CS 816 - Software Production Engineering

Mini Project - Scientific Calculator with DevOps

Nisha Hitesh Rathod, MT2023195

Project Links

Github: https://github.com/nisharathod2301/Calculator_DevOps.git
Docker Hub: https://hub.docker.com/r/nisharathod232001/calculator_image

Problem Statement

Create a scientific calculator program with the following user menu driven operations:

- Square root function - $\sqrt{x}$
- Factorial function - x!
- Natural logarithm (base e) - ln(x)
- Power function - x_b

Devops Tool Chain Used

- Source control management tool - GitHub
- Testing - JUnit
- Build - Maven
- Continuous Integration - Jenkins
- Containerization - Docker
- Deployment - Ansible
- Language - Java

What & Why DevOps?

Ref: https://www.oreilly.com/library/view/the-devops-handbook/9781457191381/

DevOps combines development and operations practices to streamline software delivery. It emphasises collaboration, automation, and continuous improvement to accelerate the development lifecycle. In this project, DevOps ensures seamless integration of features like square root, factorial, logarithm, and power functions into a scientific calculator. By automating testing, building, and deployment processes, DevOps enables faster delivery of reliable software updates. Continuous integration ensures code quality through automated testing with JUnit. Containerization with Docker ensures consistent deployment across different environments. Integration with CI/CD tools like Jenkins or GitLab CI automates the pipeline,

improving efficiency. Configuration management tools like Ansible or Puppet ensure consistent deployment configurations. Cloud deployment options like AWS or Google Cloud offer scalability and flexibility. Overall, DevOps fosters a culture of collaboration and innovation, resulting in faster, more reliable software delivery.

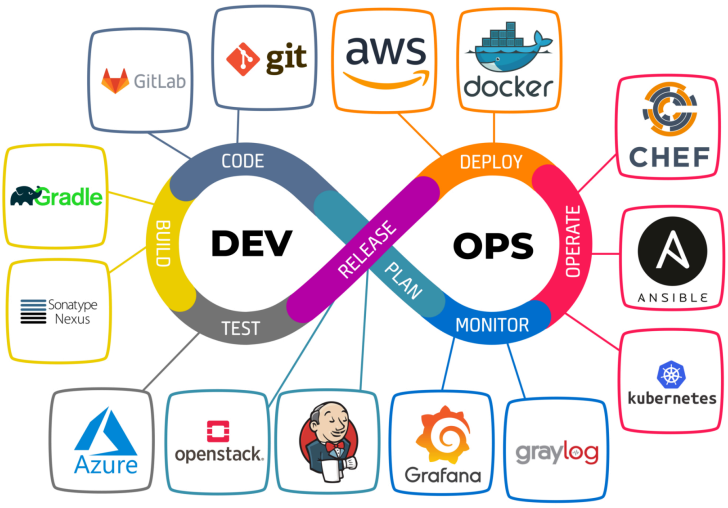


Fig 1:.Devops Model

So using DevOPs for our project introduces us to these technologies. By integrating these tools into a cohesive pipeline, the scientific calculator application can undergo continuous development, testing, and deployment with efficiency and reliability!

Project Flow

Creating a New Project

1. Firstly, we start by creating a new project on **IntelliJ IDEA,** making a new Maven project.
2. You'll see a file system, with a nested Java repo, this is where we will write our code.
3. In org.example > Main: this is the entry point of our project. In our case we want to make Calculator as our entry point class so left click on Main class and go to Refactor and Rename it to Calculator.
4. Now we are ready to write our code in this Calculator class. Another option is to make a separate Calculator class and then call it to the main class, but I will stick to the earlier mentioned method.

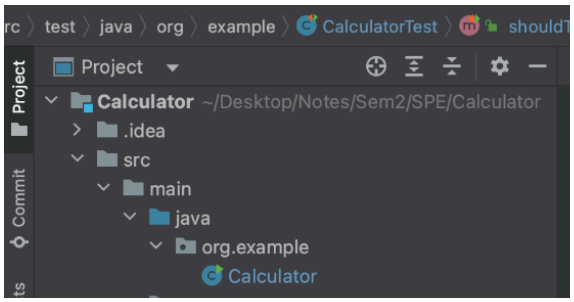


Fig 2:. File system

Changing pom.xml file

Now we need to add the entry point of our project for the compiler to identify, hence we add it in our pom.xml file. In my case the entry point is Calculator class. As you can see in the image attached below, we have added org.example.Calculator in our <mainClass> tag, we also have maven-jar-plugin added with the installed version.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.2.0</version>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>org.example.Calculator</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Fig 3:. pom.xml

So we will run this on our terminal by using ***mvn clean install.*** The clean phase removes any previously compiled Java classes, resources, and other files generated during the build process.The install phase compiles the source code, executes any tests defined in the project, packages the compiled code into a distributable format (e.g., JAR or WAR file), and installs the packaged artifact into the local Maven repository. After execution is completed you'll be able to see an example.jar file created in the target repository!This has all the class files and it runs the project simply by using ***java -jar example.jar***

Making Docker File

Create a file directly under the root folder, in the same indentation as pom.xml file, and name it Dockerfile. This Dockerfile defines a Docker image for running a Java application.

- FROM openjdk:17-oracle: Specifies the base image to use, which is an OpenJDK image with Java 17 from Oracle.
- WORKDIR /app: Sets the working directory inside the Docker container to /app.
- COPY target/Calculator_MT2023195-1.0-SNAPSHOT.jar caculator.jar: Copies the JAR file named Calculator_MT2023195-1.0-SNAPSHOT.jar from the target directory of the local filesystem into the /app directory of the Docker container, renaming it to calculator.jar.

```
FROM openjdk:17-oracle
WORKDIR /app
COPY target/Calculator_MT2023195-1.0-SNAPSHOT.jar caculator.jar
```

Fig 3:. Docker file

Making Jenkins Pipeline

Before creating a complex pipeline lets create a simple pipeline, with 3 stages, Checkout, Build, Docker Build! We will first create a Jenkins file in the same indentation as Docker file, now we open our terminal and use command ***jenkins-lts*** (since i am using lts), this will redirect you to Jenkins on your local host, we install required plugins and set up credentials. So let's go back to our Jenkinsfile and edit it!
This Jenkinsfile defines a Jenkins pipeline with the following stages:

- Checkout:Checks out the code from the specified Git repository (https://github.com/nisharathod2301/Calculator_DevOps.git) and the main branch.
- Build:Executes the Maven command mvn clean install to clean the project, compile the source code, run tests, and package the code into a distributable format.
- Docker Build:Builds a Docker image using the Dockerfile located in the repository root directory (.).Tags the Docker image with the name nisharathod232001/calculator_image. (nisharathod232001 / calculator_image)

  In summary, this pipeline automates the process of checking out the code, building the Java project with Maven, and creating a Docker image for the application.

Making Docker image

Now we will make a docker container for our image by using following commands:
docker run -it username/image_name sh
docker ps -all

```
nisharathod@Nishas-MacBook-Air ~ % docker run nisharathod232001/calculator_image

Feb 24, 2024 4:51:58 PM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 17.0.2
|  For an introduction type: /help intro

jshell> %
                                                    nisharathod@Nishas-MacBook-Air
 ~ % docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED      STATUS
PORTS                      NAMES
6db37e213818   alpine         "sh"                     6 days ago   Up 6 days
                           loving_napier
15ce709fd1a1   alpine/socat   "socat tcp-listen:23…"   8 days ago   Up 8 days
127.0.0.1:2376->2375/tcp   gifted_brattain
nisharathod@Nishas-MacBook-Air ~ % docker ps --all
CONTAINER ID   IMAGE                                COMMAND                  CRE
ATED          STATUS                       PORTS                     NAMES
a56e1ac4e405   nisharathod232001/calculator_image   "jshell"                 18
```

Fig 4:. Creating docker container

Setting up docker access token

Login to Docker Hub and go to Profile > Security > New Access Token > Enter Credentials
Now you can copy paste your access token. We make this access token so that Jenkins knows where to pull the image from.
Go to Jenkins > Credentials > Global Credentials > Add New Credentials >
Kind : Secret text

Scope : Global
Secret : *access token*
Id : dockerhub
Now go to plugins and install Docker pipeline, Docker-build-step, and restart Jenkins.

Docker push stage in pipeline

Go to jenkinsfile and add a new stage. This stage in the Jenkins pipeline handles pushing the Docker image to DockerHub:

- Push Image to DockerHub:
  Executes the Docker login command (docker login -u $DOCKER_USERNAME -p $DOCKER_PASSWORD) using environment variables for DockerHub username and password.
  Authenticates with DockerHub using the provided credentials.
  Pushes the Docker image tagged as *nisharathod232001/calculator_image* to the DockerHub repository.

```
stage('Push Image to DockerHub'){
    steps {
        script {
            sh 'docker login -u $DOCKER_USERNAME -p $DOCKER_PASSWORD'
            sh 'docker push nisharathod232001/calculator_image'
        }
    }
}
```

Fig 5:. Push Image to Docker Hub

Now go to Jenkins and Build this, after build is executed you'll be able to see this container on Docker Hub! This image will be pulled by our prod server and will be used, but for now lets try on our local system!
docker pull user_name/image_name
docker run -it user_name/image_name sh
ls
java -jar calculator.jar

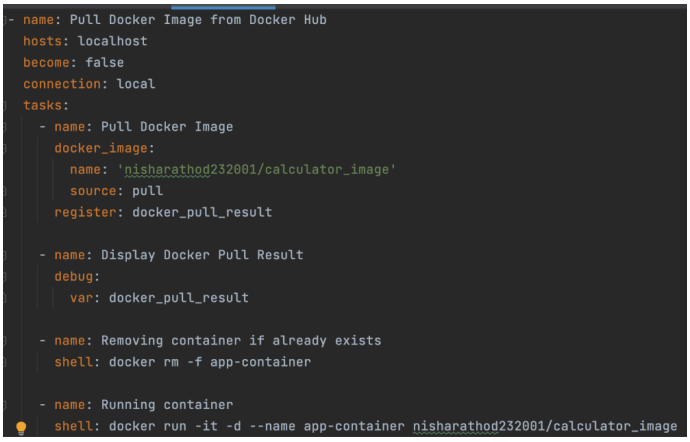
```
sh-4.4# ls
caculator.jar
sh-4.4# java -jar caculator.jar

Scientific Calculator
1. Square root (√x)
2. Factorial (x!)
3. Natural logarithm (ln(x))
4. Power (x^b)
5. Exit
Enter your choice: 1
Enter a number: 16
Square root of 16.0 is: 4
```

Fig 6:. Running .jar file

Setting up Ansible

Ansible is an open-source automation tool that simplifies IT orchestration, configuration management, and deployment tasks through declarative YAML syntax. We will install it by using ***brew install ansible,*** now let's configure it on Jenkins. First download ansible plugins. Now go to jenkins > Tools > Ansible Installations > Add Ansible
Name : local ansible server
Path : /opt/homebrew/bin/
Now make two files deploy.yml and inventory. Inventory will have ***ansible_host = localhost*** and deploy.yml will have following parts:

- Pull Docker Image from Docker Hub:
  Connects to the localhost and pulls the Docker image named nisharathod232001/calculator_image from Docker Hub.
  Registers the result of the Docker image pull operation for later use.
- Display Docker Pull Result:
  Prints the result of the Docker image pull operation for debugging purposes.
  Removing the container if it already exists.
  Removes any existing Docker container named app-container.
- Running container:
  Executes a Docker command to run a new container named app-container in detached mode (-d) based on the pulled Docker image.



```yaml
- name: Pull Docker Image from Docker Hub
  hosts: localhost
  become: false
  connection: local
  tasks:
    - name: Pull Docker Image
      docker_image:
        name: 'nisharathod232001/calculator_image'
        source: pull
      register: docker_pull_result

    - name: Display Docker Pull Result
      debug:
        var: docker_pull_result

    - name: Removing container if already exists
      shell: docker rm -f app-container

    - name: Running container
      shell: docker run -it -d --name app-container nisharathod232001/calculator_image
```

Fig 7:. Deploy.yml

Ngrok

Since Jenkins is on local host, how will my github know where my jenkins is? So ngrok will create a public ip address that will point to my jenkins. Whenever we install github plugin on jenkins it automatically creates a webhook endpoint in jenkins at <Jenkins_url>/GitHub_webhook/Url

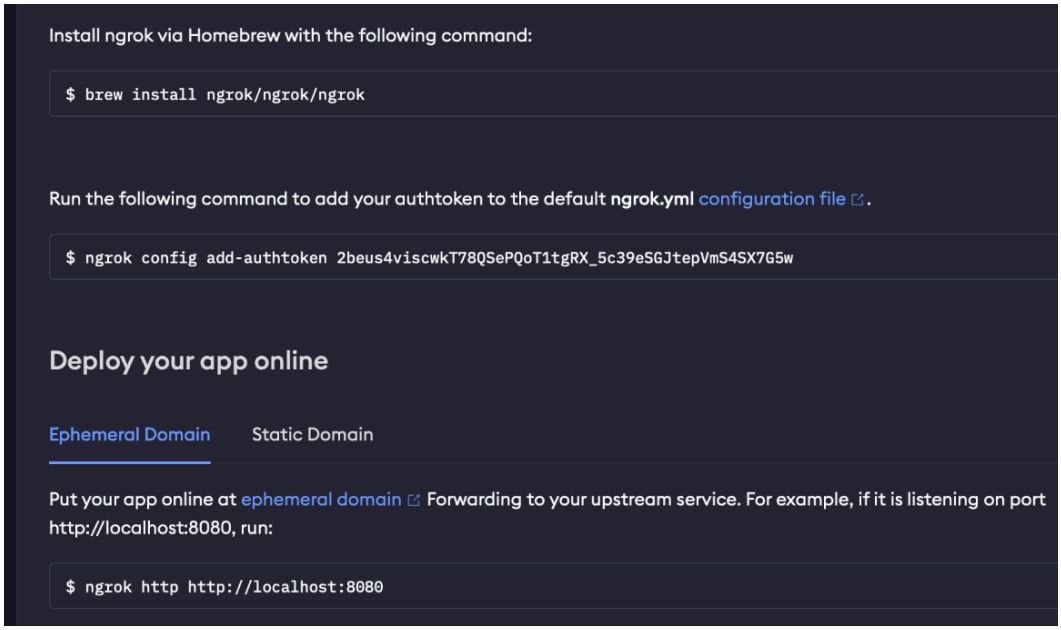
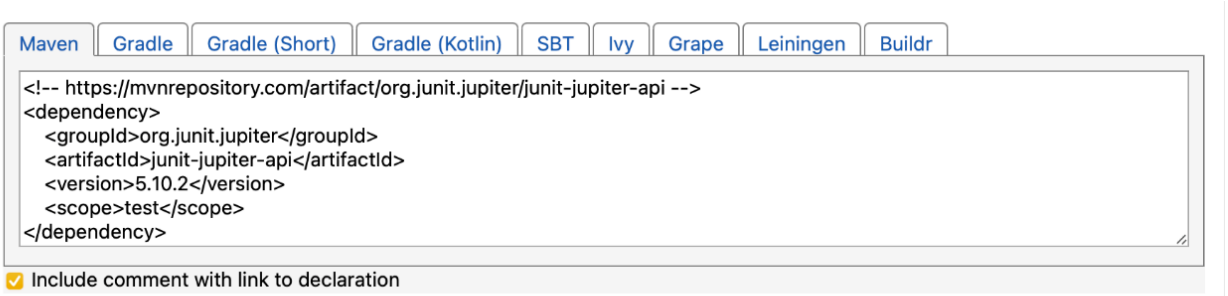


Fig 8:. Ngrok installation

Writing JUnit test cases

Go to https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api/5.10.2 and copy paste this code and add it under your root tag of pom.xml file



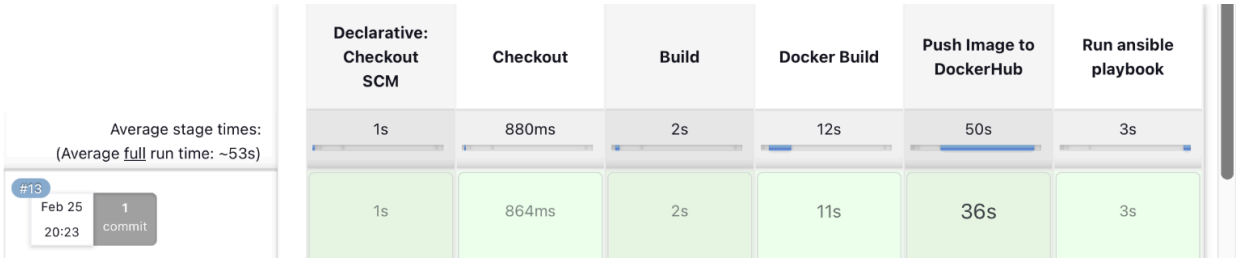
We will be using the tdd **(test driven development)** approach, so it follows Red_Green_Refactor so we will write our test first and then we will add corresponding methods to our file! For example the snippet shows us one of the test cases of a factorial program to find a factorial of zero! It should return 1, but let's write code first run it and then add method to solve it.

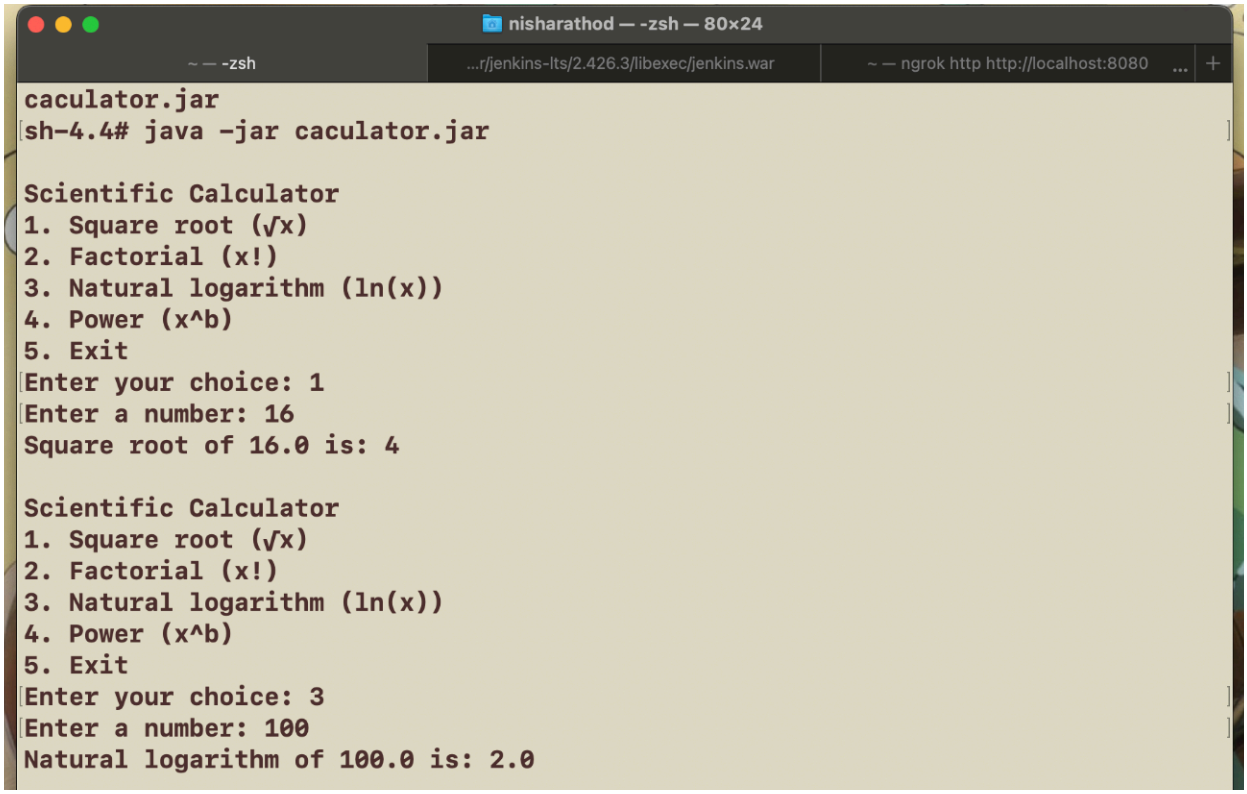
```java
nisharathod2301
@Test
void shouldReturnFactorialZero() {
    //Arrange
    int actualValue = 1;
    int number = 0;
    Calculator calculator = new Calculator();

    //Act
    int expectedValue = calculator.factorial(number);

    //Assert
    assertEquals(expectedValue, actualValue);
}
```

Now that we have a test case, we will add corresponding method to Calculator class:

```java
5 usages   Nisha Rathod +1 *
static int factorial(int number) throws IllegalArgumentException{
    if (number == 0) {
        return 1;
    } else if (number < 0) {
        throw new IllegalArgumentException("Cannot find factorial of a negative number");
    }
    return number * factorial( number: number - 1);
}
```

Similarly we will do for all the functions! After making changes we will push the code on github which will be built automatically by jenkins and we are good to go and run it on our local system! Following snippet shows Jenkins pipeline build and our application output on local host respectively:





Conclusion

In conclusion, this project successfully demonstrates the implementation of a scientific calculator application integrated with a DevOps pipeline. Utilizing a combination of development practices and DevOps tools, the project achieves efficient software delivery and deployment processes. In essence, this project exemplifies the integration of development practices with DevOps principles to deliver a robust, scalable, and efficient scientific calculator application, showcasing the power of automation and collaboration in modern software development workflows.