Assignment 2 – Jenkins
(Creating a DevOps Pipeline, CI/CD tool)

Introduction:

Welcome to Assignment 2! This simple exercise is designed to introduce you to Jenkins and continuous integration.

Jenkins is an Open source, Java-based automation tool. This tool automates the Software Integration and delivery process called Continuous Integration and Continuous Delivery.

What is Jenkins?

Jenkins supports various source code management, build, and delivery tools. Jenkins provides features like Jenkins Pipelines which makes the delivery process very easy and helps teams to adopt DevOps easily.

Overview of the Experiment

- Setup Jenkins Using Docker.
- Set up a job in Jenkins to connect to your repository and build C++ hello.cpp.
- Set up a second job to run the program after the build completes.
- Add a webhook trigger to your GitHub repository in order to automate execution of jobs in Jenkins
- Create a basic Jenkins pipeline.

Prerequisites:

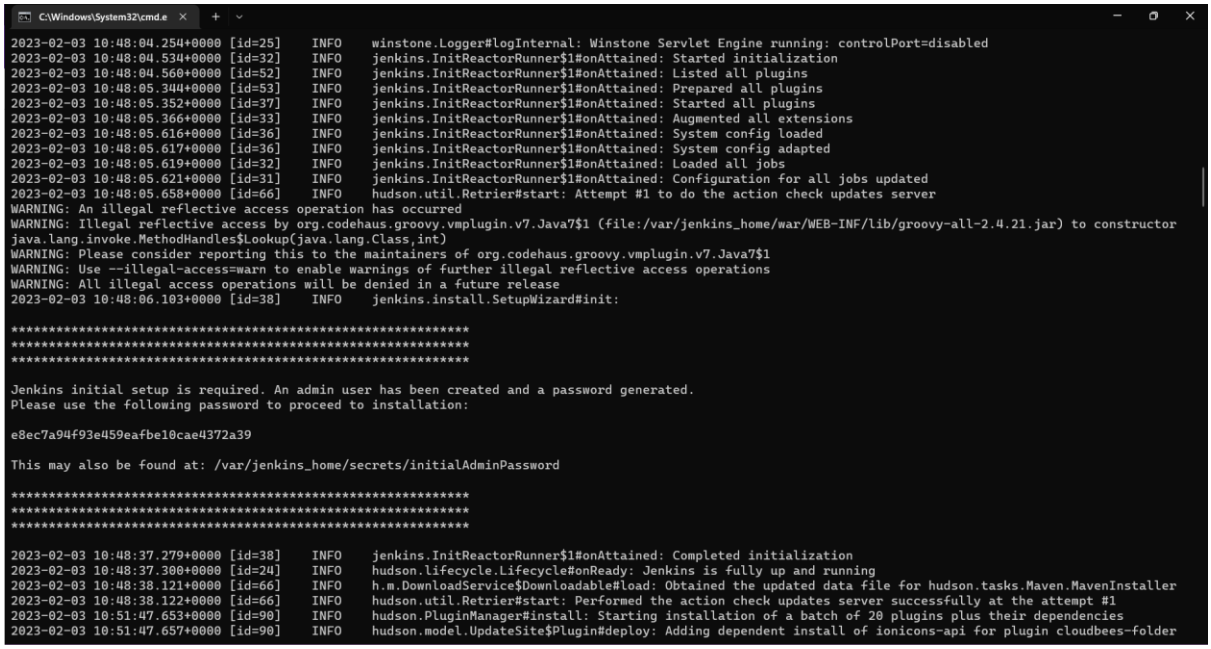
- Docker Installed on your system. (Refer to installation guide steps in the Lab Experiment 2 manual).
- Git installed on your system and a GitHub account

  Follow this tutorial to install and make yourself familiar with git
  https://www.youtube.com/watch?v=2j7fD92g-gE
- Create a GitHub repo with the name as YOUR_SRN_Jenkins

Task-1

Aim: Set up Jenkins using Docker.

Deliverables:



1. Screenshot of the running Docker Container after installing Jenkins

Steps:

1. Use this repository link: https://github.com/ectagithub/Jenkins_lab and download the zip file, extract the Jenkins_lab-main folder.
2. You will be given a Dockerfile, open a terminal in that folder.
3. Build the dockerfile using this command : "`sudo docker build . -t jenkins:YOUR_SRN`" [Note: Omit **sudo** if working with Windows WSL or MacOS]
4. Run your container using this command "`sudo docker run -p 8080:8080 -p 50000:50000 -it jenkins:YOUR_SRN`" (Expose any other port for e.g. 8090:8080, if you are already using port 8080 for some other purpose).(Note down the password shown on the terminal).
5. Open URL: localhost:8080 on your browser.
6. Enter the password shown on your terminal after running the container (You can set the password to ADMIN later).
    a. In case you did not note down the password displayed on the terminal, you can find the password by connecting to the container (via "sudo docker exec

–it <container_id> /bin/bash") and checking the file /var/Jenkins_home /secrets/initialAdminPassword inside the container.

7. **Integrate GitHub to Jenkins**: When prompted for plugin installation, click on "Select Plugins to Install" and then search for GitHub and check the **GitHub** option. (This step may take a few minutes to complete)
8. Take the necessary Screenshots as mentioned in *Deliverables.*

Task-2

Aim: Set up a job in Jenkins to connect to your repository and build C++ hello.cpp.

Deliverables:

1. Picture showing the console output after the build is successful
2. Picture showing the Stable state of the task in Build History of Jenkins

Steps:

1. Make changes to hello.cpp file if you wish. Complete prerequisite 3(If not done already). Then, commit and push the Jenkins-main folder to your GitHub repository. Open Git Bash, navigate to Jenkins-main folder. (Note:- Please make sure to push both the main and Dockerfile folders to your repository, otherwise you may face errors in subsequent tasks). Use the following commands in Git Bash
   - git init
   - git checkout -b main
   - git remote add origin "Your repository's URL"
   - git add .
   - git commit -m "Describe this commit"
   - git log
   - git push -f origin main

2. Navigate to Jenkins server Dashboard. Click ***New Item***.
3. Enter the name for your project as YOUR_SRN-1 (as this is job 1. Ensure the project name is unique to avoid collisions)
4. Click *Freestyle Project*, then *OK*.
5. Select GitHub project and Enter your repository's URL.
6. Set up *Source Code Management*, Select *git*. Enter the URL of git repository.
7. Add another branch with the value "`*/main`" in **Branches to build** (Do not delete the existing */master branch)

8. Setting up *Build Triggers.* Select *Poll SCM*. (This will keep on scanning and poll changes from your repository after a specified interval of time).
9. Set up job by putting in "`H/5 * * * *`" in the Schedule box
   (H/5 * * * * implies it will check your job every 5 minutes. * the syntax of CRON.
   A CRON expression is a string comprising five or six fields separated by white space that represents a set of times, normally as a schedule to execute some routine.)
10. Set up *Build*. In **Add build step** pull-down menu, select *Execute Shell*.
11. Enter "`make -C main`" (This will run the Makefile).
12. Click *Save*.
13. Click on build now.
14. Take the required SS.

Task-3

Aim: Set up a second job that automatically runs after the project builds. This is different from the other job because this will not have a git repository - it doesn't even build anything.

Just a note: In a real-life scenario you wouldn't run a program through a build job just like this because I/O is not possible via this console. There are other tools people use at this step like SeleniumHQ, SonarQube, or a Deployment. The point of this is to show downstream/upstream job relationships.

Deliverables

1. *Console output of second job*
2. *Status page of first job*
3. *Build History of Jenkins*
4. *Jenkins Dashboard*

Steps:

1. Create a new Job in Jenkins, Click *New Item* in the left panel.
2. Enter a name for your second job as YOUR_SRN-2 (as this is your 2nd job), click *Freestyle Project*, then *OK*.
3. Go immediately to the build step and select *Execute Shell*.
4. Enter the following Command `/var/jenkins_home/workspace/<the name of your first project>/main/hello_exec`
5. Click on Save.

Now, set your first job to call the second.

6. Go to your first job (i.e. YOUR_SRN-1) and open the *Configure* page in the pull-down menu.
7. Scroll to bottom and add a Post-Build Action. Select ***Build other projects***.
8. Enter the name of your second job.
9. Click on Save.
10. Run your first job.
11. Do this by clicking build now on the main page.
12. After that successfully builds, go, and check your second job. You should see it successfully run.
13. Select a Build Job from History and go to the console log to see your program output. If you program has run there, then you successfully set up a basic pipeline.
14. Take the required Screenshots.

Task-4

Aim: Add a webhook trigger to your repository in order to automate builds in Jenkins

In the previous tasks, we were polling changes from the repository at an interval of every 5 mins. It is an expensive approach. There is, however, a better approach. By adding a Webhook trigger to your repository and connecting it to your Jenkins server, the instant you commit a change to your repository, your job is automatically executed.

Webhooks allow external services to be notified when certain events happen. When those events happen, a POST request is sent to the designated URL.

Deliverables:

1. Webhook added to your GitHub repository

2. Console Output of second job displaying the change made in hello.cpp file.

Steps:

1. Download ngrok from https://ngrok.com/download

   What is ngrok?

   ngrok is a cross-platform application that enables developers to expose a local development server to the Internet with minimal effort.

2. Open command prompt, navigate to the path where ngrok is downloaded and run the following commands:-

   ngrok –version

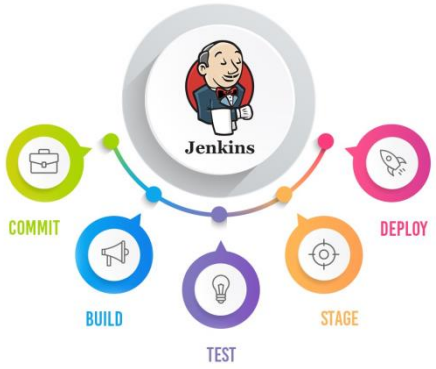
ngrok http 8080 (Since, Jenkins runs on port 8080)

Copy the provided https URL.

3. Go to settings of your GitHub repo, look for Webhooks→ Add webhook → Paste the https URL in the Payload URL field and append /github-webhook/ to it. Keep the default settings, however you can explore individual events to trigger this webhook → Add webhook.
4. Now make a change to your hello.cpp file and commit the change to your repo. Go to the Jenkins server and observe whether your job is executed automatically. Awesome, isn't it?

With this task, we automated our build/execution of jobs and thereby achieving Continuous Integration.

What is Jenkins Pipeline?



In simple words, Jenkins Pipeline is a combination of plugins that support the integration and implementation of continuous delivery pipelines using Jenkins. The pipeline as Code describes a set of features that allow Jenkins users to define pipelined job processes with code, stored and versioned in a source repository.

Why do we need to use Jenkins Pipeline: -

- Pipelines are better than freestyle jobs, you can write a lot of complex tasks using pipelines when compared to Freestyle jobs.

- You can see how long each stage takes to execute so you have more control compared to freestyle.

- Pipeline is a Groovy based script that has a set of plug-ins integrated for automating the builds, deployment and test execution.

- Pipeline defines your entire build process, which typically includes stages for building an application, testing it and then delivering it.

- You can use a snippet generator to generate pipeline code for the stages where you don't know how to write groovy code.

Task-5

Aim: To create a basic Jenkins pipeline.

Deliverables:

1. Code/script written to create basic pipeline using GitHub repository

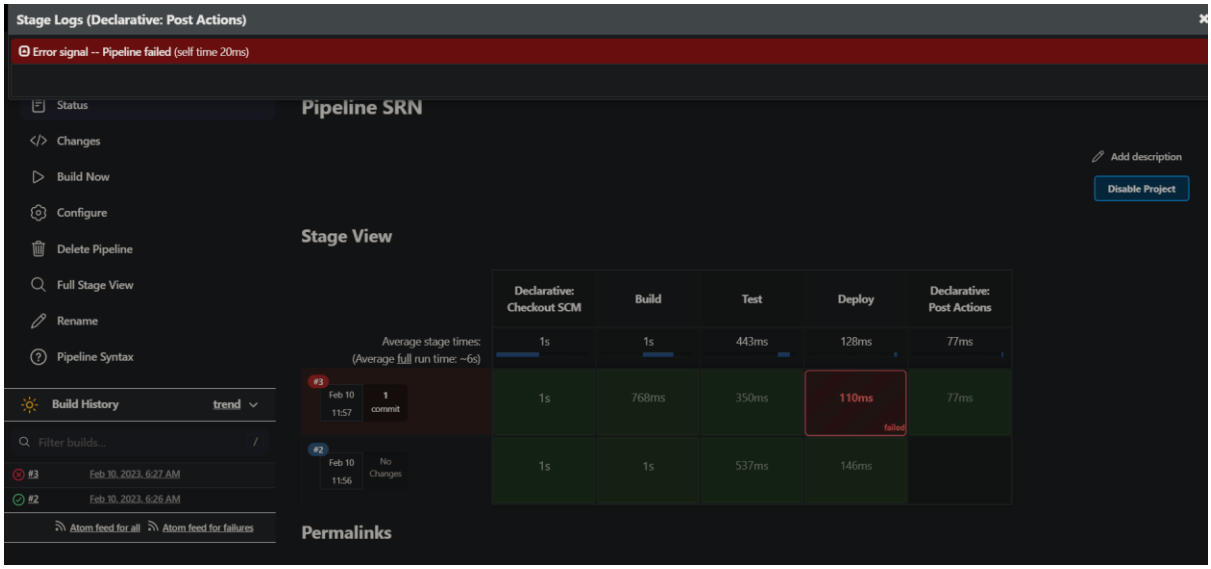
Sample 1 (reference template)

```
pipeline {
    agent {
        docker {
            image 'node:14'
        }
    }
    stages {
        stage('Clone repository') {
            steps {
                git branch: 'main',
                url: 'https://github.com/<user>/<repo>.git'
            }
        }
        stage('Install dependencies') {
            steps {
                sh 'npm install'
            }
        }
        stage('Build application') {
            steps {
                sh 'npm run build'
            }
        }
        stage('Test application') {
            steps {
                sh 'npm test'
            }
        }
        stage('Push Docker image') {
            steps {
                sh 'docker build -t <user>/<image>:$BUILD_NUMBER .'
                sh 'docker push <user>/<image>:$BUILD_NUMBER'
            }
        }
    }
}
```

Sample 2:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean install'
                echo 'Build Stage Successful'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
                echo 'Test Stage Successful'
                post {
                    always {
                        junit 'target/surefire-reports/*.xml'
                    }
                }
            }
        }
        stage('Deploy') {
            steps {
                sh 'mvn deploy'
                echo 'Deployment Successful'
            }
        }
    }
    post {
        failure {
            echo 'Pipeline failed'
        }
    }
}
```

2. Output of working created pipeline, the screenshot should include
    - Stage view / Execution status of pipeline with all stages succeeded
    - Verify Declarative: Post Actions stage succeed for handling failures.



3. Console Output of the Pipeline

4. Link to the created GitHub repository

Steps:

1. Create a job in Jenkins. Name the job/item as YOUR_SRN. Select **Pipeline** under projects.
2. Select Pipeline Script. Under sample pipelines, choose Hello World.
3. Save the pipeline and build. You should now have a basic working pipeline containing 1 stage.
4. Write a Jenkinsfile to create a basic pipeline script:
    - Go to your repository→Add file →create new file→Name the file as Jenkinsfile.
    - Write a script containing a Build, Test, and Deploy stage using Groovy. Also add a post condition to display 'pipeline failed' incase of any errors within the pipeline. Refer to attached scripts. Create a new working .cpp file, push it to your repository.
    - For Build stage:- Compile the .cpp file using shell script, build YOUR_SRN- 1.
    - For the Test stage: - Print output of .cpp file using shell script.
    - Explore pipeline syntax for references.
5. Configure the existing Hello World pipeline job.
6. In pipeline definition, choose Pipeline from SCM
7. Add the link to your GitHub repo in the URL section. Add branch "`*/main`" in **Branches to build** (Do not delete the existing */master branch). Save Pipeline.

8. Execute the pipeline and verify in the stage view whether all stages were executed successfully.
9. Now edit your Jenkinsfile, make an intentional error in one of the stages and commit. Execute the pipeline again, check if the expected stage fails and declarative post action “pipeline failed” carried out successfully.
10. Take required screenshot of the Stage View.