	Pimpri Chinchwad Education Trust's **Pimpri Chinchwad College of Engineering (PCCoE)** (An Autonomous Institute) Affiliated to Savitribai Phule Pune University(SPPU) ISO 21001:2018 Certified by TUV	

Course: DevOps Laboratory	**Code:** BIT26VS01
Name: Amar Vaijinath Chavan	**PRN:** 124B2F001
Assignment 8: Docker Integration with Jenkins for Automated Image Management.	

Aim: To set up a Jenkins job that automates the building of a Docker image from a Dockerfile and pushes the finalized image to the Docker Hub registry.

Objectives:

1. To understand the integration between Jenkins and Docker Engine.
2. To implement automated image versioning using Jenkins build numbers.
3. To securely manage registry credentials using the Jenkins Credential Store.

Theory

The Evolution of Continuous Integration (CI)

Continuous Integration is a development practice where developers integrate code into a shared repository multiple times a day. In a modern DevOps ecosystem, CI is not just about compiling code; it is about creating "Deployable Artifacts." Jenkins acts as the primary orchestrator that bridges the gap between raw source code on GitHub and a run-ready container image.

Dockerization and Container Portability

Dockerization is the process of packaging an application and its entire runtime environment (libraries, system tools, and settings) into a single "Image." This ensures that the Student-CRUD Application runs identically across development, testing, and production environments, eliminating the "it works on my machine" syndrome.

The Role of the Docker Registry (Docker Hub)

A Docker Registry is a hosted service that stores and distributes named Docker images. While the "Build" stage happens locally on the Jenkins server, the "Push" stage makes the image available globally. Docker Hub serves as a centralized source of truth where versioned images (tagged by Jenkins Build Numbers) are stored for future deployments or rollbacks.

Declarative Pipeline and SCM Integration

By using a Jenkinsfile stored in Source Control Management (SCM), the infrastructure becomes as versionable as the code itself. This "Pipeline as Code" approach allows the pipeline to trigger automatically via Webhooks whenever a developer pushes changes to GitHub.

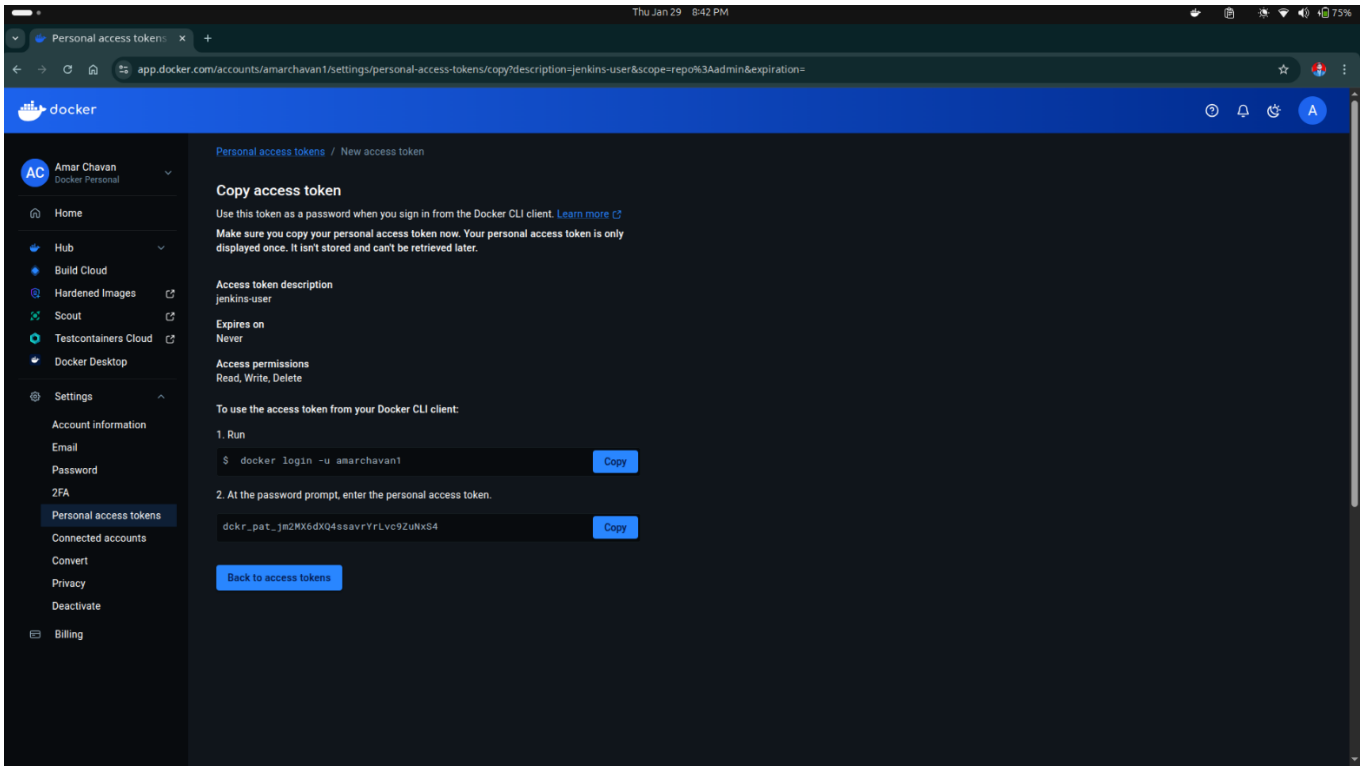
Security and Personal Access Tokens (PAT)

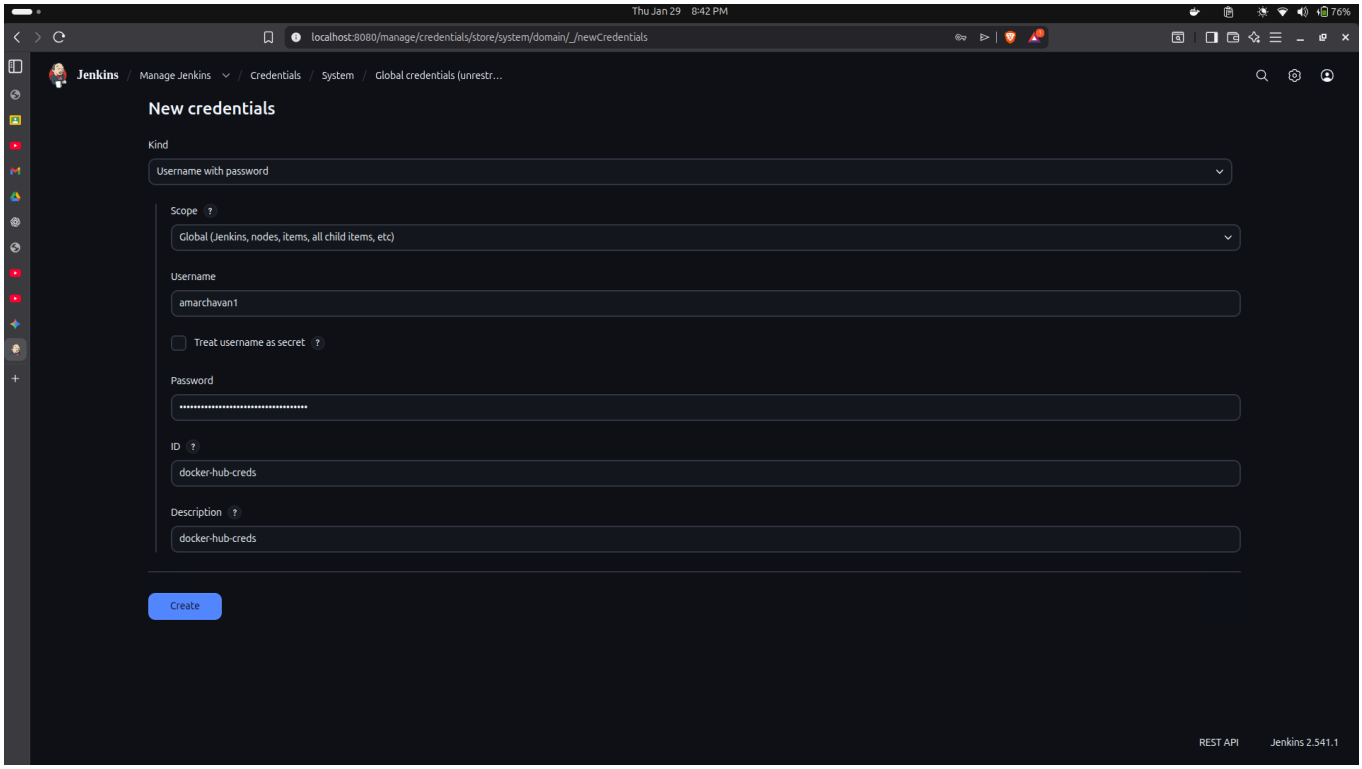
Directly using passwords in automation scripts is a significant security risk. DevOps best practices dictate the use of Personal Access Tokens (PAT). Jenkins secures these using its internal Credential Store, ensuring that sensitive data is only injected into the pipeline at runtime and is masked (hidden) in all console logs.

Practical Procedure / Steps:

Step 1: Docker Hub Token and Jenkins Credentials

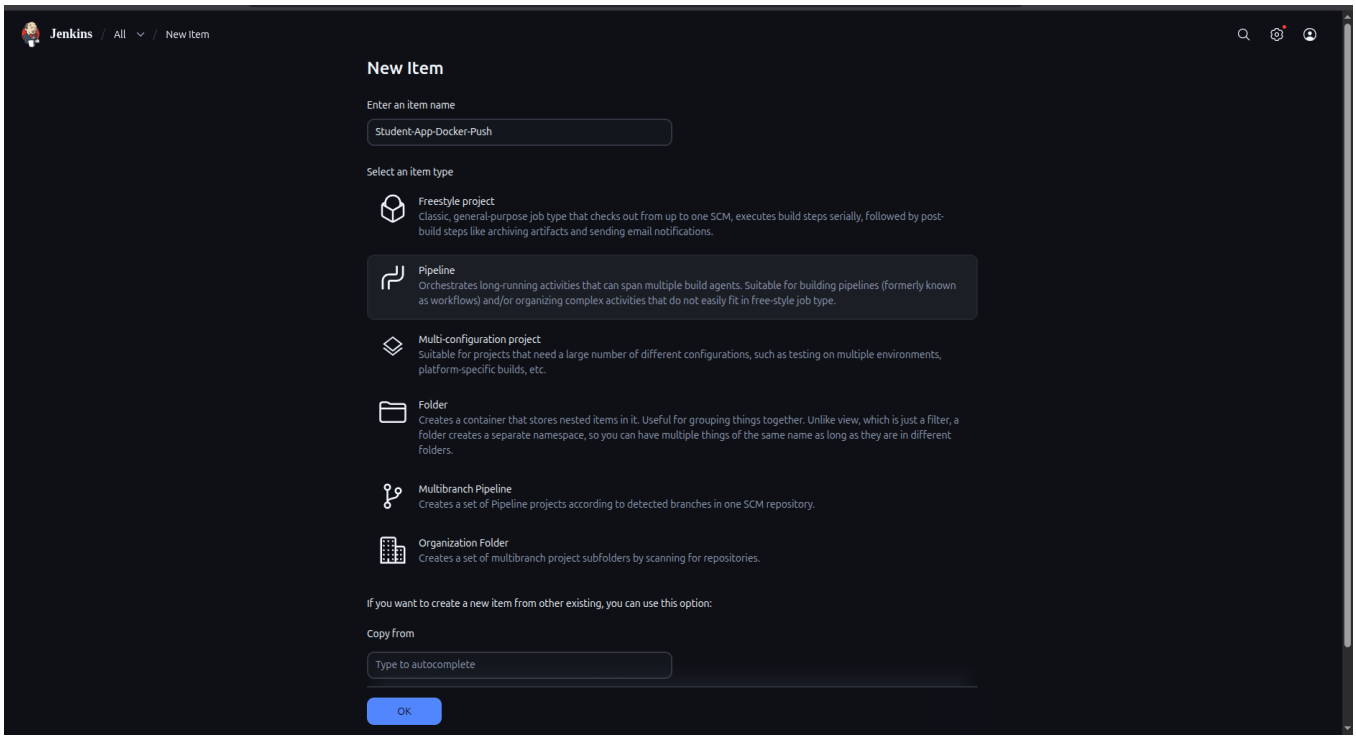
1. Generate a **Personal Access Token** on Docker Hub (Account Settings > Security) with Read & Write permissions.
2. In Jenkins, navigate to **Manage Jenkins > Credentials > Global** and add a new "Username with password" entry.
3. Set the **ID** as docker-hub-creds and enter the Docker Hub username (amarchavan1) and the generated PAT as the password.

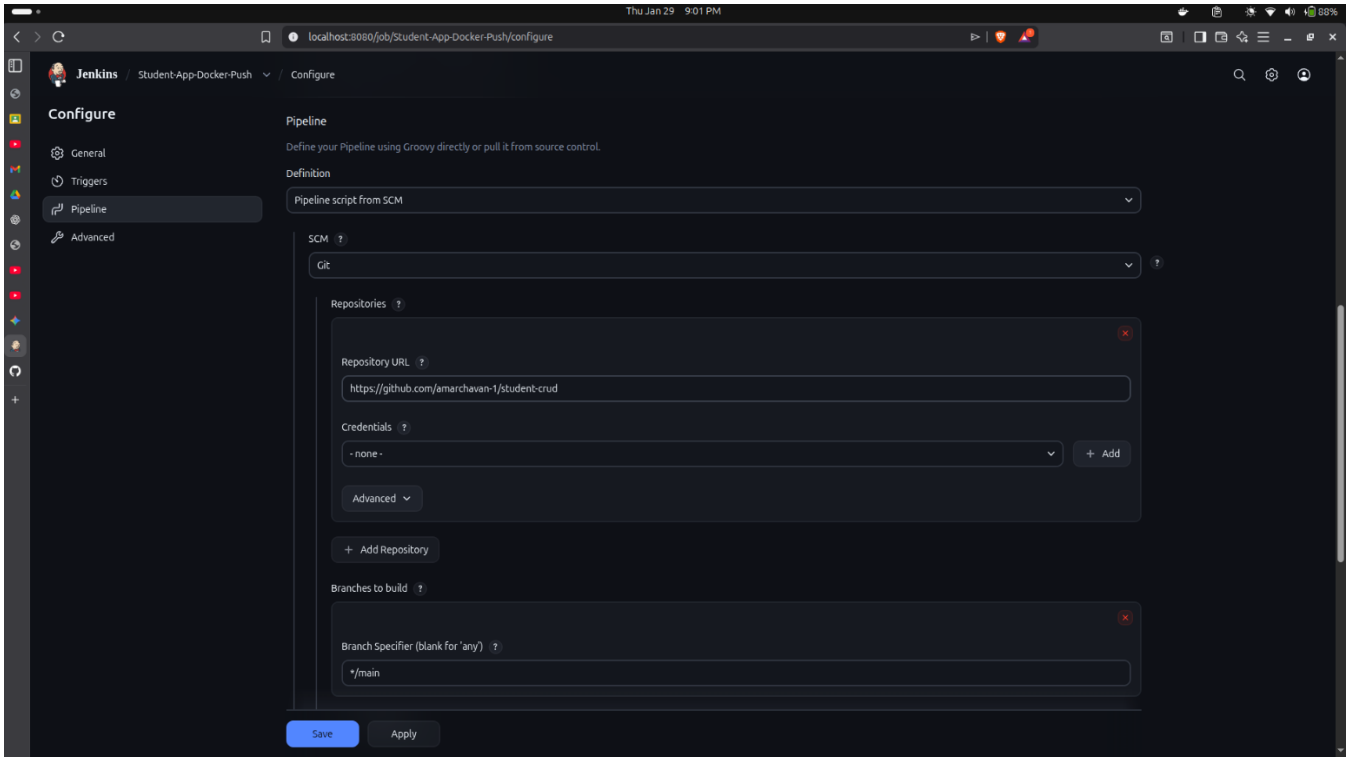




Step 2: Jenkins Pipeline Creation

1. Create a new **Pipeline** item in Jenkins named Student-App-Docker-Push.
2. Configure the job to use **Pipeline script from SCM**.
3. Provide the GitHub Repository URL: https://github.com/amarchavan-1/student-crud and set the branch to */main.



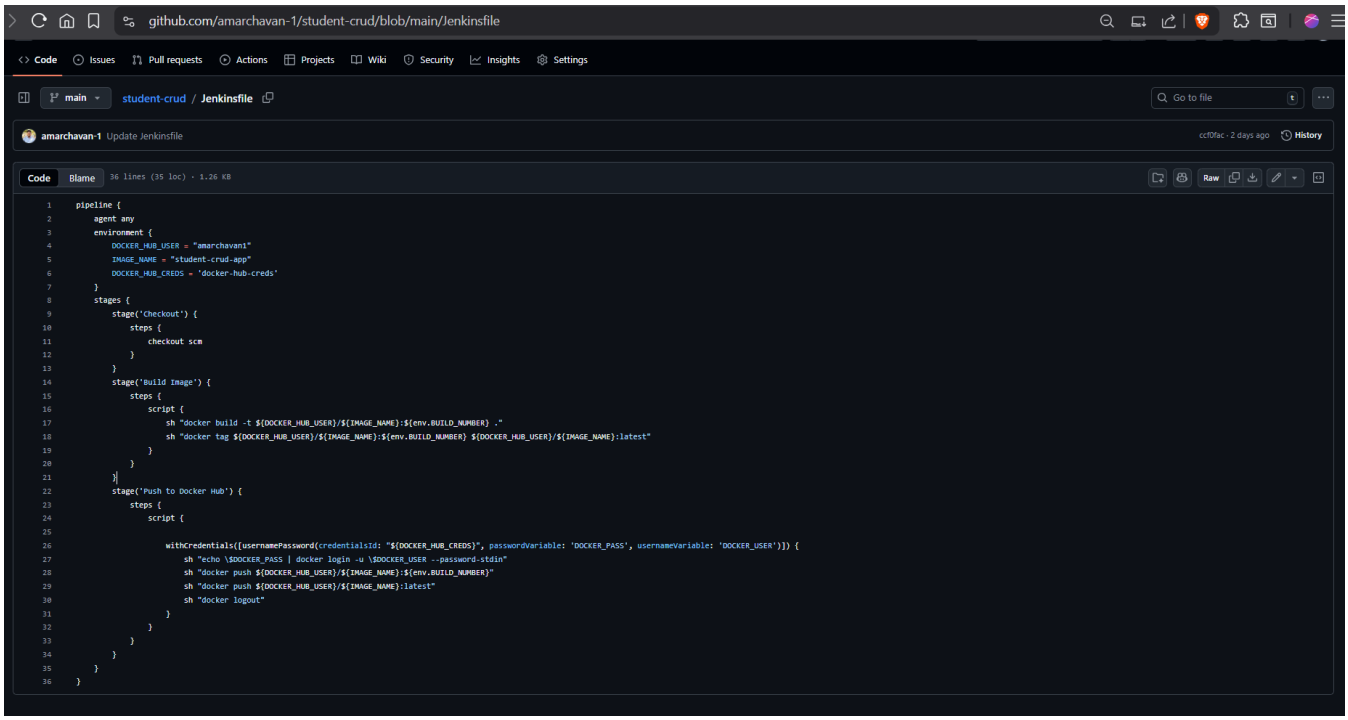


Step 3: Define the Jenkinsfile (Pipeline as Code)

The following script was placed in the root of the GitHub repository to handle the automation:

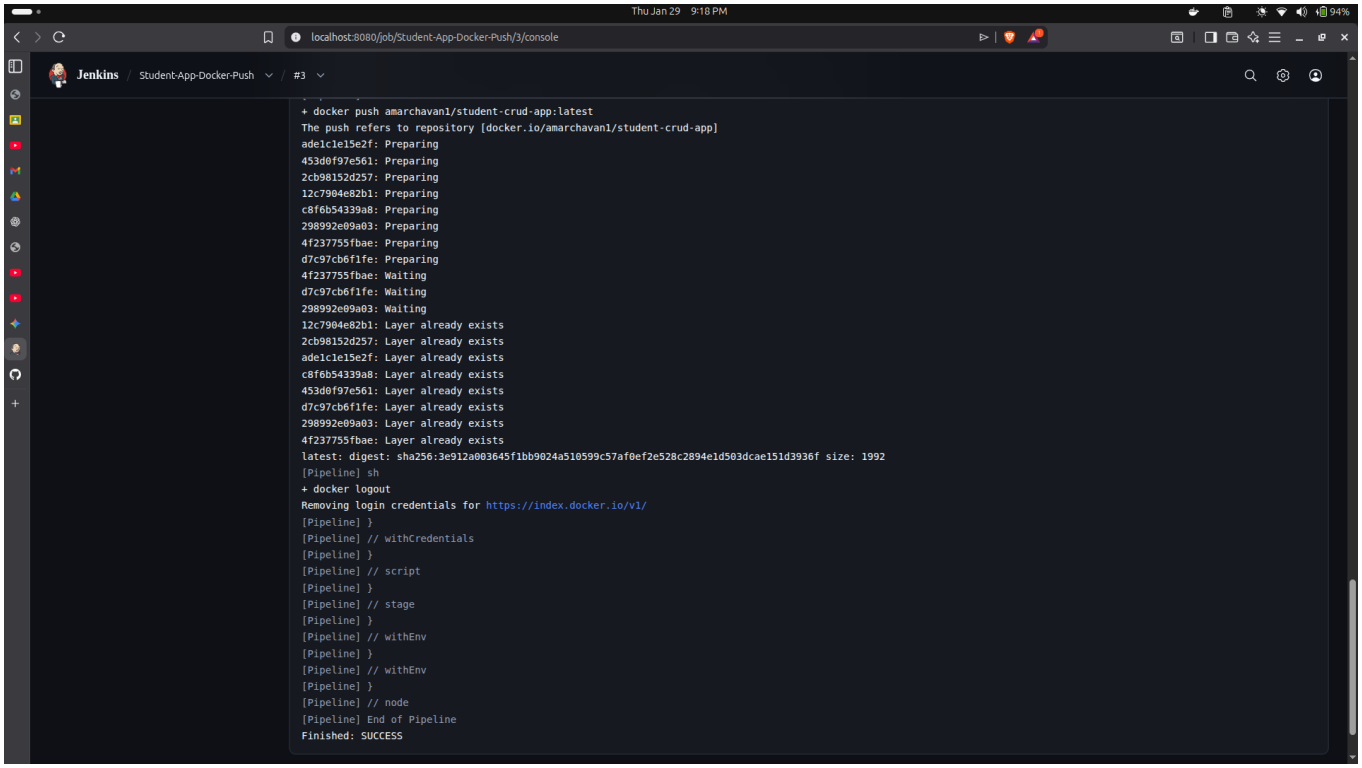
```
pipeline {
    agent any
    environment {
        DOCKER_HUB_USER = "amarchavan1"
        IMAGE_NAME = "student-crud-app"
        DOCKER_HUB_CREDS = 'docker-hub-creds'
    }
    stages {
        stage('Checkout') {
            steps {
                checkout scm
            }
        }
        stage('Build Image') {
            steps {
                script {
                    sh "docker build -t
${DOCKER_HUB_USER}/${IMAGE_NAME}:${env.BUILD_NUMBER} ."
                    sh "docker tag
${DOCKER_HUB_USER}/${IMAGE_NAME}:${env.BUILD_NUMBER}
${DOCKER_HUB_USER}/${IMAGE_NAME}:latest"
                }
            }
        }
```

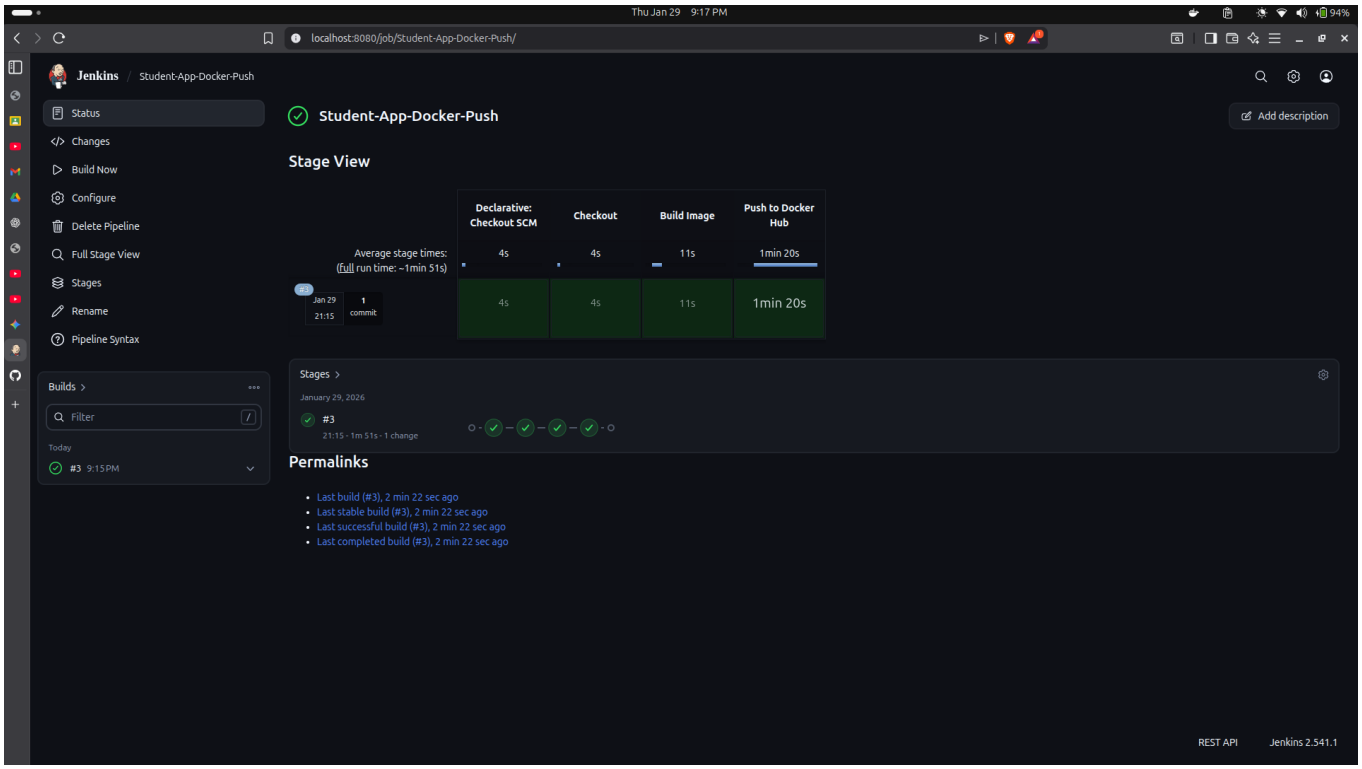
```
        stage('Push to Docker Hub') {
            steps {
                script {

                    withCredentials([usernamePassword(credentialsId:
"${DOCKER_HUB_CREDS}", passwordVariable: 'DOCKER_PASS', usernameVariable:
'DOCKER_USER')]) {
                        sh "echo \$DOCKER_PASS | docker login -u \$DOCKER_USER --
password-stdin"
                        sh "docker push
${DOCKER_HUB_USER}/${IMAGE_NAME}:${env.BUILD_NUMBER}"
                        sh "docker push ${DOCKER_HUB_USER}/${IMAGE_NAME}:latest"
                        sh "docker logout"
                    }
                }
            }
        }
    }
}
```



Step 4: Execution and Monitoring

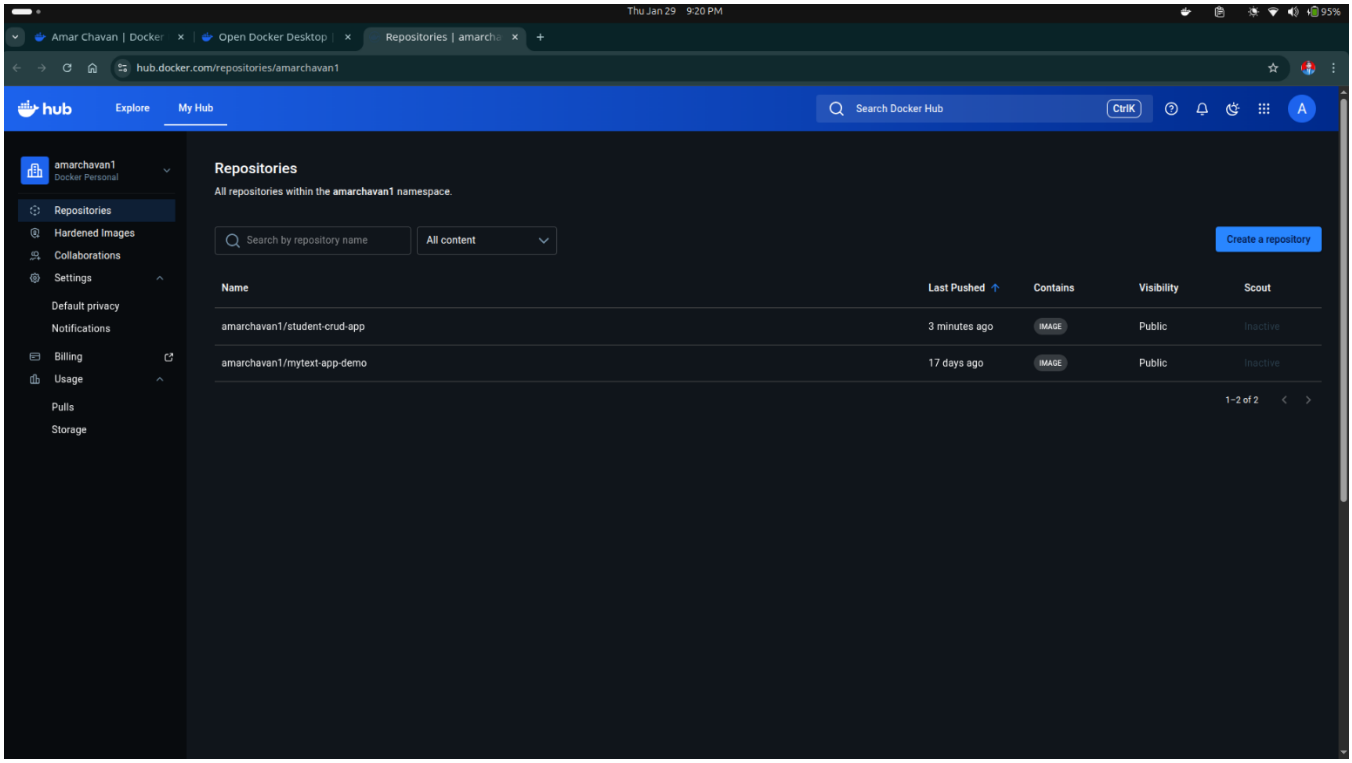
1. Trigger the build by clicking **Build Now**.
2. Monitor the **Stage View** to ensure all stages (Checkout, Build Image, Push to Docker Hub) are completed successfully.
3. Verify the **Console Output** for successful Docker login and image layer uploads.

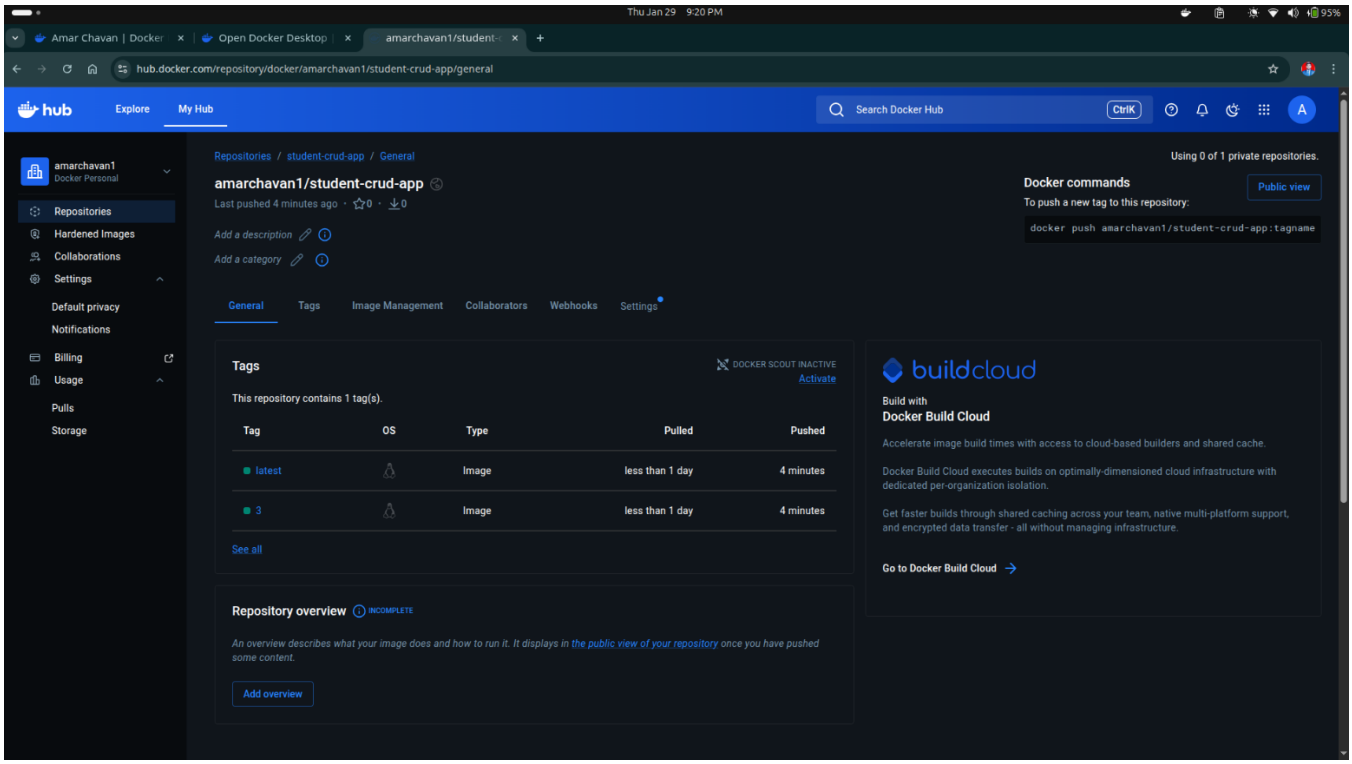




Step 5: Verification on Docker Hub

1. Log in to Docker Hub and navigate to the repository (amarchavan1/student-crud-app).
2. Verify that the image tags (latest and the specific Build Number, e.g., 3) are present and show "Pushed less than a minute ago."





Conclusion:

The successful implementation of Assignment 8 demonstrates the powerful synergy between Jenkins and Docker within an automated CI/CD pipeline. By adopting a Declarative Jenkinsfile, we transitioned to a "Pipeline as Code" model, ensuring that the entire image-building lifecycle is version-controlled and reproducible. The integration facilitated seamless artifact management, where each image was uniquely versioned using Jenkins build numbers to ensure absolute traceability. Security was maintained as a top priority by utilizing the Jenkins Credential Store to handle Docker Hub authentication via masked tokens. This automation significantly reduces the potential for human error and ensures that the Student-CRUD Application is consistently packaged and ready for deployment. Furthermore, the use of containerization ensures environmental consistency across different development and production stages. Ultimately, this assignment establishes a professional foundation for modern software delivery, where infrastructure and application packaging are managed entirely through code.