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

**Week 2**

**Revision 1.2 – 08/14/22**

Tech Skills Transformations LLC / Brent Laster

**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.

Graphical user interface, text, application

Description automatically generated

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.yaml file (click on the name) and edit it by clicking on the pencil icon.

Graphical user interface, text, application, Teams

Description automatically generated

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.

Graphical user interface, text, application, email

Description automatically generated

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.

Next, click on the "Upload a Build Artifact" item. Take a look at the page that comes up from that. Let's look at the full listing on the Actions Marketplace. Click on the "View full Marketplace listing".

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

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.

Graphical user interface, text, application

Description automatically generated

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

Graphical user interface, website

Description automatically generated

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.)

- name: Upload Artifact

uses: actions/upload-artifact@v3

with:

name: greetings-jar

path: build/libs

Table

Description automatically generated with low confidence

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

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

Description automatically generated

10. 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.

Graphical user interface, text, application, email

Description automatically generated

**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, edit the pipeline.yaml 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. **Make sure you are on the "artifact" branch again!**

- name: Conventional Changelog Action

**id: changelog**

uses: [TriPSs/conventional-changelog-action@v3.14.0](mailto:TriPSs/conventional-changelog-action@v3.14.0)

Graphical user interface, text, application

Description automatically generated with medium confidence

1. 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**

1. 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 }}.jar**

Graphical user interface, text, application, email

Description automatically generated

1. 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.

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

Description automatically generated

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.

Text

Description automatically generated

(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 hightest 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.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**Lab 8 – Merging branches**

**Purpose: In this lab, we’ll look at how to do a Pull Request against branches, deal with conflicts and how to use the Visual Studio web editor.**

1. Let's force an update of a file in main to make sure we have a merge conflict. For simplicity, we'll just update the "package.json" file in the main branch. Switch to the branch "main" and **edit the package.json file and bump one of the revision numbers and then commit the change.**

Graphical user interface, text, application

Description automatically generated

2. Going back to our enhancements for artifact versioning, now that we've proven that the change works in our *artifact* branch, let's merge that branch back into our primary workflow in the *main* branch. In the greetings-ci repo, click on the top-level "Pull requests" menu and then click on "New Pull Request".

Graphical user interface, text, application, email

Description automatically generated

3. This will default to creating a PR to merge back to the original project that this was forked from. We need to adjust the Base Repository back to this one. Click on the "base repository" dropdown and select the "<your primary github userid>/greetings-ci" entry.

Graphical user interface, text, application

Description automatically generated

4. Then the comparison will switch to ones for branches within the repository. On the "compare:" side, select the "artifact" branch.

Graphical user interface, text, application

Description automatically generated

5. You'll get a prompt that says "Can't automatically merge". We'll fix that shortly. Go ahead and click the green "Create pull request" button. Enter a comment if you want and then just click the second "Create pull request" button.

Graphical user interface, text, application

Description automatically generated

6. On the next screen, click on the "Resolve conflicts" button.

Graphical user interface, text, application, email

Description automatically generated

7. The next screen will be the conflict for CHANGELOG.md and package.json. You will already be in the edit screen for this conflict. Just click on each file and highlight the lines you want to remove and Delete them. You can highlight the lines between and including the "========" and ">>>>>>>>> main" lines and then hit the "Delete" key. Also you can delete the line labeled "<<<<<< artifact" at the top.

After that, you can click on the "Mark as resolved" button.

Then just click on the package.json selection on the left and repeat the process.

Graphical user interface, text, application

Description automatically generated

Chart

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

8. Now click on the green "Commit merge" button to the right.

Graphical user interface, text, application, email

Description automatically generated

9. After a few moments, the pre-check should complete. Before we merge the pull request, let's tell GitHub to squash all our commits into one. Click the down arrow next to the "Merge pull request" button and select "Squash and merge". Afterwards, the button should say "Squash and merge" - now click on that.

Graphical user interface, text, application, email

Description automatically generated

10. We don't need all the history from the previous commits, so you can just clear out the comment window except for the most recent commit message specifically for the PR and then click on the "Confirm squash and merge" button.

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

Description automatically generated Graphical user interface, text, application, chat or text message

Description automatically generated

11. We might have expected that the merge would have kicked off another run of our workflow. Why didn't it? Take a look at the updated pipeline.yaml file that is now in main. Notice that the branch that it is triggered on was set to "artifact" when we just did the merge.

Graphical user interface, text, application

Description automatically generated

12. We need to correct that so it is set to trigger on "main" again. We'll edit the pipeline.yaml file in main and change the branch "artifact" references to be "main" instead. Instead of clicking on the pencil icon to edit, change to the file and just hit "." on your keyboard. This will open the VSCode editor in the browser.

Graphical user interface, text, application, email

Description automatically generated

13. Change the references for branches to be "main" instead of "artifact" in the editor.

Graphical user interface, text, application, email

Description automatically generated

14. In the lefthand column, select the branching symbol to change to the source control menu. Click in the commit message area and enter "feat: merge in artifact versioning" and then click the checkmark above to commit and push the changes.

Graphical user interface, text, application

Description automatically generated

15. After this, your change will be committed. You can just go back in the browser to get out of the editor. When you go to the Actions menu, a new run of the workflow should have completed and if you look at the CHANGELOG file, you should see a new minor version update.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

**Lab 9 – Adding in a test case**

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

1. Now let's add a second job to our workflow (in pipeline.yml) to do a simple "test". To start, add the job definition for a job called "test-run" that runs on ubuntu-latest. 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.

You can use either the built-in editor or the vscode editor to edit the file and add the code below. For simplicity, we'll just make the changes in the "main" branch. The screenshot shows where it should go. Pay attention to indentation - "test-run:" should line up with "build:".

**test-run:**

**runs-on: ubuntu-latest**

**needs: build**

**Graphical user interface, text, application, email

Description automatically generated**

2. Now add a step to the job to download our artifact that was uploaded by the build job. Just as there was for the upload function, there is also an action to download the artifact. You can look up more about this if you want at *https://github.com/actions/download-artifact*. For the parameter, we'll just pass the name we called our artifact when we uploaded it.

**steps:**

**- name: Download candidate artifacts**

**uses: actions/download-artifact@v3**

**with:**

**name: greetings-jar**

A picture containing graphical user interface

Description automatically generated

3. Next, 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 and then click on the "Add file" button next to the green "Code" button. From the list that pops up, select "Create new file".

Graphical user interface, text, application, email, Teams

Description automatically generated

4. 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.

**# 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**

Graphical user interface, text, application, email

Description automatically generated

5. 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 *main* branch.

Graphical user interface, text, application, email

Description automatically generated

6. 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.

Edit the pipeline.yaml file in main and 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 }}**

Graphical user interface, text, application

Description automatically generated

7. 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

Graphical user interface, application

Description automatically generated with medium confidence

8. Finally, we can add a step to our test job to invoke the script. This step in the job will just be a shell command, so we'll use the "run" invocation.

We first need to make the file executable on the runner, so we'll use the chmod command.

For testing, our code echoes out the message that we input, so we'll just use the value that is input through our event. Notice that we pass the version value from the previous job as the first parameter so the correct versioned deliverable can be found and the remaining parameters passed in after that. That the "needs.build.outputs.artifact-tag" using the output we setup in step 6.

Edit the pipeline.yml file in the main branch and add the code below as the next step in the test-run job, paying attention to the alignment again.

**- name: Execute test**

**shell: bash**

**run: |**

**chmod +x ./test-script.sh**

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

Graphical user interface, text, application, email

Description automatically generated

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

Graphical user interface

Description automatically generated

10. 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.

Graphical user interface, text

Description automatically generated

11. Let's make one more change to make it easier to run our workflow manually to try things out, start runs, etc. Edit the pipeline.yaml file again 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.

**workflow\_dispatch:**

**inputs:**

**myValues:**

**description: 'Input Values'**

Graphical user interface, application

Description automatically generated

12. 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 "main" is selected. Since we can't force a version through from the manual start, just enter an empty string '' (**that's two single quotes together - not a double quote**) for the first argument then 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.

Graphical user interface, text, application

Description automatically generated

**Lab 10: Working with fast feedback and automatically reporting issues**

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

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

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.
2. In the repository, browse to the "extra" folder and to the "create-failure-issue.yml" file.
3. 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.
4. 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

1. 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

1. 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

1. 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

1. 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 single quotes together on the front and two backslashes between the "de" and "f"). As long as you have the empty string at the start and two backslashes somewhere, this should fail.

'' abc. de\\f ghi

Graphical user interface, text, application, email

Description automatically generated

12. 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