### **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".)

1. 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.

Graphical user interface, text, application, email

Description automatically generated

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

![Graphical user interface, text, application, email

Description automatically generated]()

1. 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.

Graphical user interface, text, application, email

Description automatically generated

1. 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.

Graphical user interface, text, application

Description automatically generated Graphical user interface, application

Description automatically generated

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'**

**Text, timeline

Description automatically generated**

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

Graphical user interface, text, application

Description automatically generated

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 }}

**A picture containing text

Description automatically generated**

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.

Graphical user interface, text, application, email

Description automatically generated

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.

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\_dispatch section in the "on" area, which means it can be run manually.
   2. It has two inputs - a title and body for the issue.
   3. 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, Teams

  Description automatically generated]()

1. 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".

Graphical user interface, text, application

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

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".

Graphical user interface, text, application, Teams

Description automatically generated

3. 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

Description automatically generated

4. 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 }} \*\*"}

}'

A picture containing graphical user interface

Description automatically generated

5. 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.

Graphical user interface, application

Description automatically generated

1. 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

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, text, application

Description automatically generated

Graphical user interface, application

Description automatically generated

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 }}**

Text

Description automatically generated with medium confidence

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`

Graphical user interface, text, application, email

Description automatically generated

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.

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:

**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"**

Graphical user interface, text, application, email

Description automatically generated

 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.

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 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, 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 this action. Select the test-script.sh file, edit it, and then 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. 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"

Graphical user interface, application, Teams

Description automatically generated

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**

Graphical user interface, text, application, email

Description automatically generated

6. 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 }}

Graphical user interface, text, application

Description automatically generated

7. 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

8. 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.

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

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.

Graphical user interface, text, application, email

Description automatically generated

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.

Graphical user interface, text, application, email

Description automatically generated

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.**

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'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" ]

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 <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

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

<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

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, 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.

Graphical user interface, text, application

Description automatically generated

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.

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. 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, email

Description automatically generated

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.

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. Because it runs in main, it should kick off the deploy-prod job.

Graphical user interface, application

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

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.

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

Description automatically generatedGraphical user interface, text, application, chat or text message

Description automatically generated

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

END OF LAB