.inputs.myVersion }}.jar
51
52 - name: Upload Artifact
53   uses: actions/upload-artifact@v3
54   with:
55     name: greetings-jar
56     path: |
57       build/libs
58       test-script.sh
```

5. Now we can commit this with a commit message that will trigger a new version, for example: "feat: version artifact". Go ahead and do the commit - directly to the artifact branch - with the conventional commit msg.





6. After this commit, there will be a new run of the workflow and the build job. If you select the build job in the workflow and expand the conventional changelog steps and the "Tag artifact" step, you'll be able to see the newly generated version and the rename that occurs.

build

succeeded 1 hour ago in 17s

```
83
84 New version: 0.6.0
85 /usr/bin/git add .
86 /usr/bin/git commit -m chore(release): v0.6.0 [skip ci]
87 [artifact dff35a4] chore(release): v0.6.0 [skip ci]
88 2 files changed, 19 insertions(+), 2 deletions(-)
89 /usr/bin/git tag -a v0.6.0 -m v0.6.0
90 Push all changes
91 /usr/bin/git push origin artifact --follow-tags
92 To https://github.com/gwstudent/greetings-ci.git
93 20d8f33..dff35a4 artifact -> artifact
94 * [new tag]          v0.6.0 -> v0.6.0
95
96
97
98
99
```

>  Set up JDK 11

>  Build with Gradle

▼  Tag artifact

```
1 ▼ Run mv build/libs/greetings-ci.jar build/libs/greetings-ci-0.6.0.jar
2 mv build/libs/greetings-ci.jar build/libs/greetings-ci-0.6.0.jar
3 shell: /usr/bin/bash -e {0}
4 env:
5   JAVA_HOME: /opt/hostedtoolcache/Java_Temurin-Hotspot_jdk/11.0.16-8/x64
6   GRADLE_BUILD_ACTION_SETUP_COMPLETED: true
7   GRADLE_BUILD_ACTION_CACHE_RESTORED: true
```

(Note: If you run into an issue where a new tag is not generated, you can try editing the package.json file and updating the version in it to be higher than the highest current tag on the repo.)

7. If you go to the page for the run of the action, you can see the new jar in the *Artifacts* section. You can click on it and download it and extract it to see the actual versioned artifact that was created.

fix: update package.json Java CI with Gradle #23 Re-run all jobs ...

Summary

Triggered via push 4 days ago
 gwstudent pushed → 20d8f33 artifact Success 27s 1

Jobs

build

pipeline.yml
 on: push

build 17s

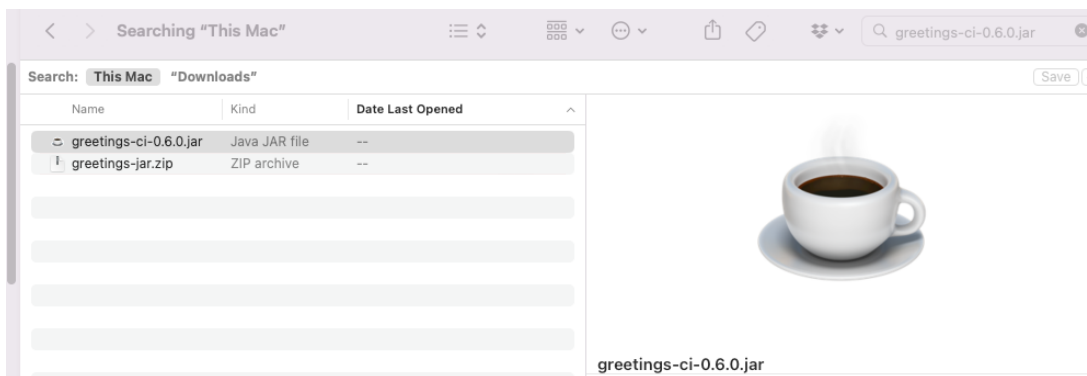
Artifacts
 Produced during runtime

Name	Size
greetings-jar	1006 Bytes

build summary

Gradle Builds

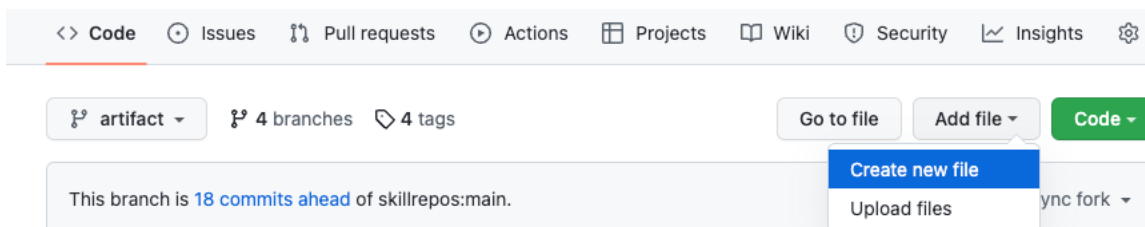
Root Project	Requested Tasks	Gradle Version	Build Outcome	Build Scan™
greetings-ci	build	4.10	✓	Build Scan™ NOT PUBLISHED



Lab 8 – Adding in a test case

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

- First, let's create a new script to test our code. To create a new file via the browser, go back to the "Code" tab at the top. You should still be in the *artifact* branch. Click on the *Add file* button next to the green *Code* button. From the list that pops up, select *Create new file*.

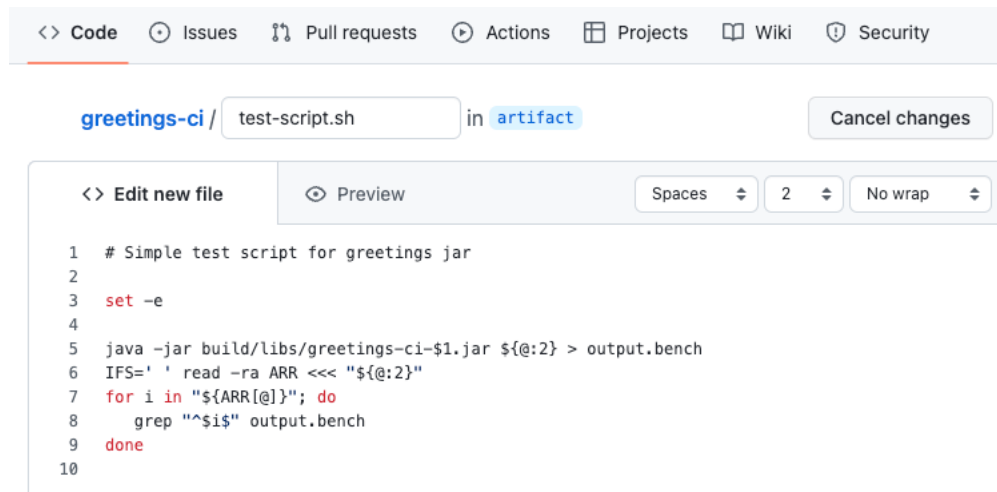


- In the new editor that pops up, you'll be at the location to type in a name. You can name this "test-script.sh". Then copy and paste the following code into the new file. (A screenshot is shown after the code so you can see how things line up.)

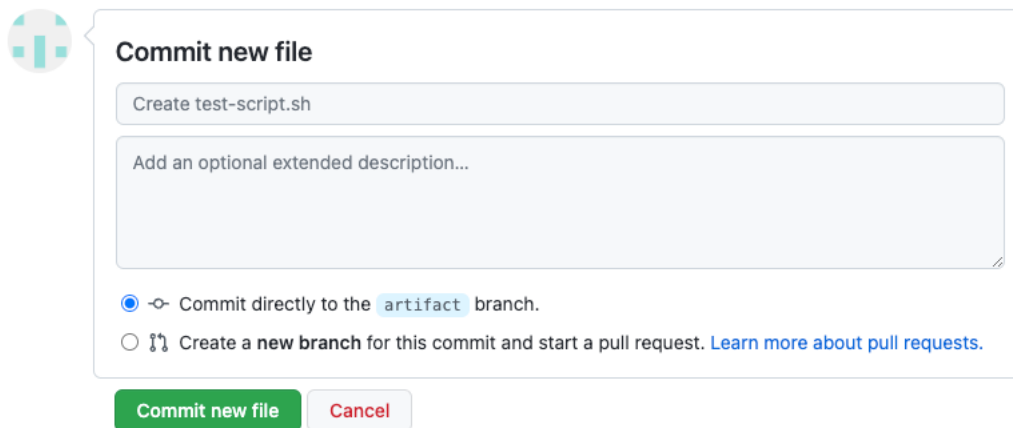
```
# Simple test script for greetings jar

set -e

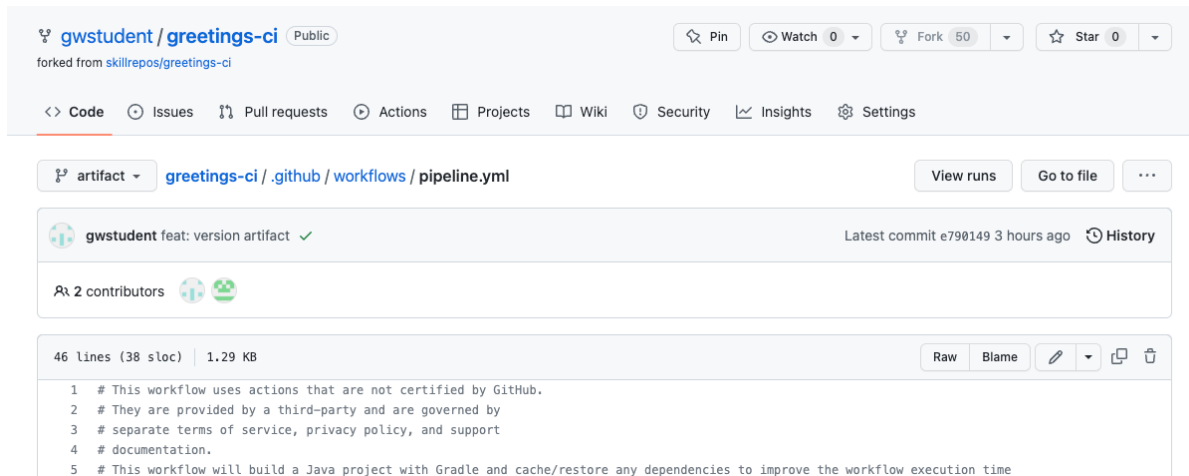
java -jar build/libs/greetings-ci-$1.jar ${@:2} > output.bench
IFS=' ' read -ra ARR <<< "${@:2}"
for i in "${ARR[@]}"; do
    grep "^$i$" output.bench
done
```



- This script takes the version of the jar to run as its first parameter and the remaining values passed in as the rest of the parameters. Then it simply cycles through all but the first parameter checking to see if they print out on a line by themselves.
- Go ahead and commit this file into the repository on the *artifacts* branch.



- Now let's add a second job to our workflow (in `pipeline.yml`) to do a simple "test". We'll continue working in the separate branch "artifacts" as we build out our pipeline. 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. 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-tag} | github.event.inputs.myVersion }} ${github.event.inputs.myValues }}
```

```

47     arguments: build
48
49   - name: Tag artifact
50     run: mv build/libs/greetings-ci.jar build/libs/greetings-ci-`${ steps.changelog.outputs.version || github.event.inputs.myVersion }`.jar
51
52   - name: Upload Artifact
53     uses: actions/upload-artifact@v3
54     with:
55       name: greetings-jar
56       path: |
57         build/libs
58         test-script.sh
59
60   test-run:
61
62     runs-on: ubuntu-latest
63     needs: build
64
65     steps:
66     - name: Download candidate artifacts
67       uses: actions/download-artifact@v3
68       with:
69         name: greetings-jar
70
71     - name: Execute test
72       shell: bash
73       run: |
74         chmod +x ./test-script.sh
75         ./test-script.sh `${ needs.build.outputs.artifact-tag || github.event.inputs.myVersion }` `${ github.event.inputs.myValues }`
76
77

```

7. There are a couple of "housekeeping" tasks we need to take care of before we call our script. First, since we want to be able to identify a specific version of the script, we need to capture the version from the "build" job into a "job output" that can be accessed from another job.

Add the lines below **in the "build:" job** definition after the "runs-on" and before the "steps". This will setup a new output from the job named artifact-tag.

Map a step output to a job output

outputs:

artifact-tag: `\${ steps.changelog.outputs.version }`

```

19 jobs:
20   build:
21
22     runs-on: ubuntu-latest
23
24     # Map a step output to a job output
25     outputs:
26       artifact-tag: `${ steps.changelog.outputs.version }`
27
28     steps:
29     - uses: actions/checkout@v3
30

```

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

items that are included in the uploading of artifacts. Modify the **path** section of the "Upload Artifact" step in the "build" job to look like below.

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

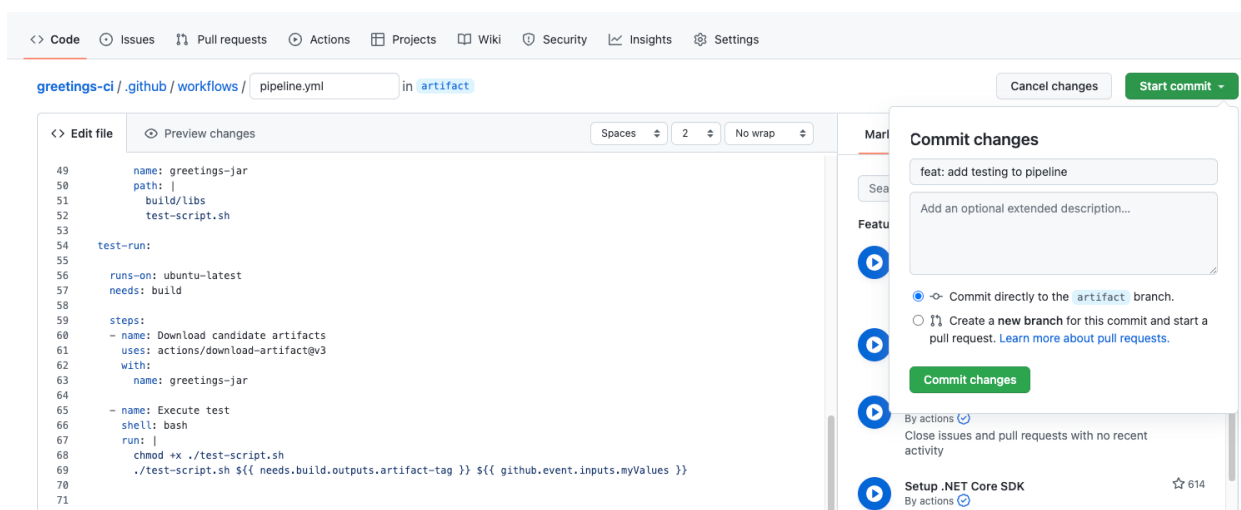
- name: Tag artifact
  run: mv build/libs/greetings-ci.jar bui

- name: Upload Artifact
  uses: actions/upload-artifact@v3
  with:
    name: greetings-jar
    path: |
      build/libs
      test-script.sh

test-run:

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

9. Now, you can just commit the pipeline changes with a simple message like "feat: add testing to pipeline".



10. 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.

The screenshot shows the GitHub Actions Summary page for a pipeline triggered by a push 3 minutes ago. The status is 'Success', the total duration is 35s, and there is 1 artifact. The pipeline is named 'pipeline.yml' and is triggered 'on: push'. The jobs listed are 'build' and 'test-run', both with green checkmarks. A visual pipeline diagram shows 'build' (14s) followed by 'test-run' (2s).

You can look at the logs from the test-run job if you want to see the downloaded script and execution. Note which version got passed in the `./test-script.sh` line. You'll need this later.

The screenshot shows the GitHub Actions interface with the 'Actions' tab selected. The workflow is 'feat: add testing to pipeline Java CI with Gradle #30'. The 'test-run' job is selected, showing it succeeded 2 minutes ago in 4s. The job steps are: 'Set up job' (1s), 'Download candidate artifacts' (1s), 'Execute test' (0s), and 'Complete job' (0s). The logs for 'Execute test' show the following commands:

```

1 Run chmod +x ./test-script.sh
2 chmod +x ./test-script.sh
3 ./test-script.sh 0.7.0
4 shell: /usr/bin/bash --noprofile --norc -e -o pipefail {0}

```

- Now that we've proven out the changes in the *artifact* branch, let's go ahead and merge the changes back into the main branch via a pull request. In the *Pull requests* menu at the top, click on the *New pull request* button. Change the base value to be for **your repo** and **main** (not skillrepos/greetings-ci) and the compare value to be for **your repo** and **artifact**. Then click on the *Create pull request* button.

On the next dialog enter a conventional commit message like "chore: merge artifact to main", then click the 2nd *Create pull request* button.

On the third dialog, all of the checks should be green, so just go ahead and select *Merge pull request*, then *Confirm merge*.

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

base: main
←
compare: artifact
✓ Able to merge. These branches can be automatically merged.

Discuss and review the changes in this comparison with others. [Learn about pull requests](#)

Create pull request

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base: main
←
compare: artifact
✓ Able to merge. These branches can be automatically merged.

chore: merge artifact to main

Write

Preview

H

B

I

≡

<>

↻

⋮

⋮

⋮

@

↗

↶

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Create pull request

Add more commits by pushing to the **artifact** branch on **gwstudent/greetings-ci**.

Require approval from specific reviewers before merging

[Branch protection rules](#) ensure specific people approve pull requests before they're merged.

Add rule

×

✓

All checks have passed

2 successful checks

Show all checks

✓

This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

- Let's make one more change to make it easier to run our workflow manually to try things out, start runs, etc. **Switch to the main branch.** Then edit the pipeline.yaml file again. In the "on:" section near the top, change the branch names from "artifact" to "main" And 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.

© 2022 Tech Skills Transformations, LLC & Brent Laster

Brent Laster

Page 18

```

workflow_dispatch:
  inputs:
    myVersion:
      description: 'Input Version'
    myValues:
      description: 'Input Values'

```

Note: Don't forget to change the branch names!

```

81 lines (62 sloc) | 2.06 KB
1  # This workflow uses actions that are not certified by
2  # They are provided by a third-party and are governed b
3  # separate terms of service, privacy policy, and suppor
4  # documentation.
5  # This workflow will build a Java project with Gradle a
6  # For more information see: https://docs.github.com/en/
7
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

```

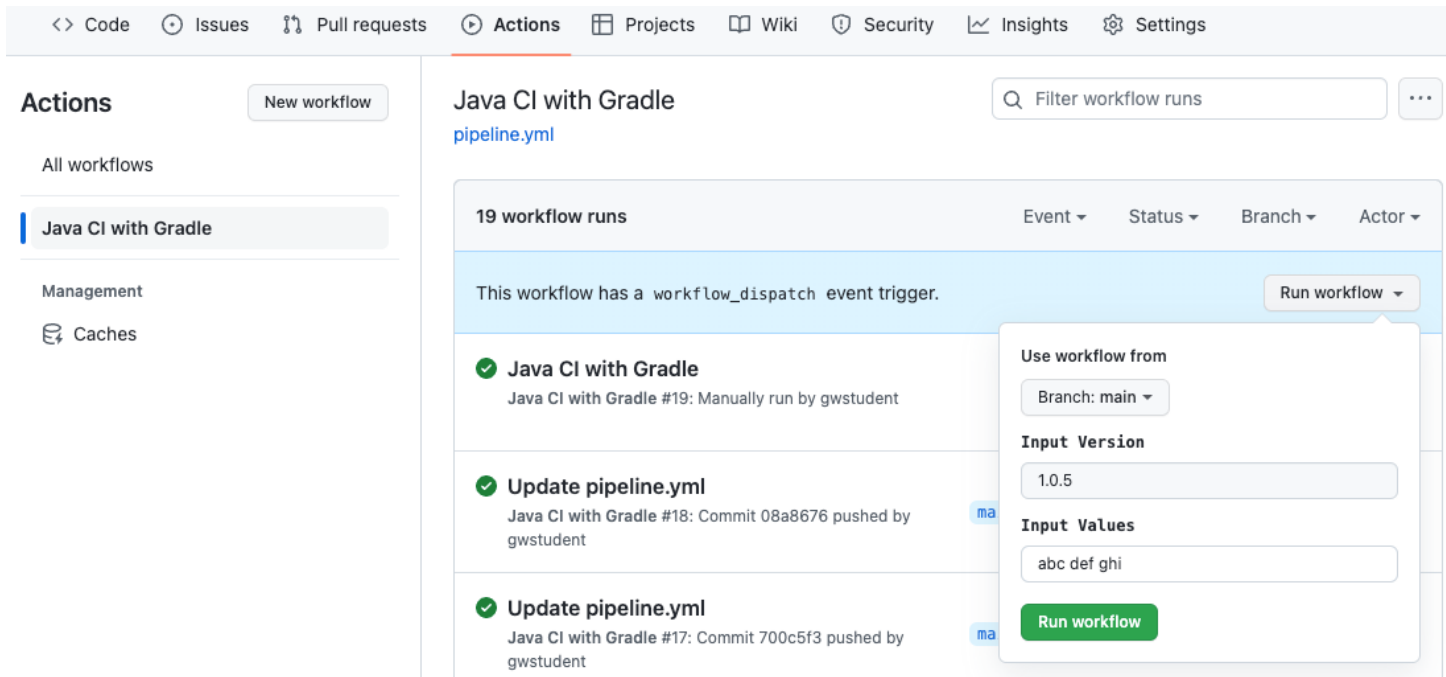
13. Commit your changes to the main branch.
14. Since we added the *workflow_dispatch* event trigger in our last set of code changes, if you select the workflow, you should see a blue bar that has a button that allows you to manually run the workflow.

Click on *Run workflow*. Make sure *Branch* is set to *main*.

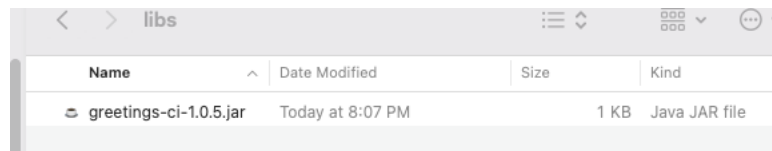
In the *Input Version* section, put in a semantic version number such as "1.0.5".

In the *Input Values* section, enter whatever you want for arguments.

Click on the green "Run workflow" button. You can look at the output of the test-run job again to see what occurred.



15. If you want, you can download the artifact, extract the contents and see the versioned jar.



Homework: Get a GitHub Personal Access Token and Store it as a Secret

1. For next week, you will 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.

GitHub Apps

OAuth Apps

Personal access tokens

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

workflows

What's this token for?

Expiration *

30 days

The token will expire on Tue, Oct 5 2021

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<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

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

tokens

Generate new token

erated that can be used to access the [GitHub API](#).

your personal access token now. You won't be able to see it again!

tQwNXWU9u2mXYJNVa3Ksu4Iz0iG

- Now 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.

Code

Issues

Pull requests

Actions

Projects

Wiki

Security

Insights

Settings

General

Access

Collaborators

Moderation options

Code and automation

Branches

Tags

Actions

Webhooks

Environments

Pages

Security

Code security and analysis

Deploy keys

Secrets

Actions

Actions secrets

Secrets are environment variables that are **encrypted**. Anyone with **collaborator** access to this repository can use these secrets for Actions.

Secrets are not passed to workflows that are triggered by a pull request from a fork. [Learn more](#).

Environment secrets

There are no secrets for this repository's environments.

Encrypted environment secrets allow you to store sensitive information, such as access tokens, in your repository environments.

[Manage your environments and add environment secrets](#)

Repository secrets

There are no secrets for this repository.

Encrypted secrets allow you to store sensitive information, such as access tokens, in your repository.

- 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.

General

Access

Collaborators

Moderation options

Code and automation

Branches

Tags

Actions

Webhooks

Environments

Pages

Security

Actions secrets

Name

PIPELINE_USE

Value

ghp_ctQHmpueXJG0SAY3r

Add secret

General

Access

Collaborators

Moderation options

Code and automation

Branches

Tags

Actions

Webhooks

Environments

Pages

Security

Code security and analysis

Deploy keys

Secrets

Actions

Dependabot

Actions secrets

Secrets are environment variables these secrets for Actions.

Secrets are not passed to workflow

Environment secrets

There are
Encrypted environment se

Ma

Repository secrets

PIPELINE_USE