### **CI/CD and DevOps in 3 Weeks**

**Week 3**

**Revision 2.3 – 02/04/24**

Tech Skills Transformations LLC / Brent Laster

**HELPFUL HINT:** For some of the labs, you will need to copy code from a file in the "extra" subdirectory of the *greetings-ci* project. To make it easy to copy the code into the editor, you can open a second tab in your browser to the same greetings-ci repository, and bring the file up in the "extra" subdirectory in that second tab. Then you can just copy and paste from the second tab into where you are editing on the first one.

**Lab 9: Working with fast feedback and automatically reporting issues**

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

(Note: Code for steps 2 and 5 is also in this gist: <https://gist.github.com/brentlaster/5372bd941e3186a39bd2685c748c1447> )

1. Start out in the **"greetings-ci" repository** in GitHub under your **primary GitHub** userid, and in the **main** branch. 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.

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.

1. In the repository, browse to the "extra" folder and to the "create-failure-issue.yml" file.
2. Take a few moments to look over the file and see what it does. Notice that:
   1. It has a workflow\_call section in the "on" area, which means it can be run from another workflow.
   2. It has a workflow\_dispatch section in the "on" area, which means it can be run manually.
   3. It has two inputs - a title and body for the issue.
   4. The primary part of the body is simply a REST call (using the GITHUB\_TOKEN) to create a new issue.
3. Click the pencil icon to edit it.



*  Graphical user interface, text, application, email

  Description automatically generated

1. 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".

Graphical user interface, text, application, email

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

3. (Ignore if you get a 404 error.) 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".

Graphical user interface, text, application, Teams

Description automatically generated

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.

Graphical user interface, text, application, email, website

Description automatically generated

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.). Commit your changes when done.

create-issue-on-failure:

needs: test-run

permissions:

issues: write

if: always() && failure()

uses: ./.github/workflows/create-failure-issue.yml

with:

title: "Automated workflow failure issue for commit ${{ github.sha }}"

body: "This issue was automatically created by the GitHub Action workflow \*\* ${{ github.workflow }} \*\*"

A screenshot of a computer

Description automatically generated

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**

Graphical user interface, text, application

Description automatically generated

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.

Graphical user interface, text, application

Description automatically generated

8. And you can see the run of the reusable workflow if you select the "create-issue-on-failure" job.

A screenshot of a computer

Description automatically generated with medium confidence

9. You can also verify the new issue got created as a result of the failure through the logs of that job and by looking in the Issues menu at the top.

Graphical user interface, application

Description automatically generated

END OF LAB

**Lab 10 – 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 look at the "test-run" job and in particular, this line in the "Execute test" step:

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

A blue and white website

Description automatically generated with medium confidence

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, go back to the Actions tab, select the “Java CI with Gradle” workflow, and, using the workflow\_dispatch event for the workflow, put in a version and pass in the following as the arguments in the “Input Values” field (NOTE: That is two backquotes around ls -la) `ls -la`

Graphical user interface, text, application, email

Description automatically generated

3. After the run completes, click through the run to the job to the log and look at the output of the “test-run” 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.

A screenshot of a computer

Description automatically generated

Graphical user interface, text

Description automatically generated

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 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 ${{ github.event.inputs.myVersion || needs.build.outputs.artifact-tag }} "$ARGS"**

A screenshot of a computer program

Description automatically generated

 5. Commit back the changes and wait till the action that's run for the push has completed.

6. Now, you can execute the code again with the same arguments as before. (Actions menu and select workflow.)

Graphical user interface, text, application, email

Description automatically generated

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

END OF LAB

**Lab 11 – 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, lets 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.

Graphical user interface, text, application

Description automatically generated

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 local actions. Select the test-script.sh file, edit it, and then click in the name path area and add "**.github/actions/test-action**" to the path as shown below.

Graphical user interface, text, application, email

Description automatically generated

3. Click on the green button to commit your changes in the "test-action" branch. 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 to allow our test action to be used as an actual action. The file is already in the “extra” directory. You can just open it and rename it as in the last step. Select the file “extra/action.yml”. Take a look at the contents to try and understand it. Then edit it and rename as you did in the previous step – adding “**.github/actions/test-action**” to the path.

A screenshot of a computer

Description automatically generated

5. Commit your changes via the “Commit changes…” button.

6. This is all we need for our basic composite action. Notice that we've essentially copied over a couple of steps (line 13 on) 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".**

7. Replace the current steps of test-run, with the new set as shown below (checking alignment). 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.

**- uses: actions/checkout@v3**

**- name: run-test**

**uses: ./.github/actions/test-action**

**with:**

**artifact-version: ${{ github.event.inputs.myVersion || needs.build.outputs.artifact-tag }}**

**arguments-to-print: ${{ github.event.inputs.myValues }}**

A screenshot of a computer program

Description automatically generated

7. We need to make one code update in our workflow file due to an issue with the action that can cause the pull request merge to fail. Temporarily, we will add a parameter to the changelog action to tell it not to do a pull since that causes issue when trying to do it on a pull request. Edit the .github/workflows/pipeline.yml file and add the two lines in bold as shown below as a parameter for the action. Commit the change to the test-action branch when done.

  - name: Conventional Changelog Action

      uses: TriPSs/conventional-changelog-action@v4

**with:**

**skip-git-pull: "true"**

A screen shot of a computer program

Description automatically generated

8. 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.

Graphical user interface, application, website

Description automatically generated

9. As we've done before, **change the "base" portion to be the current repo (not skillrepos).** After this, it should show that you can merge from the "test-action" branch to the "main" branch of your "greetings-ci" repository. Fill in an appropriate comment starting with “feat:” and then **click the green "Create pull request" button.**

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

10. 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. (Select "Squash and merge" from the down arrow on the right of the green button if needed.). Confirm when asked. The merge should complete, and the Pull Request should be closed.

Graphical user interface, text, application, email

Description automatically generated

11. After the operation finishes, the workflow should have run. Now, we need to remove the extra lines we added for the changelog action. Remove the lines shown in strikethru below while editing the file pipeline.yml in the main branch and then committing the change.

- name: Conventional Changelog Action

      uses: TriPSs/conventional-changelog-action@v5.1.0

**~~with:~~**

**~~skip-git-pull: "true"~~**

12. (Optional) Workflow runs will have occurred because of the changes. But if you want to try out the merged code with the action more fully, you can do a manual workflow run as before.

Graphical user interface, text, application, email

Description automatically generated

END OF LAB

**Lab 12 – Adding Environments and Releases**

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

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

**Graphical user interface, text, application, email

Description automatically generated**

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

**on:**

**push:**

**branches: [ "main", "blue", "green" ]**

Graphical user interface, text, application

Description automatically generated

3. You can also remove the "pull\_request" portion.

Graphical user interface

Description automatically generated with low confidence

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 [extra/deploy-stage.txt](https://raw.githubusercontent.com/skillrepos/greetings-ci/main/extra/deploy-stage.txt) You can go to the same repository in another tab, open that file, and then just copy and paste. (Note double-check that the "create-issue-on-failure" job name didn't get moved to the end of the paste.)

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

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

Graphical user interface, application

Description automatically generated

5. 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 [extra/deploy-prod.txt](https://raw.githubusercontent.com/skillrepos/greetings-ci/main/extra/deploy-prod.txt) Just copy and paste. You can copy and paste this one the same way from the other tab.

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

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

Application

Description automatically generated with low confidence

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

Graphical user interface, text, application, chat or text message

Description automatically generated

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

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

END OF LAB

**Lab 13 – 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. (Note that you will need to make sure you're on the main branch before creating the blue one.)

Graphical user interface, text, application

Description automatically generated

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!");**

**}**

Graphical user interface, text, application

Description automatically generated

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

Graphical user interface, text, application, email, Teams

Description automatically generated

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.

Graphical user interface, application

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

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

Graphical user interface, text, application, email, Teams

Description automatically generated

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.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

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.

Graphical user interface, application

Description automatically generated

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

Graphical user interface, text, application, website

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

10. At this point, you can go ahead and click the "Squash and merge" button when available and confirm.

Graphical user interface, text, application, Teams

Description automatically generated

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.

Graphical user interface, text, application, email, Teams

Description automatically generated

12. This should kick off another run of the action workflow in main.

END OF LAB