

Information

- Name: Rohit Shah
 - Roll Number: IMT2021027
 - Github url: <https://github.com/RohitShah1706/calculator-spe.git>
-

Outline

1. Introduction
 - What is DevOps
 - How to implement DevOps
 2. Requirement Analysis
 - Functional Requirements of the Project
 - Non-Functional Requirements of the Project
 3. Tools Used & Setup
 - Java & Maven (`mvn`)
 - Git & GitHub
 - Docker
 - Jenkins
 - Ansible
 4. Credentials Required
 - GitHub Credentials
 - Docker Hub Credentials
 - Google Email Credentials
 5. Configuring Jenkins with credentials
 6. Project Workflow & Setup
 - Create Project using VSCode
 - Build & Test Project using `mvn`
 - Setting Up the Project with Git and GitHub
 - Containerizing the Project Using Docker
 - Jenkins Create New Project
 - Jenkins Pipeline Explanation
 - GitHub Webhook Setup
 - Ansible Setup
-

Introduction

What is DevOps

DevOps is an evolving development paradigm that builds on earlier methodologies like Agile and the now largely obsolete Waterfall model. Over time, these paradigms have helped boost the efficiency and effectiveness of software development.

While the Waterfall model followed a linear, step-by-step process, Agile introduced faster development cycles, enabling teams to move through the programming phase more quickly. But even Agile, despite breaking down some barriers between customers and development teams, keeps phases somewhat isolated.

DevOps takes this a step further by recognizing that software development is not a series of distinct phases, but a continuous and interconnected process. It blurs the lines between teams—whether it's development, operations, or testing—encouraging a more collaborative approach. This way, development isn't just about writing code, but about creating a feedback loop where features are continuously developed, deployed, and improved in real-time, with input from all teams.

The key benefit of DevOps is the seamless integration between development and operations, allowing for continuous development, rapid deployment of new features, and ongoing feedback from users. Meanwhile, the testing team works in parallel, identifying and addressing bugs in real-time.

Ultimately, this approach promotes a harmonious workflow where each team respects and understands the other's responsibilities. This collaboration leads to better productivity and eliminates much of the blame game that can arise when teams work in silos.

How to implement DevOps

To implement DevOps effectively, you need a well-coordinated set of tools that facilitates collaboration across all stakeholders—developers, operations, and testing teams. DevOps relies on automation, continuous testing, and rapid feedback to ensure that new features are quickly validated without breaking existing functionality. Here's a breakdown of the essential tools and practices involved:

1. **IDE (Integrated Development Environment)**: A good IDE supports both version control (like Git) and build tools (such as Maven). Developers use IDEs to write code and, through Maven, automate tasks like building and testing the application. Git integration ensures that multiple developers can work simultaneously without interfering with each other's code.
2. **Maven**: As a build tool, Maven automates the building, packaging, and testing process. It ensures that the project runs consistently across different environments, as long as the dependencies are met.
3. **Git (Version Control System - VCS)**: Git helps developers manage code versions, allowing them to roll back changes if needed and work on different features without disturbing the stable version of the project. Unlike GitHub, which is for remote collaboration, Git is local.
4. **GitHub (Remote VCS & Collaboration)**: For teams working remotely, GitHub acts as a global repository where changes from multiple developers are merged. This makes collaboration seamless and ensures that different parts of the project are integrated properly. Alternatives like GitLab or self-hosted Git servers can also be used for security-sensitive projects.
5. **Jenkins**: Jenkins automates the process of building the project whenever new code is pushed to the repository. By integrating it with GitHub through webhooks, Jenkins kicks off a build as soon as any changes are made, minimizing manual intervention.
6. **Jenkins Pipeline**: With the CI/CD pipeline in Jenkins, the entire process—from building, testing, to deploying—can be automated and monitored. It allows chaining of tasks based on the success or failure of each step, making operations smoother for the team.
7. **Docker**: Docker packages the project into lightweight containers, making it easy to deploy across different environments. Containers are more efficient than virtual machines because they don't require a full OS, making deployments faster and less resource-intensive.
8. **Docker Hub**: Similar to how GitHub stores code, Docker Hub stores container images. Developers can push their containers to Docker Hub, and these images can be pulled from anywhere, making testing and deployment much easier.
9. **Ansible**: Ansible ensures that the environments where your project is deployed are consistent and replicable. It eliminates the problem of "it works on my machine" by providing Infrastructure as a Service (IaaS), so that every deployment has the same setup.

Other important considerations:

- **JDK & JVM**: These are essential for Java-based projects, or other compilers might be needed depending on the language.

- **Tech Stack:** Choose a tech stack like Spring Boot or MERN, depending on the requirements of the project.
- **Team Alignment:** Ensure that development, operations, and testing teams understand and respect each other's roles. Proper training is critical to break down silos.
- **DevOps Suitability:** Not all projects require DevOps. It's important to assess whether DevOps is a good fit, based on project goals and customer requirements.

Requirement Analysis

Functional Requirements of the Project

The project aims to develop a scientific calculator program that supports the following user-driven menu operations:

- **Square root function:** Calculates the square root of a given number x .

$$\sqrt{x}$$

- **Factorial function:** Computes the factorial of a non-negative integer x .

$$x! = x \times (x - 1) \times (x - 2) \times \dots \times 1$$

- **Natural logarithm:** Finds the natural logarithm (base e) of a given number x .

$$\log_b x$$

- **Power function:** Raises a number x to the power b .

$$x^b$$

Each operation should be selectable through a user-friendly menu interface. The program will perform error checking to handle edge cases such as negative inputs for the square root and logarithm functions, or non-integer values for the factorial function.

Non-Functional Requirements of the Project

- **User Assistance:** The program must provide clear and informative help messages and error messages to guide the user through proper usage and error handling.
- **Code Quality:** The code must be clean, well-commented, and easy to understand, ensuring readability for all stakeholders, including developers and future maintainers.
- **Version Control:** Git and GitHub (or an equivalent version control system) must be used to manage and track the project's codebase. This will showcase version control, ensuring proper collaboration and history tracking.
- **Build Automation:** Maven (or an equivalent tool) must be used to automate the building and testing process, ensuring consistent and reliable builds across different environments.
- **Continuous Integration/Continuous Deployment (CI/CD):** Jenkins (or an equivalent CI/CD tool) must be used to automate the build process and create a pipeline that integrates, tests, and deploys the code automatically.
- **Containerization:** Docker (or an equivalent tool) must be used to package the application into containers. The containers should be pushed to a remote repository like Docker Hub (or equivalent) for ease of distribution and deployment.
- **Environment Management:** Ansible (or an equivalent tool) must be employed to pull the remote containers and deploy them in a consistent, repeatable, and configured environment.
- **Documentation:** A well-documented report detailing the project structure, processes, and decisions must be generated for future reference, ensuring clarity for anyone reviewing or continuing the work.

Tools Used & Setup

Since my local machine already had the necessary tools installed, I used a Docker container running `ubuntu:20.04` to demonstrate the installation steps for each tool. The following command was used to spin up the container:

```
docker run --name linux -p 8080:8080 -it ubuntu:20.04 /bin/bash
```

Java & Maven (`mvn`)

Java Installation

Java is a high-level, object-oriented programming language widely used for building platform-independent applications. In this project, Java will be utilized for running backend components and ensuring compatibility with tools like **Maven** for building and automating the project. Specifically, the Java Runtime Environment (JRE) provides the necessary environment to execute Java-based applications and libraries that may be part of the automation pipeline, testing frameworks, or deployment processes.

```
sudo apt update

sudo apt install fontconfig openjdk-17-jre

java -version
openjdk version "17.0.8" 2023-07-18
OpenJDK Runtime Environment (build 17.0.8+7-Debian-1deb12u1)
OpenJDK 64-Bit Server VM (build 17.0.8+7-Debian-1deb12u1, mixed mode, sharing)
```

Maven Installation

Maven is a powerful build automation tool used primarily for Java projects. It simplifies project management by automating tasks like compiling code, running tests, and packaging applications. Maven also handles dependencies by downloading required libraries from central repositories, ensuring that projects have the correct versions of external libraries they rely on.

In this project, Maven will be used to automate the build and testing process. It ensures that all components are compiled and tested consistently across different environments, streamlining the development and deployment workflows as part of the CI/CD pipeline.

Download the Maven Binaries

```
wget https://dlcdn.apache.org/maven/maven-3/3.9.9/binaries/apache-maven-3.9.9-bin.tar.gz

tar -xvf apache-maven-3.9.9-bin.tar.gz

mv apache-maven-3.9.9 /opt/
```

Setting the `M2_HOME` and Path Variables

Add the following lines to the user profile file (`~/.profile`)

```
M2_HOME='/opt/apache-maven-3.9.9'
PATH="$M2_HOME/bin:$PATH"
export PATH
```

Relaunch the terminal or execute `source ~/.profile` to apply the changes.

Verify the `mvn` installation

```
mvn --version
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfc97d260186937)
Maven home: /opt/apache-maven-3.9.9
Java version: 17.0.12, vendor: Ubuntu, runtime: /usr/lib/jvm/java-17-openjdk-amd64
Default locale: en_US, platform encoding: ANSI_X3.4-1968
OS name: "linux", version: "6.5.0-35-generic", arch: "amd64", family: "unix"
```

Git & GitHub

Git is a distributed version control system that allows developers to track changes in their codebase, collaborate with others, and manage multiple versions of a project. It enables developers to work on different features simultaneously, roll back changes, and maintain a history of modifications over time.

GitHub is a cloud-based platform that hosts Git repositories, making it easier for teams to collaborate remotely. It provides additional features like pull requests, issue tracking, and project management tools, all centered around Git-based version control.

In this project, **Git** will be used for local version control to manage code changes, while **GitHub** will serve as the remote repository for collaboration, ensuring that different team members can contribute to the project without conflicts. GitHub will also be integrated with Jenkins to trigger automated builds and deployments.

```
sudo apt install git

git --version
git version 2.25.1
```

Docker

Docker is a platform that enables developers to automate the deployment of applications inside lightweight, portable containers. These containers encapsulate everything needed to run an application, including the code, runtime, libraries, and dependencies, ensuring consistency across different environments.

In this project, Docker will be used to create and manage containers for the scientific calculator application. By containerizing the application, we can easily deploy it across various systems without worrying about environment inconsistencies. Additionally, Docker simplifies the integration with CI/CD pipelines, allowing for efficient testing and deployment of new features while maintaining stability and scalability.

Docker Installation

Uninstall all conflicting packages:

```
for pkg in docker.io docker-doc docker-compose docker-compose-v2 podman-docker containerd runc;
do sudo apt-get remove $pkg; done
```

Set up Docker's `apt` repository.

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

```
# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

Install the Docker packages

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Verify Docker installation

```
sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

Manage Docker as a non-root user

```
# Create the docker group if doesn't exist already
sudo groupadd docker

# Add your user to the docker group
sudo usermod -aG docker $USER

# activate the changes to groups
newgrp docker

# run docker without sudo
docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

Jenkins

Jenkins is an open-source automation server widely used for continuous integration and continuous deployment (CI/CD). It enables developers to automate various stages of the software development lifecycle, such as building, testing, and deploying applications. Jenkins supports a vast ecosystem of plugins, allowing it to integrate seamlessly with many tools and services.

In this project, Jenkins will be utilized to automate the build process of the scientific calculator application. It will monitor the GitHub repository for changes, triggering builds and tests whenever new code is pushed. By setting up a CI/CD pipeline in Jenkins, we can ensure that every change is validated through automated tests, facilitating rapid development and deployment while maintaining high code quality.

Jenkins Installation

Download Long Term Support Release

```
sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \
  https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key

echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc]" \
  https://pkg.jenkins.io/debian-stable binary/ | sudo tee \
  /etc/apt/sources.list.d/jenkins.list > /dev/null
```

```
sudo apt-get update
sudo apt-get install jenkins

# start jenkins service
sudo systemctl start jenkins

# verify jenkins service is running
sudo systemctl status jenkins
• jenkins.service - Jenkins Continuous Integration Server
   Loaded: loaded (/lib/systemd/system/jenkins.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2024-09-22 13:30:27 IST; 30s ago
 Main PID: 37342 (java)
    Tasks: 64 (limit: 8977)
  Memory: 402.5M
     CPU: 27.959s
    CGroup: /system.slice/jenkins.service
            └─37342 /usr/bin/java -Djava.awt.headless=true -jar /usr/share/java/jenkins.war --
webroot=/var/cache/jenkins/war --httpPort=8080
```

Jenkins Setup

Once Jenkins is installed and running, navigate to <http://localhost:8080>

Enter the initial password

Getting Started

Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log ([not sure where to find it?](#)) and this file on the server:

```
/var/lib/jenkins/secrets/initialAdminPassword
```

Please copy the password from either location and paste it below.

Administrator password

```
# get the initial admin password
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
cdc333cac30147899f1f79167b2c7f39
```

Install all the suggested plugins.

Getting Started

Customize Jenkins

Plugins extend Jenkins with additional features to support many different needs.

Install suggested plugins

Install plugins the Jenkins community finds most useful.

Select plugins to install

Select and install plugins most suitable for your needs.

Getting Started

Getting Started

🔄 Folders	🔄 OWASP Markup Formatter	🔄 Build Timeout	🔄 Credentials Binding	** JavaBeans Activation Framework (JAF) API
🔄 Timestamper	🔄 Workspace Cleanup	🔄 Ant	🔄 Gradle	
🔄 Pipeline	🔄 GitHub Branch Source	🔄 Pipeline: GitHub Groovy Libraries	🔄 Pipeline: Stage View	
🔄 Git	🔄 SSH Build Agents	🔄 Matrix Authorization Strategy	🔄 PAM Authentication	
🔄 LDAP	🔄 Email Extension	🔄 Mailer		

Create First Admin User

Getting Started

Create First Admin User

Username:	<input type="text" value="Jenkinsfcc"/>
Password:	<input type="password" value="....."/>
Confirm password:	<input type="password" value="....."/>
Full name:	<input type="text" value="Jenkins FCC"/>
E-mail address:	<input type="text"/>

Ansible

Ansible is an open-source automation tool that simplifies the process of managing and configuring servers and applications. It uses a simple, human-readable language (YAML) to define tasks and playbooks, making it easy to automate deployment, configuration management, and orchestration of services across multiple environments.

In this project, Ansible will be used to pull Docker containers from a remote repository and deploy them in consistent and repeatable environments. By automating the deployment process, Ansible ensures that all instances of the application are configured identically, reducing the risk of errors and simplifying maintenance. This streamlines the overall deployment workflow and helps maintain the integrity of the production environment.

Ansible Installation

```
pip3 install ansible

ansible --version
ansible [core 2.13.13]
  config file = None
  configured module search path = ['/root/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/local/lib/python3.8/dist-packages/ansible
  ansible collection location = /root/.ansible/collections:/usr/share/ansible/collections
  executable location = /usr/local/bin/ansible
  python version = 3.8.10 (default, Sep 11 2024, 16:02:53) [GCC 9.4.0]
  jinja version = 3.1.4
  libyaml = True
```

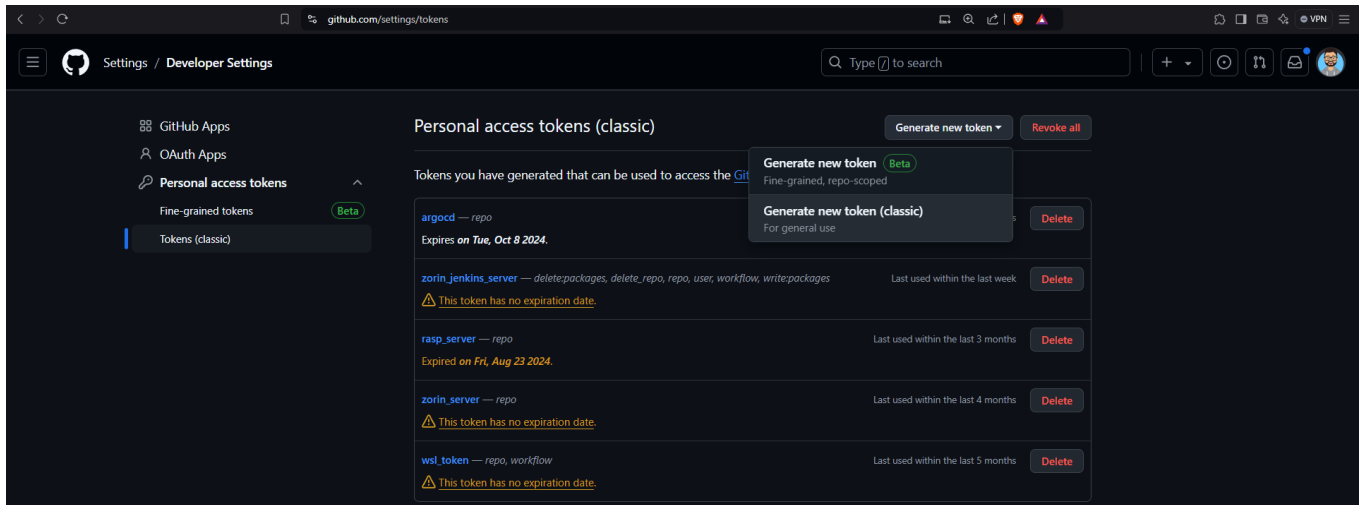
Credentials Required

Before proceeding with the setup and deployment of the project, it is essential to have the following credentials in place. Having these credentials ready will ensure a smooth setup process and enable the successful integration of various tools in the CI/CD pipeline using Jenkins.

GitHub Credentials

- **Username:** Your GitHub username is required for cloning private repositories.
- **Personal Access Token:** A GitHub token must be generated to authenticate and access private repositories. This token replaces your password for operations performed via the Git command line or GitHub API.

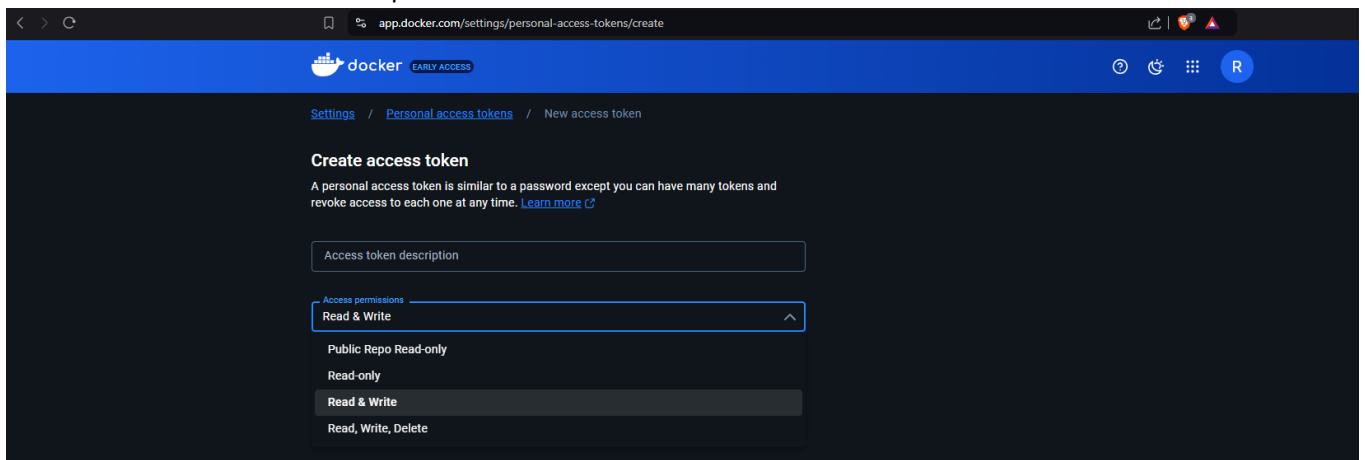
Navigate to <https://github.com/settings/tokens> and generate a new classic token.



Docker Hub Credentials

- **Username:** Your Docker Hub username is necessary for pushing and pulling images to and from private repositories.
- **Password:** The associated password for your Docker Hub account will be required for authentication during these operations.

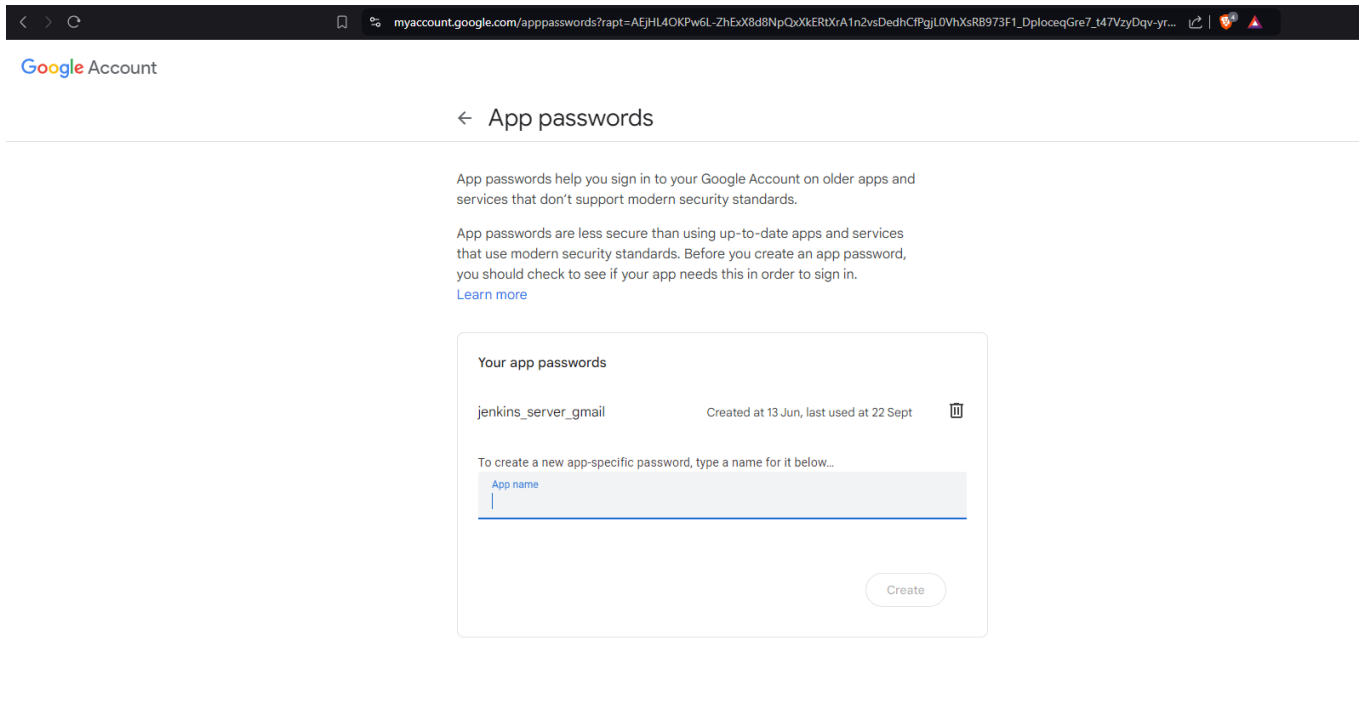
Navigate to <https://app.docker.com/settings/personal-access-tokens/create> and generate a new personal access token with **Read & Write** Access permissions.



Google Email Credentials

- **Email Address:** A valid Google email address is needed to send notifications via email.
- **App Password:** An app-specific password should be created for enhanced security when configuring Jenkins to send emails through Google's SMTP server. This password allows Jenkins to authenticate without using your main Google account password.

Navigate to <https://myaccount.google.com/apppasswords> and create a new app password



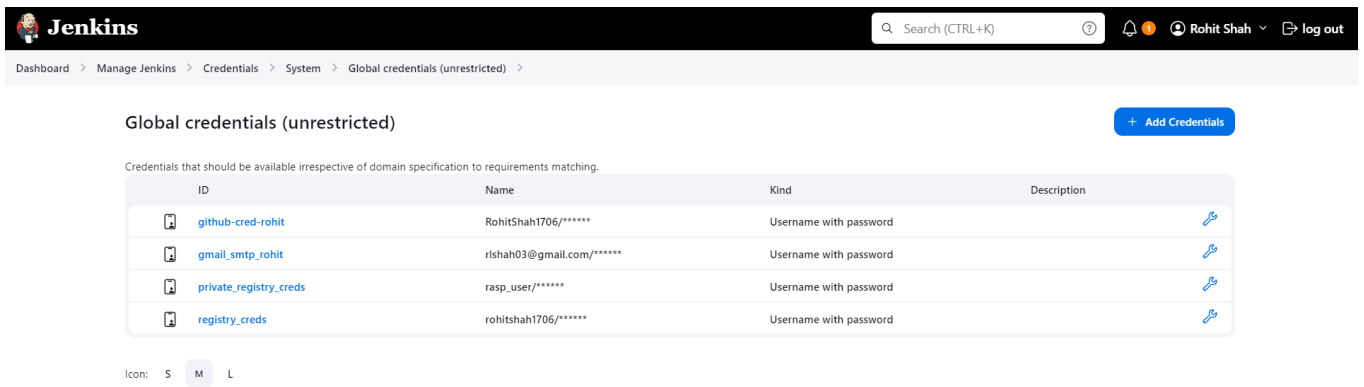
Configuring Jenkins with credentials

To securely manage credentials in Jenkins without hardcoding them in the `Jenkinsfile`, we can configure Jenkins to store and use the necessary credentials. Below are the credential IDs we will create in Jenkins for various services:

- **GitHub Credentials:**
 - **Credential ID:** `github-cred-rohit`
 - **Description:** GitHub username & personal access token
 - **Usage:** This credential will be used for cloning private repositories from GitHub in our Jenkins pipeline.
- **Docker Hub Credentials:**
 - **Credential ID:** `private_registry_creds`
 - **Description:** Docker Hub username & password
 - **Usage:** This credential will be utilized when pushing and pulling Docker images from private repositories on Docker Hub as part of the CI/CD pipeline.
- **Google SMTP Credentials:**
 - **Credential ID:** `gmail_smtp_rohit`
 - **Description:** Google email address & app password
 - **Usage:** These credentials will be used to configure Jenkins to send email notifications about the build status via Google's SMTP server, ensuring secure and authenticated communication.

By setting up these credentials in Jenkins, we eliminate the need to hardcode sensitive information in the `Jenkinsfile`. Instead, these credentials can be accessed using the credential IDs, reducing the risk of exposing sensitive information in GitHub repositories or other environments.

Navigate to `Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted)` and click on `Add Credentials`



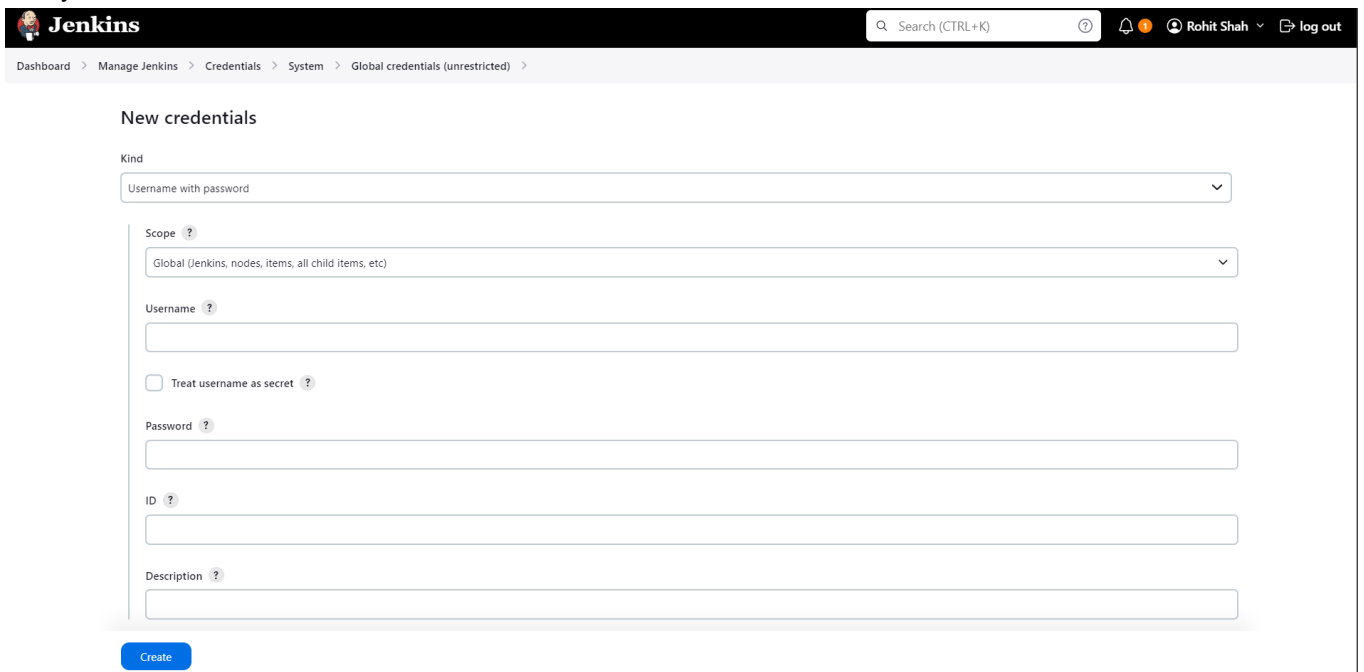
Global credentials (unrestricted) [+ Add Credentials](#)

Credentials that should be available irrespective of domain specification to requirements matching.

ID	Name	Kind	Description
github-cred-rohit	RohitShah1706/*****	Username with password	link
gmail_smtp_rohit	rishah03@gmail.com/*****	Username with password	link
private_registry_creds	rasp_user/*****	Username with password	link
registry_creds	rohitshah1706/*****	Username with password	link

Icon: S M L

Add your details and hit save.



New credentials

Kind: Username with password

Scope: Global (Jenkins, nodes, items, all child items, etc)

Username:

☐ Treat username as secret

Password:

ID:

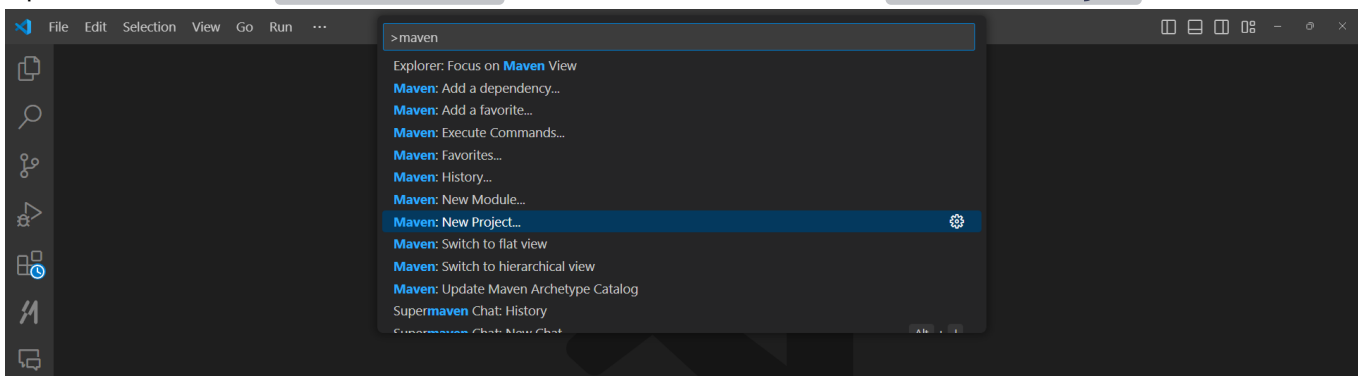
Description:

[Create](#)

Project Workflow & Setup

Create project using VSCode

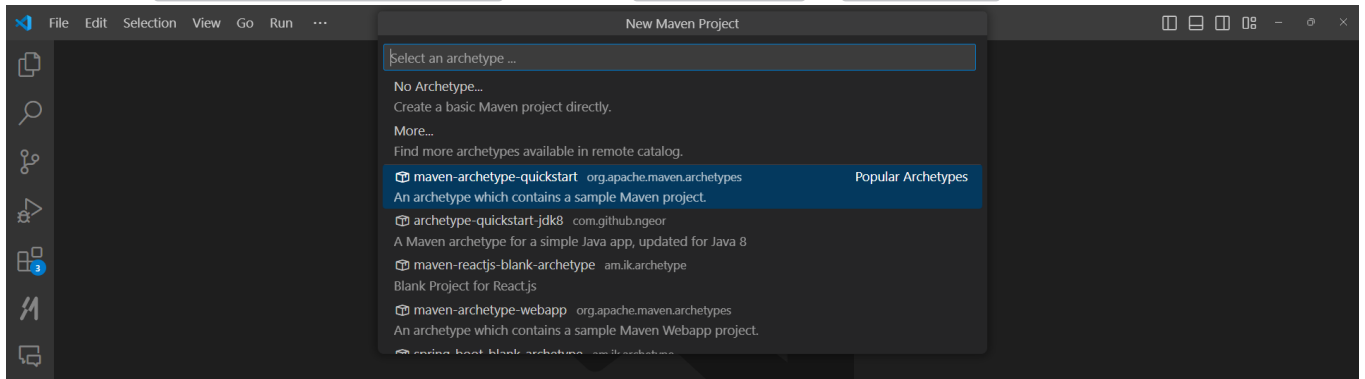
Open **VSCode** and hit **Ctrl + Shift + P**. Search for maven and click on **Maven: New Project**



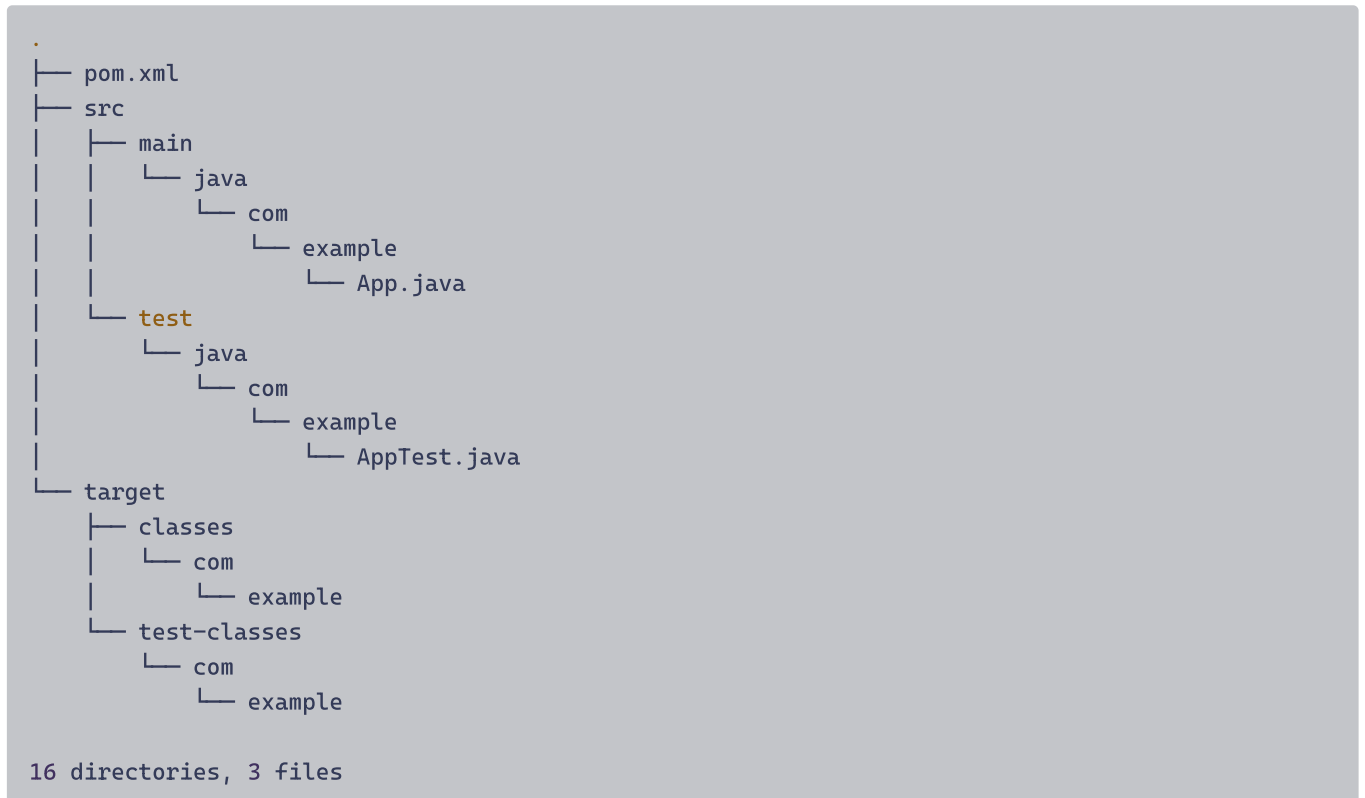
VSCode interface showing the command palette with the search term **>maven**. The list of commands includes:

- Explorer: Focus on **Maven** View
- Maven**: Add a dependency...
- Maven**: Add a favorite...
- Maven**: Execute Commands...
- Maven**: Favorites...
- Maven**: History...
- Maven**: New Module...
- Maven: New Project...** (highlighted with a gear icon)
- Maven**: Switch to flat view
- Maven**: Switch to hierarchical view
- Maven**: Update Maven Archetype Catalog
- Super**maven** Chat: History
- Super**maven** Chat: New Chat

Then select `maven-archetype-quickstart` and enter `package name` & `artifact id` when prompted.

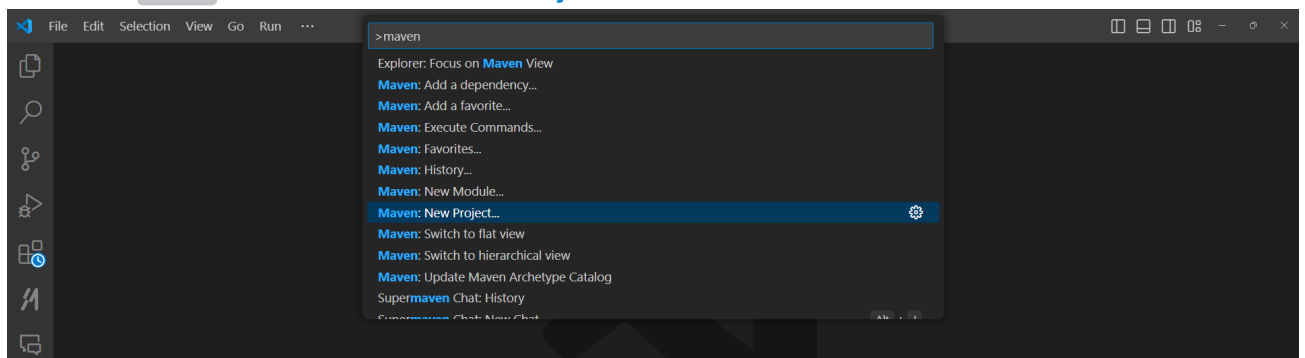


This will generate a blank project with some boilerplate code. The project structure looks like following.

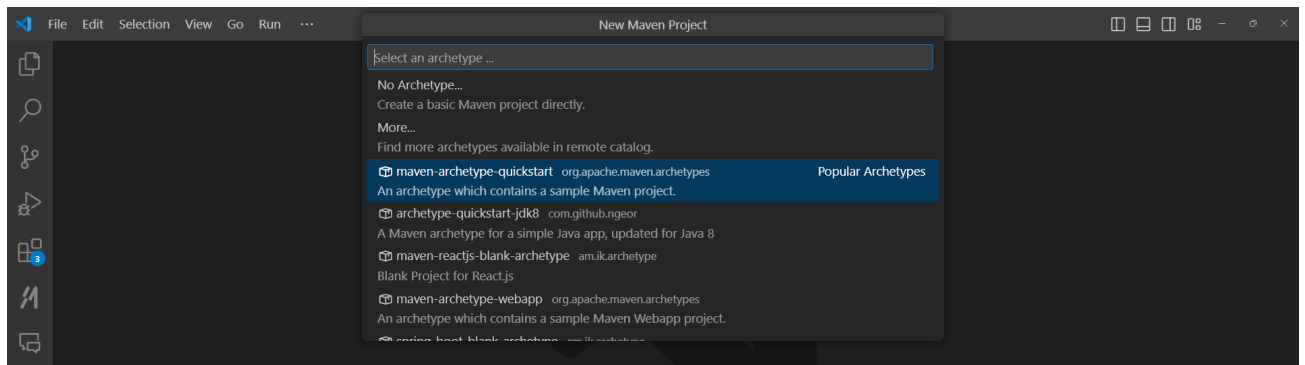


To begin setting up the project, follow these steps to create a Maven project:

1. **Open VSCode** and press `Ctrl + Shift + P` to open the command palette.
2. Search for `Maven` and select **Maven: New Project** from the list.

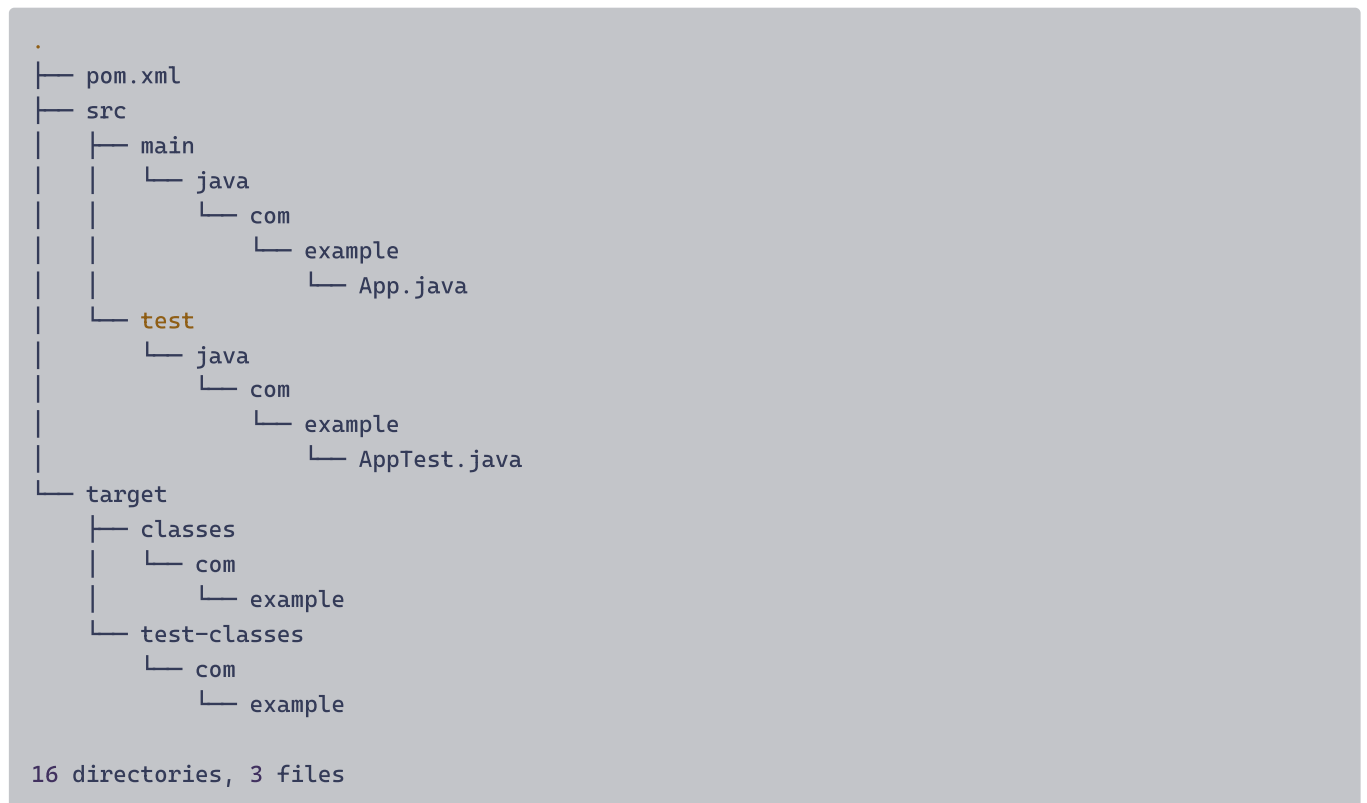


3. In the next step, select `maven-archetype-quickstart` from the available archetypes to generate a simple Java project structure.



4. When prompted, enter your desired **Package Name** and **Artifact ID** for the project.

This will generate a basic Maven project with some initial boilerplate code, which can be customized later. The structure of the generated project will look like the following:



This setup gives you a basic Java application with a corresponding test class, allowing you to begin building your scientific calculator project while following best practices for project structure and testing.

Directory/File	Description
<code>pom.xml</code>	The Project Object Model (POM) file that contains the project's configuration, dependencies, and build instructions.
<code>src/main/java/com/example</code>	The directory where your main application code resides.
<code>App.java</code>	The main class of the project. It contains the core logic of your application.
<code>src/test/java/com/example</code>	The directory for test classes. This is where unit tests for the application are written.
<code>AppTest.java</code>	The test class corresponding to <code>App.java</code> , used for testing the functionality of your application.
<code>target/classes</code>	The directory where compiled classes are stored after building the project.
<code>target/test-classes</code>	The directory where compiled test classes are stored after building the project.

Build & Test Project using `mvn`

Once your Maven project is set up, you can start building and testing it using Maven commands. This section will guide you through the steps required to build and test your project.

1. Build the Project

To build the project, Maven compiles your source code and packages it into a distributable format (such as a `.jar` file). Maven also resolves dependencies specified in your `pom.xml`.

Command to build the project:

```
mvn clean install
```

Breakdown of the Command:

- `mvn`: This invokes the Maven tool.
- `clean`: This phase removes all previously compiled files and ensures a fresh build.
- `install`: This phase compiles the code, runs tests, and packages the application. The final product is stored in the local Maven repository for reuse.

Upon running this command, you'll see output showing Maven downloading dependencies, compiling the source files, and packaging the project into a `target/` directory. The final `.jar` file or `.war` file will be created in `target/`.

2. Test the Project

Testing is a critical part of the development process. Maven makes it easy to run unit tests using the `mvn test` command. By default, Maven will automatically execute any tests located in the `src/test/java` directory.

Command to run the tests:

```
mvn test
```

Breakdown of the Command:

- `test`: This phase compiles the test code, executes the unit tests defined in the `src/test/java/` directory, and reports the results.

Maven uses a testing framework like JUnit (if it's configured in your `pom.xml`) to run the tests. After executing the tests, Maven will display the results, including any test failures or errors.

3. Clean the Project

Before running a new build, it is good practice to clean your project to remove any leftover compiled files from previous builds. Maven provides a `clean` command to achieve this.

Command to clean the project:

```
mvn clean
```

Breakdown of the Command:

- `clean`: This goal removes the `target/` directory where compiled files and the packaged project reside.

Running this command ensures that the next build starts from a clean state, avoiding potential issues caused by stale files.

4. Full Lifecycle Command

You can combine the clean, build, and test phases into one command for simplicity:

```
mvn clean install
```

This command will clean the project, build it, and run the tests in a single step.

```
calculator ➤ mvn clean install -DskipTests
[INFO] Scanning for projects...
[INFO]
[INFO] -----< tech.rohitshah1706:calculator >-----
[INFO] Building calculator 1.0.0
[INFO] from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-clean-plugin:3.3.2:clean (default-clean) @ calculator ---
[INFO] Deleting C:\D_DRIVE\SEM7\SPE\Labs\calculator\target
[INFO]
[INFO] --- maven-resources-plugin:3.3.1:resources (default-resources) @ calculator ---
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO] Copying 0 resource from src/main/resources to target/classes
[INFO]
[INFO] --- maven-compiler-plugin:3.13.0:compile (default-compile) @ calculator ---
[INFO] Recompiling the module because of changed source code.
[INFO] Compiling 2 source files with javac [debug parameters release 17] to target\classes
[INFO]
[INFO] --- maven-resources-plugin:3.3.1:testResources (default-testResources) @ calculator ---
[INFO] skip non existing resourceDirectory C:\D_DRIVE\SEM7\SPE\Labs\calculator\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.13.0:testCompile (default-testCompile) @ calculator ---
[INFO] Recompiling the module because of changed dependency.
[INFO] Compiling 1 source file with javac [debug parameters release 17] to target\test-classes
[INFO]
[INFO] --- maven-surefire-plugin:3.2.5:test (default-test) @ calculator ---
[INFO] Tests are skipped.
[INFO]
[INFO] --- maven-jar-plugin:3.4.2:jar (default-jar) @ calculator ---
[INFO] Building jar: C:\D_DRIVE\SEM7\SPE\Labs\calculator\target\calculator-1.0.0.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:3.3.3:repackage (repackage) @ calculator ---
[INFO] Replacing main artifact C:\D_DRIVE\SEM7\SPE\Labs\calculator\target\calculator-1.0.0.jar with repackaged archive, adding nested dependencies in BOOT-INF/.
[INFO] The original artifact has been renamed to C:\D_DRIVE\SEM7\SPE\Labs\calculator\target\calculator-1.0.0.jar.original
[INFO]
[INFO] --- maven-install-plugin:3.1.3:install (default-install) @ calculator ---
[INFO] Installing C:\D_DRIVE\SEM7\SPE\Labs\calculator\pom.xml to C:\Users\risha.m2\repository\tech\rohitshah1706\calculator\1.0.0\calculator-1.0.0.pom
[INFO] Installing C:\D_DRIVE\SEM7\SPE\Labs\calculator\target\calculator-1.0.0.jar to C:\Users\risha.m2\repository\tech\rohitshah1706\calculator\1.0.0\calculator-1.0.0.jar
[INFO]
[INFO] BUILD SUCCESS
[INFO]
```

5. Verify the Build and Test Results

After building the project, you can navigate to the `target/` directory to verify the compiled output. If there are any test failures, they will be displayed, and the build will fail. In addition to the compiled `.class` files, you will see:

- The final packaged `.jar` or `.war` file (depending on your project type).
- A `test-classes` directory containing compiled test code.
- A `surefire-reports/` directory containing reports on the test results.

6. Run the JAR File

Once your Maven project has been successfully built, you can run the packaged JAR file that was generated in the `target/` directory. This is the executable version of your application.

In your `pom.xml`, based on the following configuration:

```
<groupId>tech.rohitshah1706</groupId>
<artifactId>calculator</artifactId>
<version>1.0.0</version>
<name>calculator</name>
```

Maven will generate a JAR file named `calculator-1.0.0.jar` inside the `target/` directory.

Command to run the JAR file:

```
java -jar target/calculator-1.0.0.jar
```

Breakdown of the Command:

- `java`: This invokes the Java runtime.
- `-jar`: This option specifies that you want to run a JAR file.

- `target/calculator-1.0.0.jar`: This is the path to the JAR file generated by Maven.

```
calculator java -jar .\target\calculator-1.0.0.jar
=== Calculator Menu ===
1. Square Root
2. Factorial
3. Natural Logarithm
4. Power
5. Exit
Select an option: 2
Enter a number: 5
Factorial: 120

=== Calculator Menu ===
1. Square Root
2. Factorial
3. Natural Logarithm
4. Power
5. Exit
Select an option: 1
Enter a number: 16
Square root: 4.0

=== Calculator Menu ===
1. Square Root
2. Factorial
3. Natural Logarithm
4. Power
5. Exit
Select an option: 5
Exiting...
```

Setting Up the Project with Git and GitHub

In this section, we will walk through the steps required to initialize a Git repository for your project, connect it to a GitHub repository, and push your code to the remote repository.

1. Initialize Git in Your Project

Before using Git to track your project's files, you need to initialize a local Git repository in your Maven project folder.

Steps:

1. Navigate to the root directory of your Maven project in the terminal:

```
cd /path/to/your/maven/project
```

2. Initialize the Git repository: This command sets up a new Git repository in your project folder. It creates a hidden `.git/` directory to track changes in your files.

```
git init
```

2. Create a `.gitignore` File

To prevent unnecessary files from being tracked by Git (such as compiled files, temporary files, or IDE-specific configurations), you should create a `.gitignore` file.

Example `.gitignore` file:

```
HELP.md
target/
!.mvn/wrapper/maven-wrapper.jar
!**/src/main/**/target/
!**/src/test/**/target/

### IntelliJ IDEA ###
.idea
*.iws
*.iml
*.ipr

build/
!**/src/main/**/build/
!**/src/test/**/build/

### VS Code ###
.vscode/
```

Add this file to your project's root directory.

3. Stage and Commit Changes

Now that Git is initialized and you've specified the files to ignore, you can stage and commit the project files.

Steps:

1. Stage all files for commit:

```
git add .
```

2. Commit the staged files with a message:

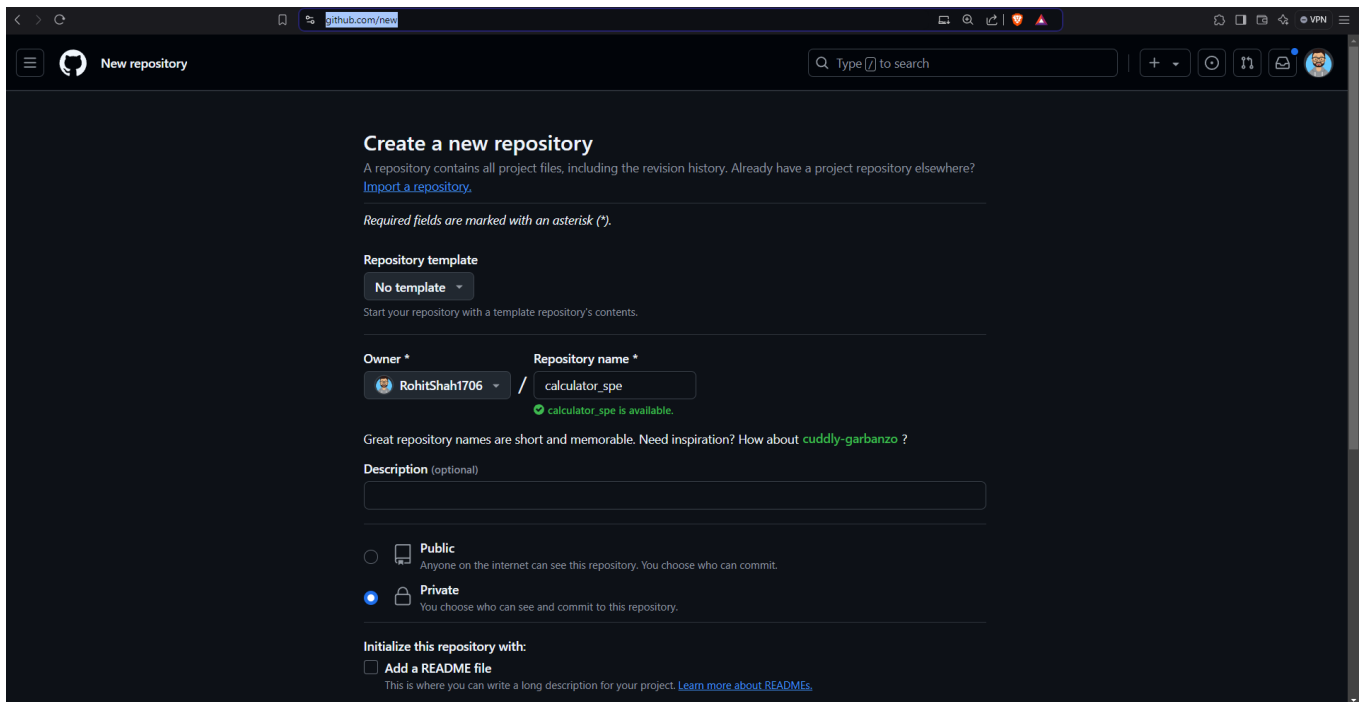
```
git commit -m "Initial commit for Maven project"
```

4. Create a GitHub Repository

To store your project remotely and collaborate with others, create a GitHub repository.

Steps:

1. Log in to your GitHub account.
2. Navigate to <https://github.com/new>
3. Fill in the repository details:
 - **Repository Name:** For example, `calculator-spe`.
 - **Description:** Optionally, add a description for your project.
 - **Visibility:** Choose between Public or Private, depending on your preference.
4. Click **Create Repository**.



GitHub will provide instructions on how to connect your local Git repository to this new GitHub repository.

5. Connect Local Repository to GitHub

Once the GitHub repository is created, you need to connect your local repository to the remote GitHub repository.

Steps:

1. Add the GitHub repository URL as the `origin` remote:

```
git remote add origin https://github.com/your-username/your-repo-name.git
```

2. Push the local repository to GitHub:

```
git push -u origin main
```

Collaborators can now:

- Make changes.
- Push updates back to the remote repository.
- Collaborate on features, fixes, and enhancements using GitHub's pull request feature.

Containerizing the Project Using Docker

In this section, we will go through the process of containerizing the Maven project using Docker. Containerization enables the application to run in a consistent environment, regardless of where it is deployed. This section includes creating a `Dockerfile`, understanding each step, the concept of multi-stage builds, and how to push the Docker image to a private Docker Hub repository.

1. Create `Dockerfile`

The `Dockerfile` is a set of instructions used by Docker to build a container image. Below is the reference `Dockerfile` for your Maven-based calculator project.

```
# Stage 1: Build the application
FROM maven:3.8.5-openjdk-17 AS build
```

```

# Set the working directory inside the container
WORKDIR /app

# Copy the pom.xml and download the dependencies
COPY pom.xml ./

# Copy the entire project source code
COPY ./src /app/src

# Package the application
RUN mvn clean install -DskipTests

# Stage 2: Run the application
FROM openjdk:17-jdk-slim AS run

# Set the working directory inside the container
WORKDIR /app

# Copy the jar file from the build stage
COPY --from=build /app/target/*.jar ./app.jar

# Command to run the application
ENTRYPOINT ["java", "-jar", "app.jar"]

```

2. Step-by-Step Explanation of Dockerfile

Stage 1: Build the Application

- Base Image:** `FROM maven:3.8.5-openjdk-17 AS build` --> This stage uses the official Maven image with OpenJDK 17. It provides all the necessary tools to build the Java application, including Maven and the required JDK.
- Set Working Directory:** `WORKDIR /app` --> Sets the working directory inside the container to `/app`, where all the subsequent actions (copying, building, etc.) will take place.
- Copy pom.xml and Download Dependencies:** `COPY pom.xml ./` --> Copies the `pom.xml` file into the container. This allows Maven to download the dependencies required for the project before copying the source code.
- Copy the Entire Project Source Code:** `COPY ./src ./app/src` --> Copies the source code from your local machine to the container's `/app/src` directory.
- Build the application:** `RUN mvn clean install -DskipTests` --> This command builds the project using Maven and skips running tests. It compiles the source code, packages it, and places the output `.jar` file in the `target/` directory.

Stage 2: Run the Application

- Base Image for Running:** `FROM openjdk:17-jdk-slim AS run` --> A much lighter image with only the essential runtime environment (JDK) for running the Java application.
- Set Working Directory:** `WORKDIR /app` --> -- Again, set the working directory to `/app`, where the `.jar` file will be placed.
- Copy the Jar File from the Build Stage:** `COPY --from=build /app/target/*.jar ./app.jar` --> Copies the packaged `.jar` file from the first stage (build) into the second stage (run) under `/app/app.jar`.
- Command to Run the Application:** `ENTRYPOINT ["java", "-jar", "app.jar"]` --> The `ENTRYPOINT` directive sets the command to run the Java application inside the container.

3. Multi-Stage Docker Builds & Their Advantages

This `Dockerfile` uses a **multi-stage build**. In multi-stage builds, different stages are used for different purposes (building, testing, running, etc.), which brings several advantages:

- **Reduced Image Size:** Only the final stage (`run`) is included in the image. The intermediate stages (e.g., `build`) are discarded, keeping the final image lightweight. This is crucial when deploying to production environments where minimal dependencies are preferred.
- **Separation of Concerns:** The build stage includes all the tools (Maven, compilers, etc.) necessary for building the project, while the run stage only contains the essentials (JRE) for executing the packaged `.jar`. This clear separation ensures that the final image is secure and free of unnecessary build tools.
- **Faster Builds:** By separating the build and run stages, the build cache is more effectively utilized. For example, copying `pom.xml` separately ensures dependencies are only downloaded if there are changes in the `pom.xml` file.

4. Create `.dockerignore` File

Before building the Docker image, it's important to create a `.dockerignore` file. This file tells Docker which files and directories to exclude when building the image. It helps reduce the image size and prevents unnecessary files (e.g., local configurations, build files) from being copied into the image.

Create a `.dockerignore` file in the root directory of your project with the following contents:

```

**/.classpath
**/.dockerignore
**/.env
**/.git
**/.gitignore
**/.project
**/.settings
**/.toolstarget
**/.vs
**/.vscode
**/.next
**/.cache
**/*.proj.user
**/*.dbmdl
**/*.jfm
**/charts
**/docker-compose*
**/compose.yaml
**/target
**/Dockerfile*
**/node_modules
**/npm-debug.log
**/obj
**/secrets.dev.yaml
**/values.dev.yaml
**/vendor
LICENSE
README.md

```

5. Push to Docker Hub Repository

Once you've built the Docker image locally, the next step is to push it to a private repository on Docker Hub.

Steps to Push Docker Image:

1. **Build the Docker Image:** Run the following command to build the Docker image:

```
docker build -t your-dockerhub-username/calculator:1.0.0 .
```

2. **Login to Docker Hub:** Use the following command to log in to your Docker Hub account:

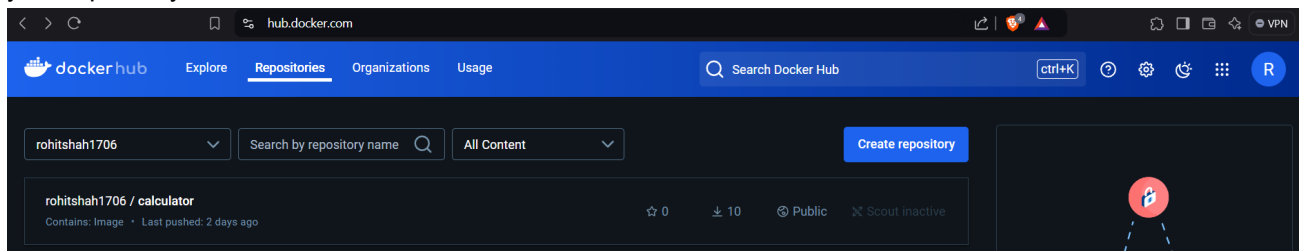
```
docker login --username your-dockerhub-username
```

- You'll be prompted to enter your Docker Hub password.
- Enter the dockerhub personal access token we created in the **Credentials Required** section.

3. **Push the Image to Docker Hub:** Once logged in, push the image to your private Docker Hub repository:

```
docker push your-dockerhub-username/calculator:latest
```

4. **Verify the Push:** Log in to Docker Hub and verify that your image `calculator:1.0.0` has been uploaded to your repository.



By pushing the image to Docker Hub, the containerized application can be easily pulled and deployed on any server or environment that supports Docker.

Jenkins Create New Project

NOTE: Make sure you followed the **Credentials Required** & **Configuring Jenkins with credentials** section before starting this section.

Setting up a Jenkins pipeline for continuous integration and continuous deployment (CI/CD) involves creating a new pipeline in Jenkins that will automatically test, build, and deploy your project. Here's how you can set it up:

1. Navigate and Create a New Pipeline

1. **Open Jenkins Dashboard:** Start by logging into your Jenkins instance.
2. **Create New Pipeline:**
 - Click on **"New Item"** from the Jenkins dashboard.
 - You will be prompted to give the project a name. Choose a descriptive name for your pipeline (e.g., `calculator-spe`).
 - Select **"Pipeline"** as the item type and click **"OK"**.

New Item

Enter an item name

calculator_spe

Select an item type

- Freestyle project**
Classic, general-purpose job type that checks out from up to one SCM, executes build steps serially, followed by post-build steps like archiving artifacts and sending email notifications.
- Pipeline**
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.
- Multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.
- Folder**
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.
- Multibranch Pipeline**
Creates a set of Pipeline projects according to detected branches in one SCM repository.

OK

2. Configure the Pipeline

After creating the pipeline, you'll need to configure it.

1. Build Trigger:

- Scroll down to the **"Build Triggers"** section.
- Select **"GitHub hook trigger for GITScm polling"**. This will allow Jenkins to start a build whenever there's a new commit pushed to the repository.
- Later, we will configure a GitHub Webhook to notify Jenkins about repository changes.

Dashboard > calculator-spe > Configuration

Configure

- General
- Advanced Project Options
- Pipeline

☐ Throttle builds ?

Build Triggers

☐ Build after other projects are built ?

☐ Build periodically ?

☒ GitHub hook trigger for GITScm polling ?

☐ Poll SCM ?

☐ Quiet period ?

☐ Trigger builds remotely (e.g., from scripts) ?

2. Pipeline Script from SCM:

- Scroll down to the **"Pipeline"** section and select **"Pipeline script from SCM"**. This tells Jenkins that the pipeline definition will be stored in the source code management (SCM) system, such as GitHub.
- In **SCM**, choose **"Git"**. Enter the **Repository URL** for your project's GitHub repository.
- In the **Credentials** dropdown, choose the **GitHub credentials** that we configured earlier (e.g., `github-cred-rohit`). This ensures secure access to your private GitHub repository without exposing credentials in the pipeline script.
- In the **Script Path** field, provide the path to your `Jenkinsfile` if it's not located at the root directory (default is `Jenkinsfile`).

Dashboard > calculator-spe > Configuration

Configure

- General
- Advanced Project Options
- Pipeline**

Definition: Pipeline script from SCM

SCM: Git

Repositories:

- Repository URL:
- Credentials:
- + Add
- Advanced

Add Repository

Branches to build:

Branch Specifier (blank for 'any'):

Save Apply

3. Save and Apply:

- After filling in all the required fields, click **"Save"**.
- Click **"Apply"** to finalize the setup.

Jenkins Pipeline Explanation

This section explains the Jenkins pipeline defined in the `Jenkinsfile`. The pipeline consists of various stages that automate the testing, build, Docker image creation, and deployment of the project, while also sending notifications about the build status via email.

Pipeline Overview

The pipeline is structured as follows:

- **Agent:** The pipeline is set to run on any available agent (`agent any`).
- **Environment Variables:** The environment variable `IMAGE_NAME` is defined to store the Docker image name (`rohitshah1706/calculator`).

The pipeline consists of multiple stages, each performing a specific task in the CI/CD process.

```

pipeline {
    agent any

    environment {
        IMAGE_NAME = 'rohitshah1706/calculator'
    }

    stages {

        stage('Check tools') {
            steps {
                echo 'Checking Docker and Docker Compose versions...'
                sh '''
                    docker version
                    docker compose version
                '''
            }
        }
    }
}

```



```

}

stage('Run tests') {
    steps {
        sh 'mvn test'
    }
}

stage('Build Docker Image') {
    steps {
        echo 'Building Docker image...'
        script {
            docker.build("${IMAGE_NAME}:latest")
        }
    }
}

stage('Login to DockerHub') {
    steps {
        echo 'Logging in to DockerHub...'
        script {
            docker.withRegistry('https://index.docker.io/v1/', 'registry_creds') {
                // Docker registry login happens here
            }
        }
    }
}

stage('Push Docker Image to Registry') {
    steps {
        echo 'Pushing Docker image to DockerHub...'
        script {
            docker.withRegistry('https://index.docker.io/v1/', 'registry_creds') {
                docker.image("${IMAGE_NAME}:latest").push()
            }
        }
    }
}

stage("Deploy via ansible") {
    steps {
        ansiblePlaybook(
            installation: 'Ansible',
            inventory: 'inventory/inventory.ini',
            playbook: 'playbook/deploy.yml',
        )
    }
}

post {
    always {
        // send email notification with build status
        script {
            def jobName = env.JOB_NAME
            def buildNumber = env.BUILD_NUMBER
            def pipelineStatus = currentBuild.result ?: 'UNKNOWN'
            def bannerColor = pipelineStatus.toUpperCase() == 'SUCCESS' ? 'green' : 'red'

            def body = ""

```

```

        <html>
        <body>
            <div style="border: 4px solid ${bannerColor}; padding: 10px;">
                <h2>${jobName} - Build ${buildNumber}</h2>
                <div style="background-color: ${bannerColor}; padding: 10px;">
                    <h3 style="color: white;">
                        Pipeline Status: ${pipelineStatus.toUpperCase()}
                    </h3>
                </div>
                <p>
                    Check the <a href="${BUILD_URL}/console">console output</a>
                </p>
            </div>
        </body>
    </html>
    """

    emailExt(
        subject: "${jobName} - Build ${buildNumber} -
${pipelineStatus.toUpperCase()}",
        body: body,
        to: "rlshah03@gmail.com",
        from: "rlshah03@gmail.com",
        replyTo: "rlshah03@gmail.com",
        mimeType: 'text/html'
    )
}
}
}
}
}

```

1. Check tools Stage

- **Purpose:** This stage verifies that Docker and Docker Compose are installed and accessible on the agent machine.
- **Steps:**
 - The `sh` shell command is used to check the versions of Docker and Docker Compose.
 - This step ensures that the necessary containerization tools are installed before proceeding with the build.

2. Run tests Stage

- **Purpose:** This stage runs the unit tests to verify the functionality of the application.
- **Steps:**
 - The `mvn test` command is executed to run the Maven tests defined in the project.
 - If the tests fail, the pipeline will stop here, ensuring that no further steps are taken with a potentially broken build.

3. Build Docker Image Stage

- **Purpose:** This stage builds the Docker image for the project.
- **Steps:**
 - The Docker image is built using the `docker.build()` function, and it is tagged with the latest version (`latest`).

4. Login to DockerHub Stage

- **Purpose:** This stage logs in to DockerHub using previously configured credentials.
- **Steps:**
 - The `docker.withRegistry()` function is used to authenticate with DockerHub using the credentials `registry_creds`.
 - This step ensures that we can push the Docker image to a private DockerHub repository securely.

5. Push Docker Image to Registry Stage

- **Purpose:** This stage pushes the Docker image built in the previous stage to the DockerHub repository.
- **Steps:**
 - After logging in to DockerHub, the `docker.image().push()` function pushes the image to the specified repository.
 - This allows the Docker image to be stored in DockerHub, making it accessible for deployment.

6. Deploy via Ansible Stage

- **Purpose:** This stage uses Ansible to deploy the application in a configured environment.
- **Steps:**
 - The `ansiblePlaybook` function runs the Ansible playbook (`deploy.yml`) defined in the project.
 - Ansible is used to automate the deployment process by managing configurations and deploying the Docker container in a consistent and repeatable manner.
 - **NOTE:** Refer the [Ansible Setup](#) section

Post-Build Actions


After the pipeline finishes running, whether it succeeds or fails, the following post-build steps are executed:

- **Always Send Email Notification:**
 - An HTML-formatted email is sent to notify the user about the build result.
 - The build status (success or failure) is highlighted, and a link to the Jenkins console output is provided.

Trigger Pipeline

You can trigger pipeline by clicking the [Build Now](#) button. In the [GitHub Webhook Setup](#) section we will configure Github to automatically trigger Jenkins pipeline when a new push is made to the main branch of our github

repository.

 **Jenkins**

Dashboard > calculator-spe >

Status

</> Changes

▶ Build Now

⚙️ Configure

🗑️ Delete Pipeline

📁 Stages

✎ Rename

❓ Pipeline Syntax

📄 GitHub Hook Log

✅ calculator-spe

Permalinks

- [Last build \(#5\), 1 day 23 hr ago](#)
- [Last stable build \(#5\), 1 day 23 hr ago](#)
- [Last successful build \(#5\), 1 day 23 hr ago](#)
- [Last completed build \(#5\), 1 day 23 hr ago](#)

🌞 Build History

trend ▾

🔍 Filter... /

✅ #5

Sep 22, 2024, 10:51 AM

✅ #4

Sep 19, 2024, 11:30 AM

✅ #3

Sep 19, 2024, 10:37 AM

📡 Atom feed for all

📡 Atom feed for failures

Jenkins Pipeline Stage View

✅ calculator-spe

Add description

Stage View

Average stage times:
(Average full run time: ~57s)

#5

Sep 22 10:51

1 commit

Declarative: Checkout SCM	Check tools	Run tests	Build Docker Image	Login to DockerHub	Push Docker Image to Registry	Deploy via ansible	Declarative: Post Actions
1s	608ms	3s	19s	3s	19s	4s	4s
1s	676ms	3s	4s	4s	10s	4s	4s

Email Notification after each build

calculator-spe - Build 5 - SUCCESS

Inbox x

 rishah03@gmail.com

Sun, 22 Sept, 10:51 (2 days ago)

☆ 😊 ↶ ⋮

calculator-spe - Build 5

Pipeline Status: SUCCESS

Check the [console output](#)

GitHub Webhook Setup

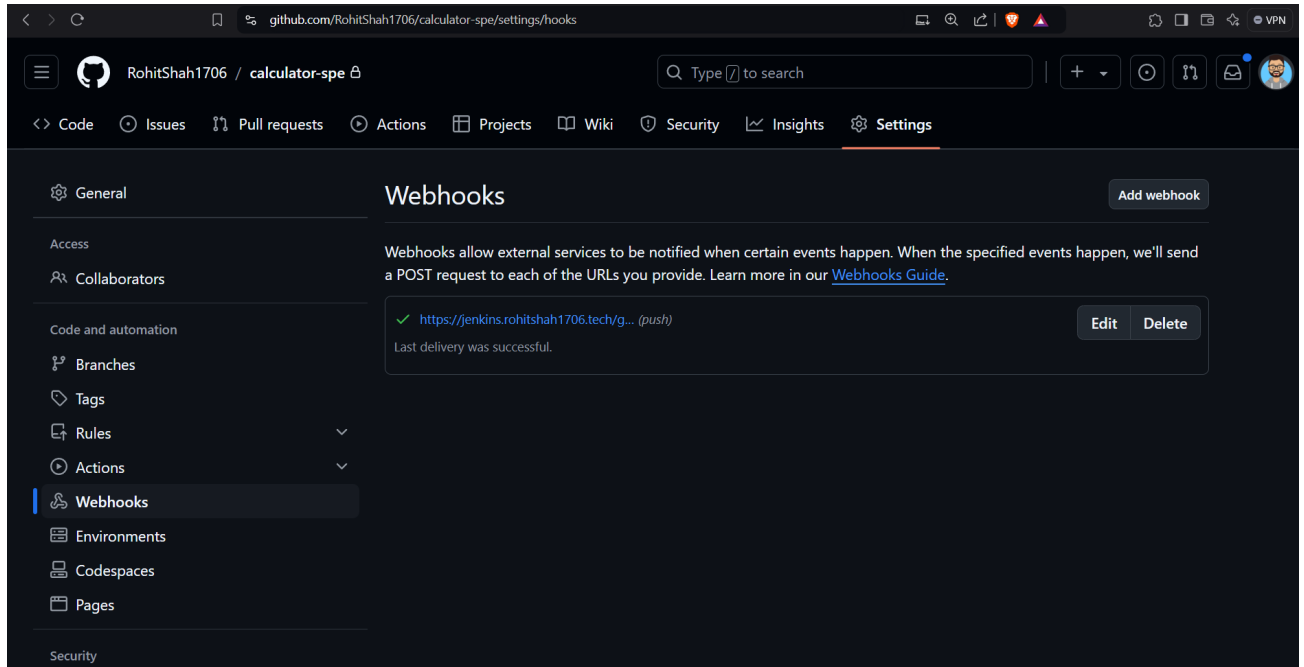
To enable Jenkins to automatically trigger builds upon code changes in the GitHub repository, we need to set up a webhook in GitHub. This allows Jenkins to listen for events (like push events) and react accordingly by starting the configured pipeline. Below are the steps to set up a GitHub webhook for your Jenkins project.

Steps to Set Up a GitHub Webhook

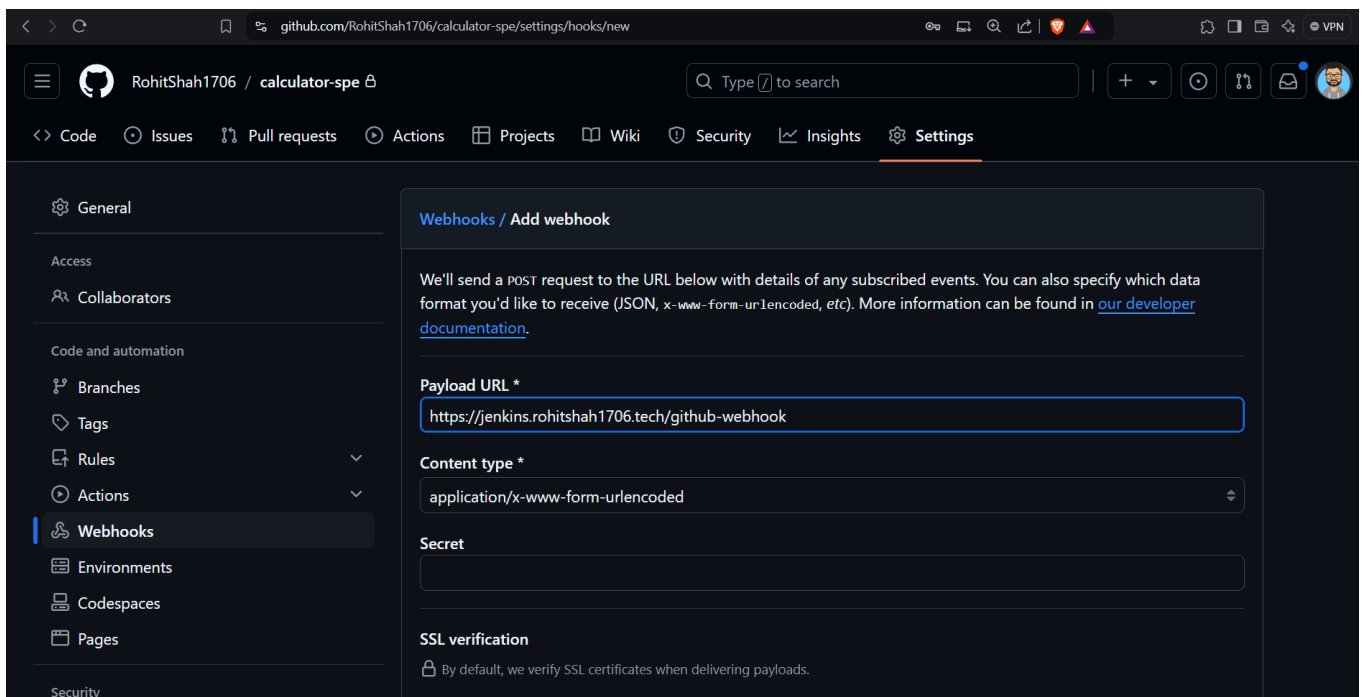
1. Navigate to Repository Settings:

- Open your GitHub repository and click on the **Settings** tab.

2. Click on Webhooks:



- ### 3. Add a Webhook:
- In the **Payload URL** field, enter your Jenkins base URL followed by `/github-webhook`. For example: `http://your-jenkins-url.com/github-webhook`



5. Configure Webhook Settings:

- Set the **Content type** to `application/json`.
- Choose which events you want to trigger the webhook. Typically, you can select **Just the push event** to trigger builds on code changes.

- Make sure the **Active** checkbox is checked.

6. Using ngrok (If Needed):

- If you are running Jenkins locally and don't have a public IP address, you will need to use a tool like **ngrok** to provide a public-facing URL.
- After installing ngrok, you can run it with the command: `ngrok http 8080`
- This command will create a secure tunnel to your local Jenkins server, providing you with a public URL to use in the **Payload URL** field.

7. **Server with Public IP:** However, in my case, I had a server set up with a public IP address and a URL. Therefore, I directly used the public URL of my Jenkins server without needing **ngrok**.

Verification

Once the webhook is set up, you can verify its functionality by pushing a change to your GitHub repository. Jenkins should automatically trigger the pipeline defined in your `Jenkinsfile`, reflecting the changes made in the repository.

Ansible Setup

To set up Ansible for deploying our application, we need to configure our inventory and playbook files, generate SSH keys for secure access, and verify that everything is working correctly. Below are the detailed steps for setting up Ansible.

1. Inventory File (`inventory.ini`)

The inventory file specifies the hosts on which the Ansible playbook will run. Here's an example of what the `inventory.ini` file might look like:

```
[myhosts]
127.0.0.1 ansible_ssh_user=rohit ansible_ssh_port=22
```

- **myhosts:** This section defines a group of servers for deployment. You can add multiple servers under this group.
- **127.0.0.1:** Replace this IP address with the actual IP address of your target server. Use localhost since we're developing on our local machine.
- **ansible_user:** This specifies the username that Ansible will use to connect to the server.

2. Playbook File (`playbook.yml`)

The playbook file contains the instructions that Ansible will execute on the specified hosts. Here's a basic example of what the `playbook.yml` file might contain:

```
- name: Deploy Calculator App

hosts: myhosts

tasks:
  - name: Remove old containers
    docker_container:
      name: calculator
      state: absent
      force_kill: yes

  - name: Deploy Calculator App
    # run in background & interactive
    docker_container:
      name: calculator
```

```
image: rohitshah1706/calculator
state: started
restart: yes
interactive: yes
detach: yes
```

- **hosts:** Specifies the group of servers where the tasks will be executed (from the inventory).
- **tasks:** A list of tasks to be performed on the hosts.
 - The example tasks removed previously running containers, and deploys the latest version of the `rohitshah1706/calculator` image

3. Generating SSH Keys

To allow Ansible to connect to your servers securely without entering a password each time, you need to set up SSH keys.

Step 1: Generate SSH Keys

Run the following command on your local machine:

```
ssh-keygen -t rsa
```

- This will prompt you for a file name and passphrase. You can press **Enter** to accept the defaults.

Step 2: Copy SSH Keys to Remote/Local Server

Use the `ssh-copy-id` command to copy your public key to the remote/local server:

```
ssh-copy-id your_username@127.0.0.1
```

- Replace `your_username` and `127.0.0.1` with your actual username and the server's IP address.

4. Testing the Setup

Ping the Remote Host

To verify that Ansible can connect to the specified hosts, you can use the ping module:

```
ansible myhosts -i inventory.ini -m ping
```

```
$ ansible myhosts -i inventory.ini -m ping
[WARNING]: Platform linux on host 127.0.0.1 is using the discovered Python interpreter at /usr/bin/python3.10, but future
installation of another Python interpreter could change the meaning of that path. See https://docs.ansible.com/ansible-
core/2.16/reference_appendices/interpreter_discovery.html for more information.
127.0.0.1 | SUCCESS => {
  "ansible_facts": {
    "discovered_interpreter_python": "/usr/bin/python3.10"
  },
  "changed": false,
  "ping": "pong"
}
```

- This command will send a ping request to all hosts listed in the `inventory.ini` file. If everything is set up correctly, you should receive a success message from each host.

Setting up Ansible with an inventory and playbook allows for automated deployments and configurations on remote servers. Generating SSH keys ensures secure and passwordless access to the servers, and testing the setup with ping and basic commands verifies that everything is functioning correctly.

