Report: Scientific Calculator with DevOps

Kandarp Dave
IMT2022115

Contents

1 Introduction

1.1 Project Overview

In this project, we were required to develop a scientific calculator with user driven menu operations. I developed a terminal-based application in Java. In order to build and deliver the project, I implemented a DevOps pipeline. The following report discusses the DevOps approach along with the setup details.

GitHub repository: Calculator_SPE
DockerHub registry: calculator_spe

1.2 What is DevOps?

Software developers usually adopt a structured process to design, develop, test, deploy, and maintain software applications. This ensures that the developed software meets the user requirements, while also being high-quality. Some of the methodologies used for this are:

1. **Waterfall Model:** A sequential approach. Each phase is started only after the previous phase ends.

2. **Agile Model:** A flexible and iterative approach. The work is broken down into small phases called sprints. Development happens incrementally.

3. **DevOps Model:** Extends agile with continuous integration and delivery. Blends development and operations for faster deployments.

1.3 Why is DevOps used?

In traditional models like `Waterfall` and `Agile`, developers solely focus on writing code. The operations team is responsible for deploying and maintaining the application. But, this often causes delays and miscommunication. This is clearly not desirable while developing any application.

The aim of `DevOps` is to integrate development and operations into a single collaborative process. It is a cultural philosophy that empowers teams to work together more effectively. Frequent updates and fixes can be deployed automatically, thus reducing manual work and human errors.

Various different tools are used to implement Continuous Integration and Continuous Deployment pipelines. Testing and monitoring of applications is also automated. This results in efficient resource usage which leads to lower costs.

2 Tools Used

2.1 Source Control Management

`GitHub` was used as the source control management system. It allows multiple developers to work on the same project without interfering in each other's work. It can also connect with tools like `Jenkins`, `Docker`, `Kubernetes`, and cloud services.

2.2 Test

The `JUnit` framework was used to create unit tests for the calculator application. Unit tests are meant to test small units of code. They help to catch errors at the unit level before they spread into larger modules.

2.3 Build

`Maven` was used to run the unit tests and build the Java project. It helps to automatically build processes and manage the various dependencies of the project.

2.4 Continuous Integration

Continuous integration is the practice of merging additional code into an already existing code repository. Each integration must be tested and build. `Jenkins` was used for this purpose.

`Jenkins` can be integrated with version control systems like `GitHub` to detect changes in the repository. After this, it can run the unit tests provided and then build the application. It can also works with tools like `Docker`, `Kubernetes`, and `Ansible` for deployment.

2.5 Containerization

A container is a self-contained unit that includes the code, runtime, libraries, and system tools required to run an application. Containerizing an application allows it to be portable and run in an isolated environment. For this, `Docker` was used.

`Docker` allows the developer to provide the necessary instructions to containerize an application. Using these instructions, a docker image can be created. This image can be run by end users in the form of containers.

2.6 Deployment

`Ansible` is an open-source IT automation tool that helps you manage systems, deploy applications, and orchestrate IT infrastructure. For this project, `Ansible` was used to deploy the application.

3 Setup

3.1 Creating a Maven Project

To develop the project, I used the `IntelliJ IDE`. I used `Java (21)` for this project.

Create a new `Maven` Project. Set the archetype to be `maven-archetype-quickstart`.
I used the bundled `Maven`, which was version **3.9.9**. Refer to my `GitHub` repository to get `pom.xml`.

Note: Change the `mainClass` attribute in the `maven-jar-plugin` as per your main class. I used org.calculator.App as my main class.

Write your code in the `src` directory and tests in the `test` directory. Additionally, create `Jenkinsfile`, `Dockerfile`, `.dockerignore`, `inventory.ini`, and `playbook.yml` files in the **root directory** of the project.
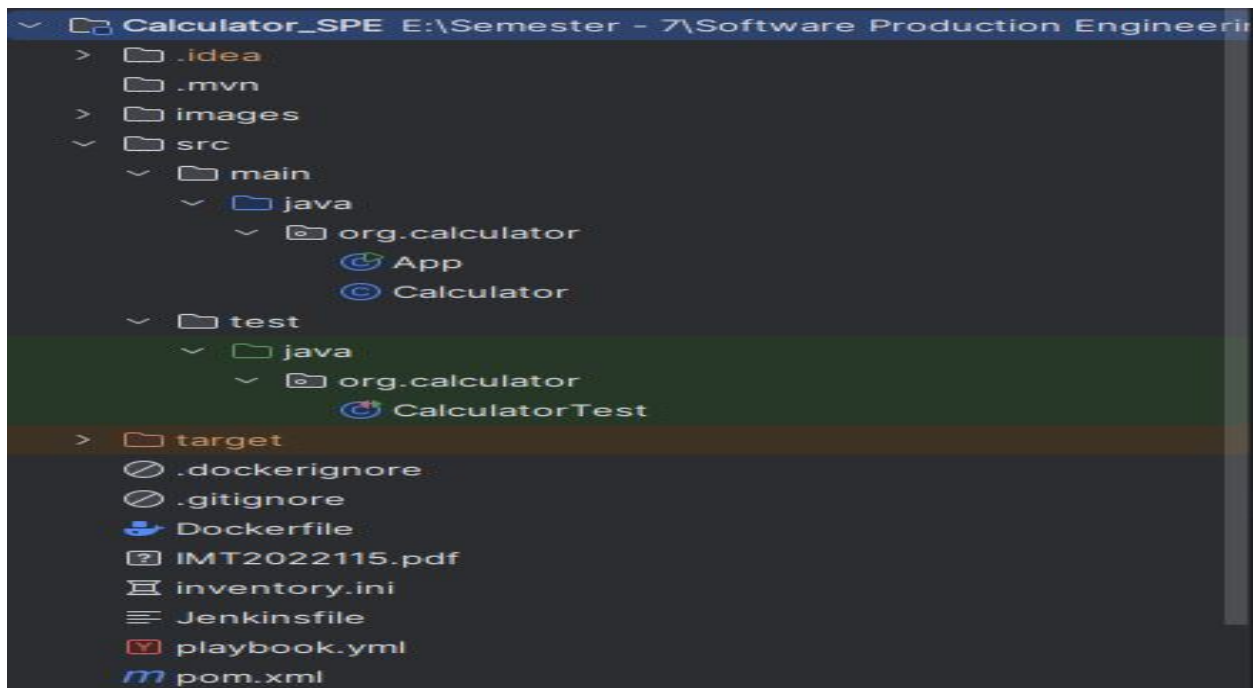

Figure 1: Maven Project Structure.

Figure 2: Calculator class.

Figure 3: App class.

3.2 Setting up the GitHub repository

Create a new **public** repository on GitHub. In your root directory of the maven project, run these commands:

```
git init
git add .
git commit -m "Initial commit"
git remote add origin https://github.com/your-username/your-repo.git
git branch -M main
git push -u origin main
```

This will push your maven project to GitHub.

Figure 4: CalculatorTest class.

3.3 Creating a Jenkins project

3.3.1 Installation

First, make sure that `java` is installed. After that, install `Jenkins`.

Start the `Jenkins` service.

```
sudo systemctl start jenkins
sudo systemctl status jenkins
```

You should see that `Jenkins` is active and running. Open `Jenkins` on your browser. The url is `http://localhost:8080`. Follow the instructions given. You may also create a new user if needed.

3.3.2 Exposing Jenkins using ngrok

`Jenkins` is currently running locally. In order to integrate it with `GitHub`, we must expose it so that `GitHub` can connect with it. This can be done using `ngrok`. First, sign up on ngrok.com. Then, install `ngrok` on your system.

Connect to your ngrok account using the `authtoken`. You can get this token from here once you sign in.

Each ngrok user has access to a static domain. It can be found here. Go to **Jenkins > Manage Jenkins > System** and update the **Jenkins URL**.

Figure 5: Update Jenkins URL.

To expose `localhost:8080` to this static domain, run the command:

```
ngrok http --url=your-static-domain 8080
```
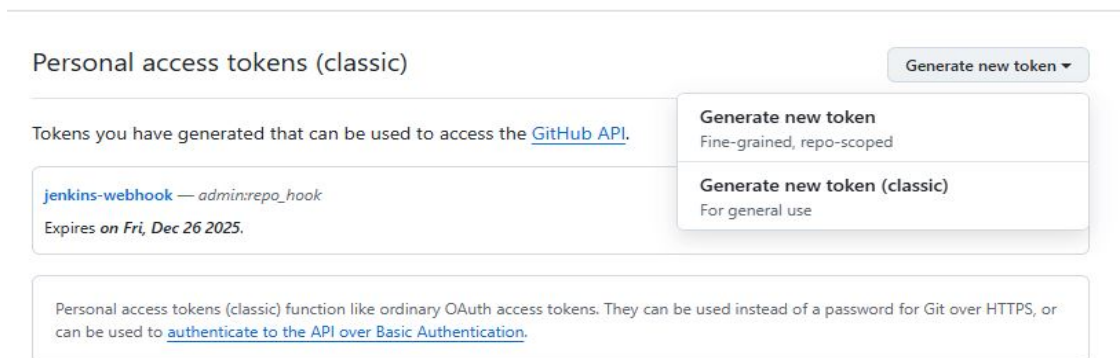

Figure 6: Using ngrok to expose jenkins.

3.3.3 Create GitHub Webhook

On your `GitHub` profile, go to **Settings** > **Developer Settings** > **Personal access tokens** > **Tokens (classic)**. Generate a new class token and copy the secret text.

Figure 7: Generate new token (classic).

6

Figure 8: Set appropriate name and expiration date.

Figure 9: Set appropriate control.

Figure 10: Get token. Make sure to copy it.

In your `GitHub` repository, go to **Settings** > **Webhooks** and click on `Add webhook`. Refer to the image for the creation of the webhook. Copy the token generated in the previous step here.

Figure 11: Create webhook.

Figure 12: Also setup credentials in **Jenkins** > **System**.

3.3.4 Create Jenkins Pipeline project

On the `Jenkins` Dashboard, click on `New Item` and create a **Pipeline** project. Apply these configurations:

- **GitHub project:** Provide `GitHub` url.

- **Triggers:** GitHub hook trigger for GITScm polling.

- **Pipeline:** Pipeline script from SCM.

– SCM: Git.

– Repository URL: Your `GitHub` url.

– Credentials: <u>Add a secret text credential. The secret has to be the same as provided in the GitHub webhook.</u>

– Branches to build: */main.

– Script path: Jenkinsfile

Figure 13: Pipeline Project.

Figure 14: Pipeline Project.

Figure 15: Pipeline Script.

3.4 Integrating Docker with Jenkins

3.4.1 Installation

After installing docker locally, sign up on DockerHub.

3.4.2 Integration with Jenkins

In `Jenkins`, install the necessary `Docker` plugins.
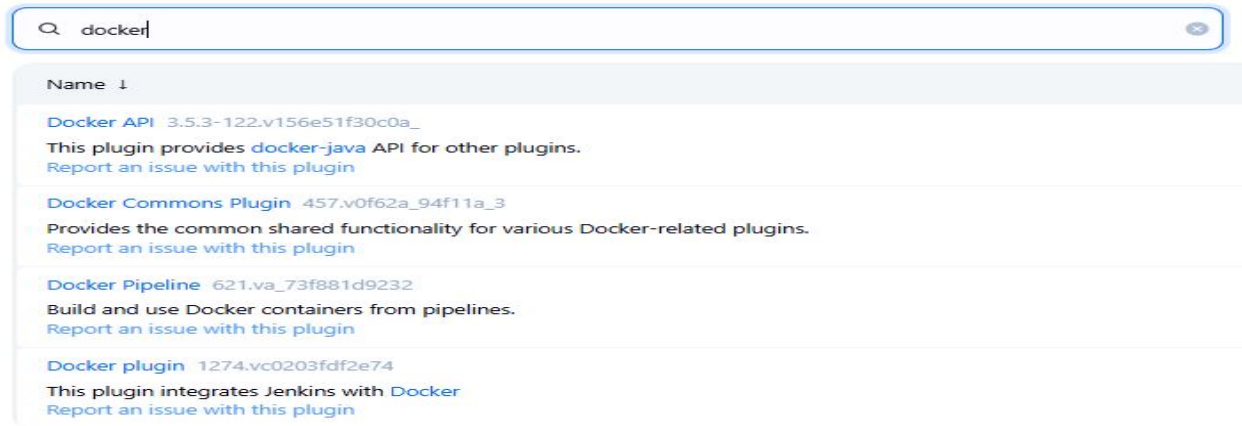
Figure 16: Install `Docker` plugins.

Add `Jenkins` user to `Docker` group.

```
sudo usermod -aG docker jenkins
sudo systemctl restart jenkins
```

Also add `DockerHub` credentials in your `Jenkins`.

3.5 Ansible

3.5.1 Installation

```
sudo apt update
sudo apt upgrade -y

sudo apt install ansible -y
```

3.5.2 Integration with Jenkins

Make sure to give appropriate permissions to the `jenkins` user in order to run the `ansible-playbook` command.

4 Pipeline Explanation

1. **Push changes to GitHub:** Write/Modify your source code. After that, push your code to your `GitHub` repository. This will activate the webhook, which communicates the changes to Jenkins.

2. **Jenkins pipeline execution**

   The stages of the pipeline are mentioned in `Jenkinsfile`.

```
pipeline
{
    agent any

    options
    {
        skipStagesAfterUnstable()
    }

    stages
    {
        stage("Run Tests")
        {
            steps
            {
                echo "Testing Calculator..."
                sh "mvn test"
            }
        }
```

Figure 17: Jenkinsfile screenshot 1.

```
stage("Check Docker version")
{
    steps
    {
        echo "Checking docker version..."
        sh '''
        docker version
        '''
    }
}

stage("Build Docker image")
{
    steps
    {
        echo "Building Docker image..."
        script
        {
            docker.build("kandarp53/calculator_spe:latest")
        }
    }
}
```

Figure 18: Jenkinsfile screenshot 2.

```
stage("Push Docker image to Registry")
{
    steps
    {
        echo "Pushing Docker image to Dockerhub..."
        script
        {
            docker.withRegistry("https://index.docker.io/v1/", "dockerhub-credentials") {
                docker.image("kandarp53/calculator_spe:latest").push()
            }
        }
    }
}

stage("Deploy using Ansible")
{
    steps
    {
        sh "ansible-playbook -i inventory.ini playbook.yml"
    }
}
```

Figure 19: Jenkinsfile screenshot 3.

```
post
{
    always
    {
        script
        {
            def jobName = env.JOB_NAME
            def buildNumber = env.BUILD_NUMBER
            def pipelineStatus = currentBuild.result ?: "SUCCESS"

            mail to: "Dave.Kandarp@iiitb.ac.in",
                subject: "${jobName} - Build ${buildNumber}",
                body: "Calculator project pipeline status: ${pipelineStatus.toUpperCase()}"

            echo "Cleaning workspace..."
            cleanWs()
        }
    }
}
```

Figure 20: Jenkinsfile screenshot 4.

<u>agent any:</u> Run the pipeline on any available node in the `Jenkins` environment. Note that the chosen node must have all the tools (`Maven`, `Docker`, `Ansible`, etc.) installed.

<u>options:</u> If at any stage the pipeline becomes unstable, then none of the subsequent stages are executed. This happens when the tests fail but the build is not marked as **FAILURE**.

(a) <u>Run Tests:</u>
Run the `JUnit` tests as mentioned in the `CalculatorTest` class. This must be done to ensure that the code is not erroneous and passes all the mentioned testcases.

```
[1;34mINFO[m] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
[1;34mINFO[m]
[1;34mINFO[m] -------------------------------------------------------
[1;34mINFO[m]  T E S T S
[1;34mINFO[m] -------------------------------------------------------
[1;34mINFO[m] Running org.calculator.[1mCalculatorTest[m
[1;34mINFO[m] [1;32mTests run: [0;1;32m4[m, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.125 s -- in
org.calculator.[1mCalculatorTest[m
[1;34mINFO[m]
[1;34mINFO[m] Results:
[1;34mINFO[m]
[1;34mINFO[m] [1;32mTests run: 4, Failures: 0, Errors: 0, Skipped: 0[m
[1;34mINFO[m]
[1;34mINFO[m] [1m------------------------------------------------------------------------[m
[1;34mINFO[m] [1;32mBUILD SUCCESS[m
[1;34mINFO[m] [1m------------------------------------------------------------------------[m
[1;34mINFO[m] Total time:  5.661 s
[1;34mINFO[m] Finished at: 2025-10-08T19:28:10+05:30
[1;34mINFO[m] [1m------------------------------------------------------------------------[m
```

Figure 21: Running `JUnit` tests using `Maven`.

(b) <u>Check Docker version:</u>
While implementing the pipeline, I faced several issues like `Docker` plugins not installed and missing permissions. The main aim of this stage is to quickly verify whether `Jenkins` is able to run the `docker` command or not.

Figure 22: Checking `Docker` version.

(c) <u>Build Docker image:</u>

After verifying that `Jenkins` can indeed run `docker`, we proceed to build the docker image. The steps to build the image are mentioned in the Dockerfile. It is a multi-stage build.

Stage 1: Build: A lightweight maven image is used for the build. First, the dependencies are installed based on the `pom.xml` file. The project is not yet built, only the dependencies are installed. After that, the code present in `src/main` directory is copied into the `/app` directory of the container. Note that we exclude the test code here. We skip testing since it is already done in the earlier stage. A JAR file for the application is created.

Stage 2: Run: For running the JAR file, we use a lightweight image which contains only the Java JRE, and not the JDK. From the build stage, the JAR file is copied to `/app` directory. The ENTRYPOINT defines the default command to run when the container starts.

Figure 23: Dockerfile.

```
+ docker build -t kandarp53/calculator_spe:latest .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  818.7kB

Step 1/10 : FROM maven:3.9.9-eclipse-temurin-21-alpine AS build
 ---> be0d0a48232f
Step 2/10 : WORKDIR /app
 ---> Using cache
 ---> 9f690acd613a
Step 3/10 : COPY pom.xml .
 ---> Using cache
 ---> 13887e6aa549
```

Figure 24: Build Docker image.

(d) Push Docker image to Registry:

After the Docker image is built, we need to push it to `DockerHub` so that it can be pulled whenever needed. Using the `DockerHub` credentials, we push the image to our `DockerHub` registry.

```
+ docker push index.docker.io/kandarp53/calculator_spe:latest
The push refers to repository [docker.io/kandarp53/calculator_spe]
b10782689bf3: Preparing
283873fb05a5: Preparing
880a6d9a5a59: Preparing
bb64f233ca86: Preparing
1eb3de508cc3: Preparing
c2d2b55d55c7: Preparing
418dccb7d85a: Preparing
c2d2b55d55c7: Waiting
418dccb7d85a: Waiting
283873fb05a5: Layer already exists
880a6d9a5a59: Layer already exists
```

Figure 25: Push docker image to `DockerHub`.

(e) Deploy using Ansible:

Using `Ansible`, we can deploy our docker image to one or more target hosts. In our case, the target host is our system itself.

The `inventory.ini` file specifies which hosts Ansible should connect to. SSH username and password need to be mentioned for each host. In this project, since we are deploying the application in our local system, no username and password were mentioned. The `playbook.yml` file mentions what tasks to run on the specified hosts. In this application, the task is to pull the image from `DockerHub` and create a running container. Any existing container is first removed.

14

Figure 26: inventory.ini.

Figure 27: playbook.yml.

Figure 28: Executing ansible playbook.

Post Actions:

After the pipeline is executed, the post actions are executed. The `always` keyword means that regardless of the pipeline status, these actions must be done. In our case, the first post-action is email notification. An email is sent which shows the result of executing the pipeline. The second post-action is clearing the workspace. All the files that are created during pipeline execution are removed.

15

```
[Pipeline] { (Declarative: Post Actions)
[Pipeline] script
[Pipeline] {
[Pipeline] mail
[Pipeline] echo
Cleaning workspace...
[Pipeline] cleanWs
[WS-CLEANUP] Deleting project workspace...
[WS-CLEANUP] Deferred wipeout is used...
[WS-CLEANUP] done
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Figure 29: Post Actions.

3. **Run the application:**

   If the Jenkins pipeline gets successfully executed, then a docker container named calculator is created on the user's system. In order to execute the application, use the command:

   ```
   docker exec -it calculator java -jar /app/app.jar
   ```

Figure 30: Execute the application.