SOFTWARE PRODUCTION ENGINEERING

MINI-PROJECT REPORT

SCIENTIFIC CALCULATOR

Jayam Shanmukha Bhuvan - IMT2020506

GitHub Repository Link - Link

DockerHUb Repository Link - Link

Problem Statement

Create a scientific calculator program with user menu driven operations:

- Addition function - a+b
- Subtraction Function - a-b
- Multiplication Function - a*b
- Division Function - a/b

The project is about creating a scientific calculator program as part of a Software Production Engineering mini project. The calculator has different functions like square root, factorial, natural logarithm, and power. It's implemented on GitHub and can be easily deployed using a Docker image on DockerHub. The project follows DevOps principles, focusing on collaboration, automation, and continuous improvement. Tools like Git, Jenkins, Docker, and Ansible are used for source code management, continuous integration and deployment,

containerization, and infrastructure management. The development process involves writing Java code, testing with JUnit, and using Jenkins for building and deploying the application. The goal is to demonstrate the integration of software development and operations through DevOps practices.

Tools

- Git is a version control system that facilitates collaborative coding and change tracking.
- JUnit is an open-source testing framework for Java, allowing developers to write and execute unit tests.
- Maven serves as a build automation tool that streamlines dependency management and project building for Java.
- Jenkins is a CI/CD tool automating the build, testing, and deployment processes.
- GitHub Webhooks are mechanisms triggering automated actions upon specific events within a GitHub repository.
- Docker is a containerization platform used for packaging applications and dependencies into lightweight, portable containers.
- Ansible automates infrastructure and application deployment and management as a configuration management tool.
- ELK, short for Elasticsearch, Logstash, and Kibana, comprises open-source tools for centralized log management and analysis.

- ngrok creates secure tunnels, allowing local servers to be accessible on the public internet, commonly used for testing and development purposes.

Before going to development and deployment processes of the project, first we need to install all the above mentioned tools.

- Git: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
- Java 17 (Open JDK 17)
- IntelliJ IDE
- Maven: https://maven.apache.org/install.html
- Jenkins: https://www.jenkins.io/doc/book/installing/
- Docker: https://docs.docker.com/engine/install/
- Ansible: https://docs.ansible.com/ansible/latest/installation_guide/index.html
- ngrok: https://ngrok.com/download

And also create repositories on git:

- GitHub: https://github.com/new

Steps to obtain the result:

Setup and Code

The initial steps of setting up the project and utilizing Maven for build automation. It begins by launching IntelliJ IDEA and creating a new project named "Calculator" with Maven as the build tool. Once the project is created, the next step is to configure Maven. We need to verify if Maven is installed by running the command "`mvn --version`". If

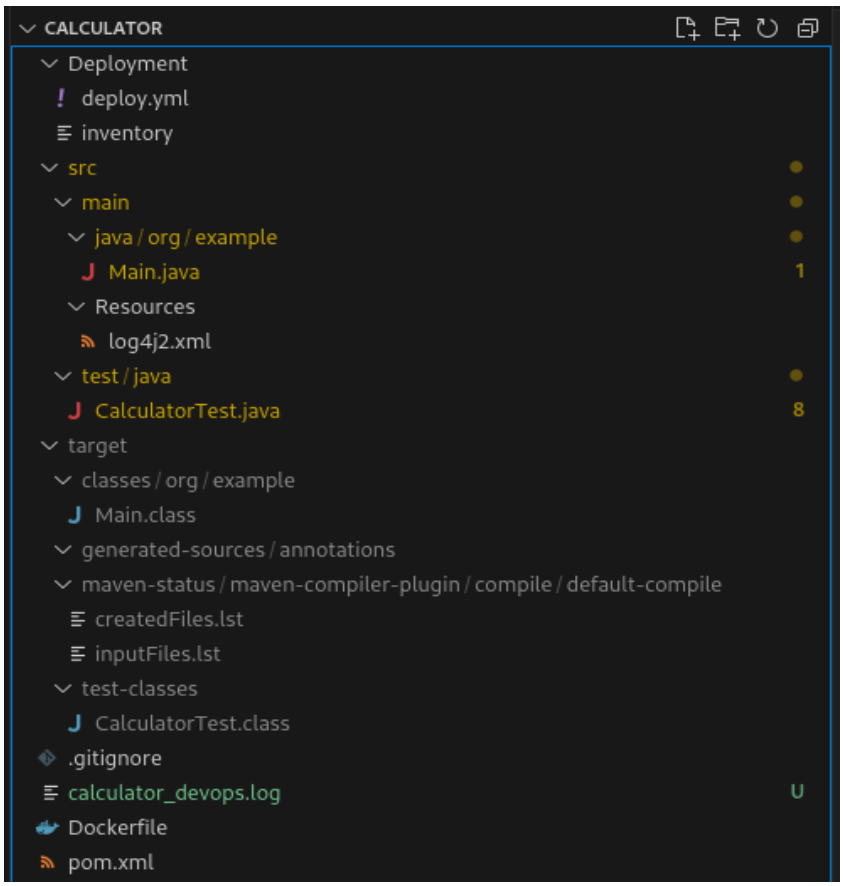
Maven is not installed, we need to install it using the command "sudo apt install maven".

After the maven is installed,

- mvn clean
- mvn compile
- mvn install

We need to run these 3 commands to clean the project, compile the code, and generate a JAR file. There is a pom.xml file, which is used to configure Maven. We can add a build configuration to the pom.xml file to create a JAR file with a specific name.

This is how the file structure looks after doing these 3 steps:



Now we need to implement version control for my project using Git and GitHub. I started by initializing Git in my project directory and creating a remote repository on GitHub. I followed the instructions to stage and commit changes using Git, and I made sure to set up my email and username. Finally, I pushed my local commits to the remote repository. I did the process of integrating Git and GitHub into my project workflow to be straightforward and efficient.

After setting up the repositories, the next step is to create a CI/CD pipeline in Jenkins. This pipeline enables us to automatically pull changes from the repository and build the project whenever new changes are pushed. Once created, we can configure the pipeline stages and dependencies. This integration with Jenkins streamlines the development process and ensures that the project is built and tested consistently with each code change.

CI/CD with Jenkins

To create the CI/CD pipeline in Jenkins, we first need to log in to Jenkins and access the dashboard. From the dashboard, we can click on "New Item" to create a new pipeline. We will name it "Calculator" and select the "Pipeline" option. After creating the pipeline, we can configure it and save the default settings.

Next, we need to add the necessary plugins for our project. To do this, we go to "Manage Jenkins" and then "Manage Plugins". In the "Available

Plugins" section, we search for and install the following plugins: Ansible, Docker, Docker Pipeline, Git Client, Git plugin, GitHub, and JUnit.

Once the plugins are installed, we can proceed to set up the pipeline by adding the pipeline syntax for each stage. However, there may be unmet dependencies for each stage, so we resolve this by installing all the required plugins at once.

Now we need to add 6 stages of pipeline each having its own function.

Stage 1: Cloning Project from Github

The first stage involves cloning the latest changes from the project's master branch, as the pipeline is triggered by commits to the project.

```
stage('Stage 1: Git Clone'){
    steps{
        git branch: 'main',
        url:'https://github.com/Bhuvan506/Calculator.git'
    }
}
```

Stage 2: Building the project using maven

In the second stage of the pipeline, Maven will be used to run the local repository and check if the project is compiling. The focus is on ensuring that the project can run the main method successfully, without running test cases at this stage.

```
stage('Stage 2: Maven Build'){
    steps{
        sh 'mvn clean install'
    }
}
```

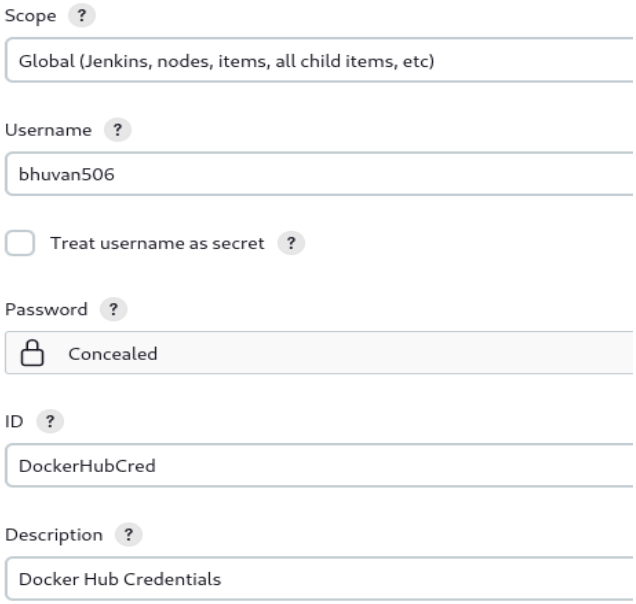
Stage 3: Creating Docker Container

To containerize the project, we will create a Dockerfile that includes the necessary steps and scripts to convert the project into a Docker container. This Docker container will encapsulate all the dependencies required to run the project and can be deployed remotely in later stages.

```
stage('Stage 3: Build Docker Image'){
    steps{
        script{
            docker_image = docker.build "bhuvan506/calculator:latest"
        }
    }
}
```

Stage 4: Pushing Docker Image to Docker Hub

After obtaining the Docker image, our next step is to push it to Docker Hub. To do this, we need to store our Docker Hub credentials in Jenkins. We can navigate to the Dashboard, then go to Manage Jenkins and select Credentials. From there, we can choose System under Stores scoped to Jenkins.



After adding the credentials, we will add the following script to the pipeline:

```
stage('Stage 4: Push docker image to hub') {
    steps{
        script{
            docker.withRegistry('', 'DockerHubCred'){
                docker_image.push()
            }
        }
    }
}
```

Stage 5: Cleaning Docker Images

To prevent errors caused by existing file names, we remove the previously created Docker images during the pipeline. Only the latest image with the "latest" tag will be pushed to Docker Hub, ensuring a clean and error-free image pull in subsequent stages of the pipeline.

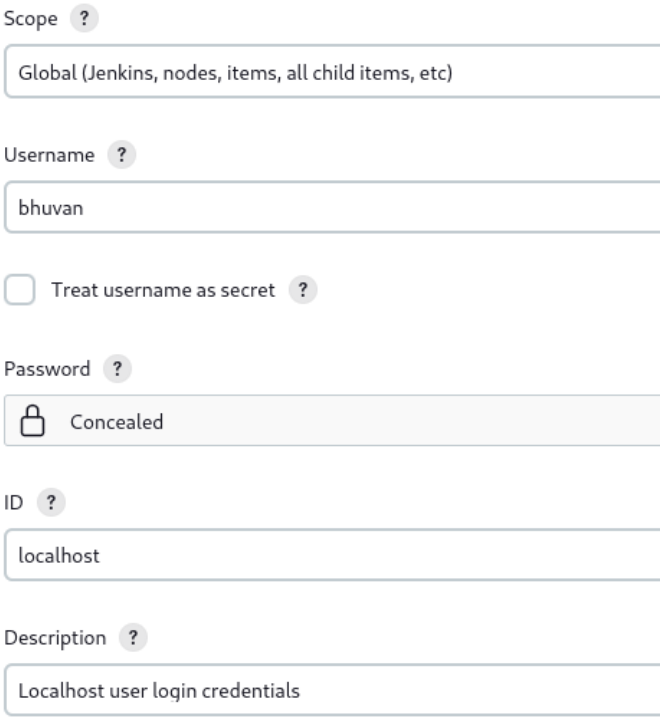
```
stage('Stage 5: Clean docker images'){
    steps{
        script{
            sh 'docker container prune -f'
            sh 'docker image prune -f'
        }
    }
}
```

Stage 6: Pulling Docker Images using Ansible

To deploy the Docker image, we store the deployment details in the "Deployment" subdirectory of the project. This directory contains two files: "inventory" which lists the clients for deployment, and "deploy.yml" which specifies the image to be pulled and deployed.

```
stage('Stage 6: Ansible Deployment'){
    steps{
        ansiblePlaybook becomeUser: null,
        colorized: true,
        credentialsId: 'localhost',
        disableHostKeyChecking: true,
        installation: 'Ansible',
        inventory: 'Deployment/inventory',
        playbook: 'Deployment/deploy.yml',
        sudoUser: null
    }
```

The credentials ID for deploying to our own system will be set as "localhost" to match what we specified in the pipeline script. This ensures that the correct credentials are used during the deployment process.



Now, if we run the pipeline, we will have all the 6 stages running like this:

All these stages together comprise the CI/CD pipeline which involves build, test, containerization, and deployment.

Testing

Now, let's run the tests that we have written in CalculatorTest.java by running the command "**mvn clean install**".

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running CalculatorTest
Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.006 sec

Results :

Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
```

As we can see all the 8 test cases are run successfully.

Logging

To enable logging in our project, we include the necessary log4j2 dependencies in the pom.xml file. We then instantiate the logger in the Java file and use log messages throughout the program to track its execution and capture important information.

```
2023-11-07 22:03:26.816 [main] INFO  org.example.Main - Start of Execution
2023-11-07 22:03:31.076 [main] INFO  org.example.Main - Start OP: Addition
2023-11-07 22:03:31.076 [main] INFO  org.example.Main - End OP: Addition
2023-11-07 22:03:31.077 [main] WARN  org.example.Main - Invalid Input
2023-11-07 22:03:37.909 [main] INFO  org.example.Main - Start OP: Subtraction
2023-11-07 22:03:37.909 [main] INFO  org.example.Main - End OP: Subtraction
2023-11-07 22:03:37.909 [main] WARN  org.example.Main - Invalid Input
2023-11-07 22:03:46.133 [main] INFO  org.example.Main - Start OP: Division
2023-11-07 22:03:46.133 [main] INFO  org.example.Main - End OP: Addition
2023-11-07 22:03:54.821 [main] INFO  org.example.Main - Start OP: Division
2023-11-07 22:03:54.822 [main] INFO  org.example.Main - End OP: Addition
2023-11-07 22:03:58.516 [main] INFO  org.example.Main - End of Execution
```

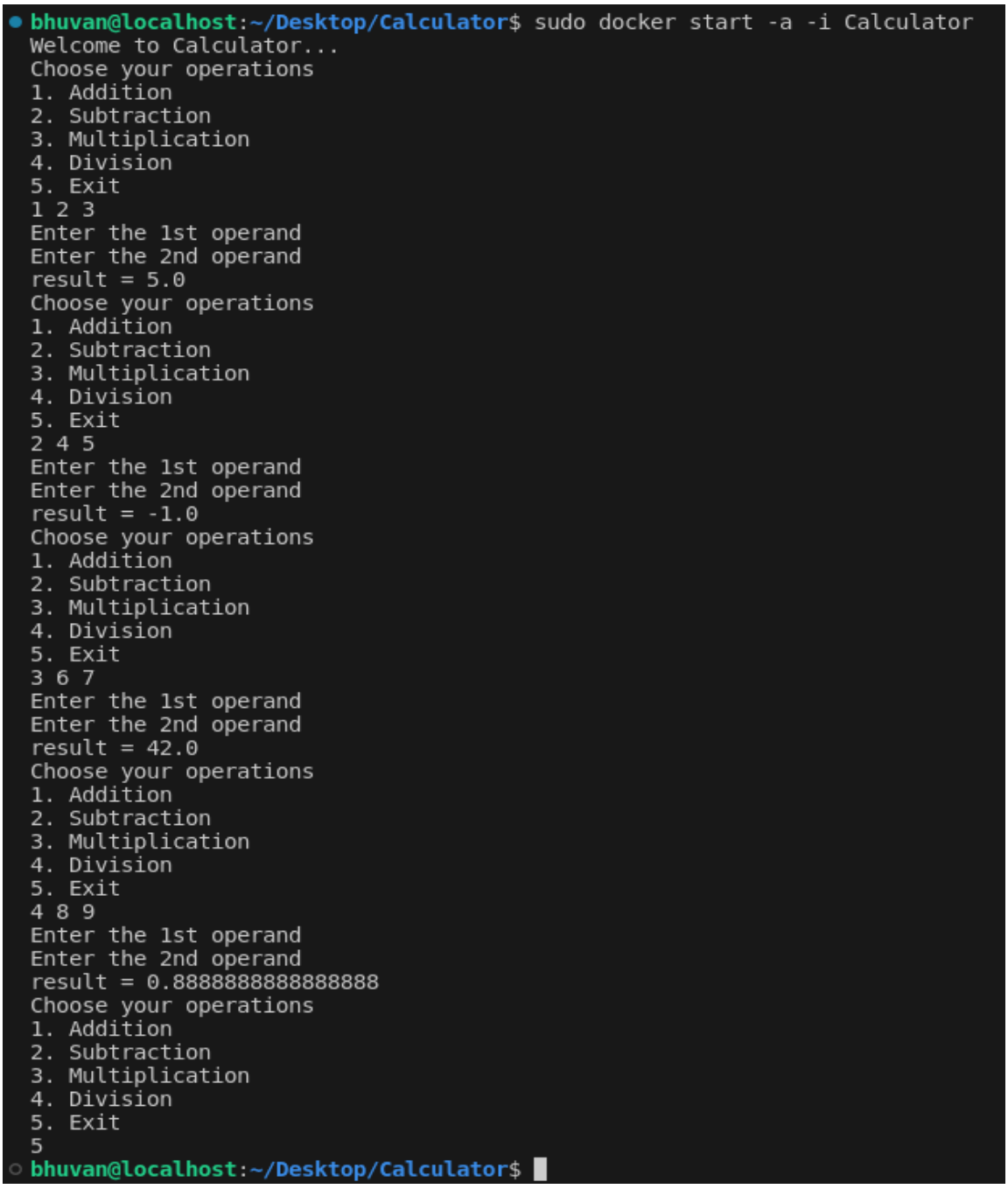
Ngrok and GitSCM Poll

To enable automatic triggering of Jenkins builds when changes are pushed to the Git repository, we follow these steps:

1. I started ngrok to create a tunnel for our Jenkins server, ensuring it remains running continuously throughout the process. Since we are using the free version of ngrok, the URL for the tunnel keeps changing each time we run ngrok.
2. Next, we create a webhook in the GitHub repository. This webhook is responsible for triggering Jenkins whenever changes are pushed to the Git repository. The payload URL for the webhook is set to the forwarding link provided by ngrok.
3. We update the Jenkins URL to match the forwarding link provided by ngrok. Additionally, we set up the GitHub server and add our Git's personal access token to establish the necessary authentication.
4. In the Jenkins configuration, we enable the "GitHub hook trigger for GITScm polling" option. This ensures that Jenkins is notified of new changes in the Git repository and triggers the build process automatically.

After completing these steps, we observe that whenever we commit changes to the repository, Jenkins is automatically triggered and the build process is initiated.

The below image depicts the successful execution of a container, displaying the output of the main program and confirming its functionality.



Below are screenshots of some important code files:

Main.java

4 Functions for the calculator

```java
public static double add(double a,double b)
{
    double res = a+b;
    return res;
}

public static double sub(double a,double b)
{
    double res = a-b;
    return res;
}

public static double mul(double a,double b)
{
    double res = a*b;
    return res;
}

public static double div(double a,double b)
{
    double res = a/b;
    return res;
}
```

Test.java

```java
@Test
public void test_add_positive()
{
    double a = 1;
    double b = 2;
    double expectedResult = 3;
    Assert.assertEquals(expectedResult, calculator.add(a, b), 0.001);
}

@Test
public void test_add_negative()
{
    double a = 1;
    double b = 2;
    double expectedResult = 0;
    Assert.assertNotEquals(expectedResult, calculator.add(a, b), 0.001);
}

@Test
public void test_sub_positiive()
{
    double a = 1;
    double b = 2;
    double expectedResult = -1;
    Assert.assertEquals(expectedResult, calculator.sub(a, b), 0.001);
}
```

Deploy.yml

```yaml
---
- name: Pull Docker Image of Calculator
  hosts: all
  vars:
    ansible_python_interpreter: /usr/bin/python3
  tasks:
    - name: Pull image
      docker_image:
        name: bhuvan506/calculator:latest
        source: pull
    - name: Start docker service
      service:
        name: docker
        state: started
    - name: Running container
      shell: sudo docker run -it -d --name Calculator bhuvan506/calculator
```

Log4j2.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
    <Appenders>
        <Console name="ConsoleAppender" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
        </Console>
        <File name="FileAppender" fileName="calculator_devops.log" immediateFlush="false" append="true">
            <PatternLayout pattern="%d{yyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </File>
    </Appenders>
    <Loggers>
        <Root level="debug">
            <AppenderRef ref="FileAppender"/>
        </Root>
    </Loggers>
</Configuration>
```

Dockerfile

```
FROM openjdk:11
COPY ./target/Calculator-1.0-SNAPSHOT-jar-with-dependencies.jar ./
WORKDIR ./
CMD ["java","-cp","Calculator-1.0-SNAPSHOT-jar-with-dependencies.jar","org.example.Main"]
```

In conclusion, this project has provided a comprehensive demonstration of implementing a DevOps workflow using various tools and technologies. From setting up the repository and configuring the pipeline in Jenkins to containerizing the application with Docker and analyzing logs with ELK Stack, each step has been meticulously explained and executed. By following this project, readers have gained valuable insights into the DevOps practices and the integration of different tools to streamline the software development process. This project serves as a solid foundation for tackling real-world case studies and further exploring the vast possibilities of DevOps in modern software development.