Program 5

Introduction to Jenkins: What is Jenkins?, Installing Jenkins on Local or Cloud Environment, Configuring Jenkins for First Use

Introduction to Jenkins

What is Jenkins?

Jenkins is an open-source automation server widely used in the field of Continuous Integration (CI) and Continuous Delivery (CD). It allows developers to automate the building, testing, and deployment of software projects, making the development process more efficient and reliable

Key features of Jenkins:

- **CI/CD**: Jenkins supports Continuous Integration and Continuous Deployment, allowing developers to integrate code changes frequently and automate the deployment of applications.
- **Plugins**: Jenkins has a vast library of plugins that can extend its capabilities. These plugins integrate Jenkins with version control systems (like Git), build tools (like Maven or Gradle), testing frameworks, deployment tools, and much more.
- **Pipeline as Code**: Jenkins allows the creation of pipelines using Groovy-based DSL scripts or YAML files, enabling version-controlled and repeatable pipelines.
- **Cross-platform**: Jenkins can run on various platforms such as Windows, Linux, macOS, and others.

Installing Jenkins

Jenkins can be installed on local machines, on a cloud environment, or even in containers. Here's how you can install Jenkins in Window local System environments:

1. Installing Jenkins Locally (Windows)

A. Prerequisites:

- Ensure that J**ava (JDK)** is installed on your system
- You can check if Java is installed by running `java -version` in the terminal.

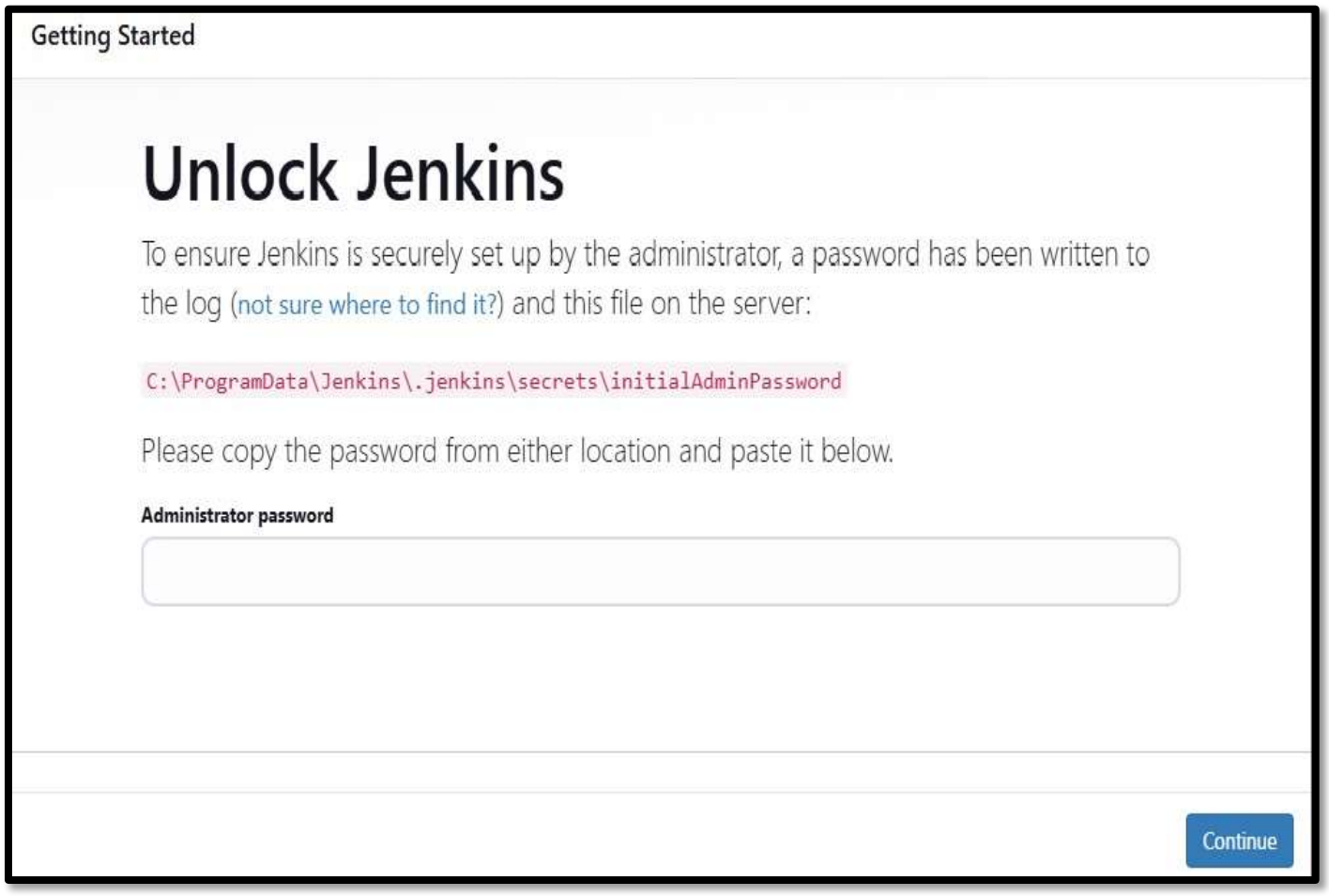
B . Install Jenkins on Window System:

- Download the Jenkins Windows installer from the official Jenkins website.
- Run the installer and select **Run as Local System** (not recommended for production).
- Choose a port (default: **8080**, or custom like **3030**) then **click on test** and **next**.
- Set Java path (e.g., C:\Program Files\Java\jdk-17\).
- Click **Next** until installation starts.
- After successfully installed, Jenkins will be accesed at http://localhost:8080 or http://localhost:3030.

2. Jenkins Setup in browser:

After opening the browser and visiting your local Jenkins address (http://localhost:8080 or your configured port), the **Jenkins setup page** will appear.

Jenkins Initial Setup Screen



3. Unlocking Jenkins (Administrator Password Required)

1. Upon accessing Jenkins in the browser, an **Administrator Password** is required. Navigate to the specified path: C:\Program Files\Jenkins\secrets\initialAdminPassword.

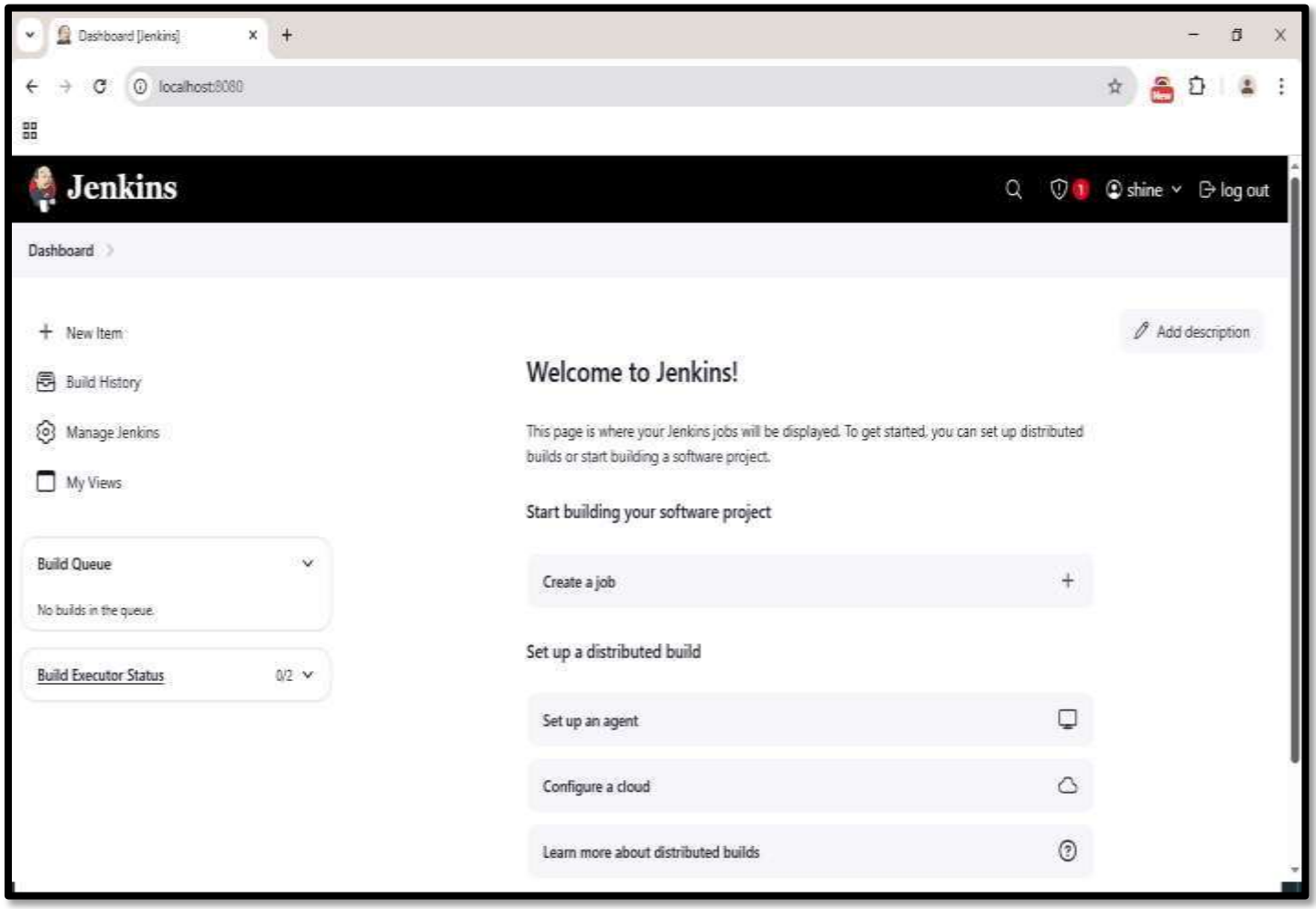
Open the file using **Notepad** or any text editor, copy the password, and paste it into the Jenkins setup page. Click **Continue** to proceed.

2. Next, Jenkins prompts for plugin installation. Select **Install Suggested Plugins** to automatically install the necessary plugins.

3. Once the plugins are installed, create an **Administrator Profile** by entering the required details. Click **Save and Continue**, then **Save and Finish**.

4. Finally, click **Start Using Jenkins** to complete the setup.

Jenkins Dashboard After Successful Installation and Set



Program - 6

Continuous Integration with Jenkins: Setting Up a CI Pipeline, Integrating Jenkins with Maven/Gradle, Running Automated Builds and Tests

Continuous Integration with Jenkins
Objectives:

1. Set up a CI pipeline using Jenkins
2. Integrate Jenkins with Maven or Gradle
3. Run automated builds and unit tests

Tools/Software used:

- Jenkins (installed & running)
- Maven or Gradle
- Git
- JDK 17
- A sample Maven project

Steps to Execute the Lab Program :

Step 1: Install Jenkins and Required Tools

- Install Jenkins from https://jenkins.io
- Install JDK and Maven/Gradle
- Configure environment variables (`JAVA_HOME`, `MAVEN_HOME`, etc.)
- Start Jenkins (`http://localhost:8080`)

Step 2: Install Jenkins Plugins

- Go to **Manage Jenkins → Plugins**
- In the available tab search and Install:
  - Git Plugin
  - Maven Integration Plugin
  - Gradle Plugin (if using Gradle)
  - Pipeline plugin

Step 3: **Create a Sample Java Project**

- Create a Maven/Gradle Java project
- Login into Github account , create a new Github repository
- Push the project into Github repository.

Step 4: **Create a Jenkins Job (Freestyle Project)**

- Go to Jenkins Dashboard → **New Item**
- Name it (e.g., `Maven-CI-Pipeline`)
- Select **Freestyle project** → OK

Step 5: **Configure Source Code Management**

- Under **Source Code Management**, select **Git**
- Enter GitHub repo URL

  Example: https://github.com/username/sample-maven-project.git
- In Branches to build section change master to main branch

Step 6: **Set Build Triggers**

- Click **Poll SCM**
- Schedule: H/5 * * * * (polls every 5 mins)

  (Optional: You can also use "GitHub hook trigger" if webhook is configured)

Step 7: Configure Build

- Under **Build**, click **Add build step → Invoke top-level Maven targets**
  - Goals: `clean compile test package`
- **Add Another build step → Execute Windows batch command**

  java -cp target/<your-jar-name>.jar <your-main-class>

Example : java -cp target/sample-ci-project-1.0-SNAPSHOT.jar com.example.App

- Click save

Step 8: Run and Verify

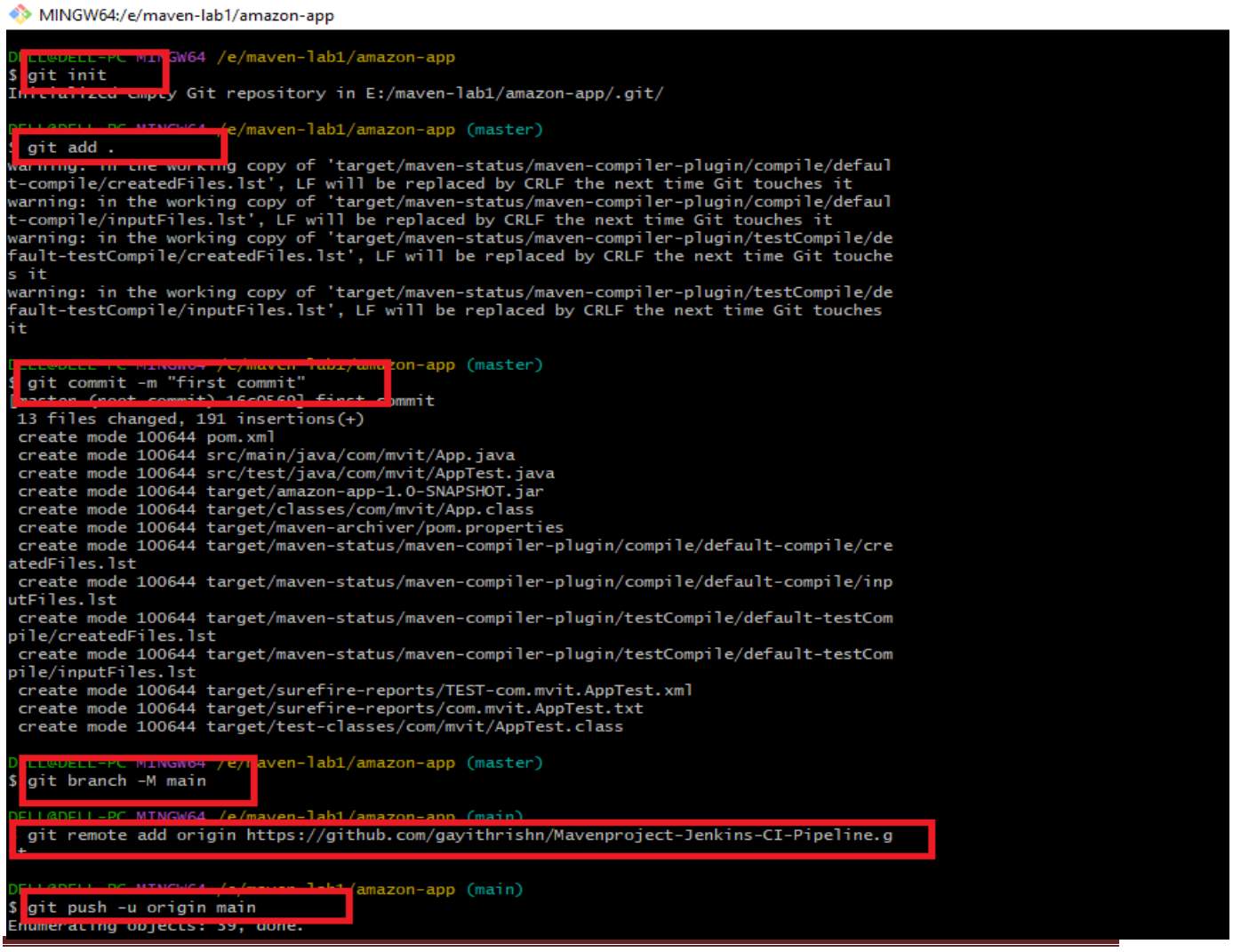
- Click **Build Now**
- Check **Console Output** for logs

Confirm:

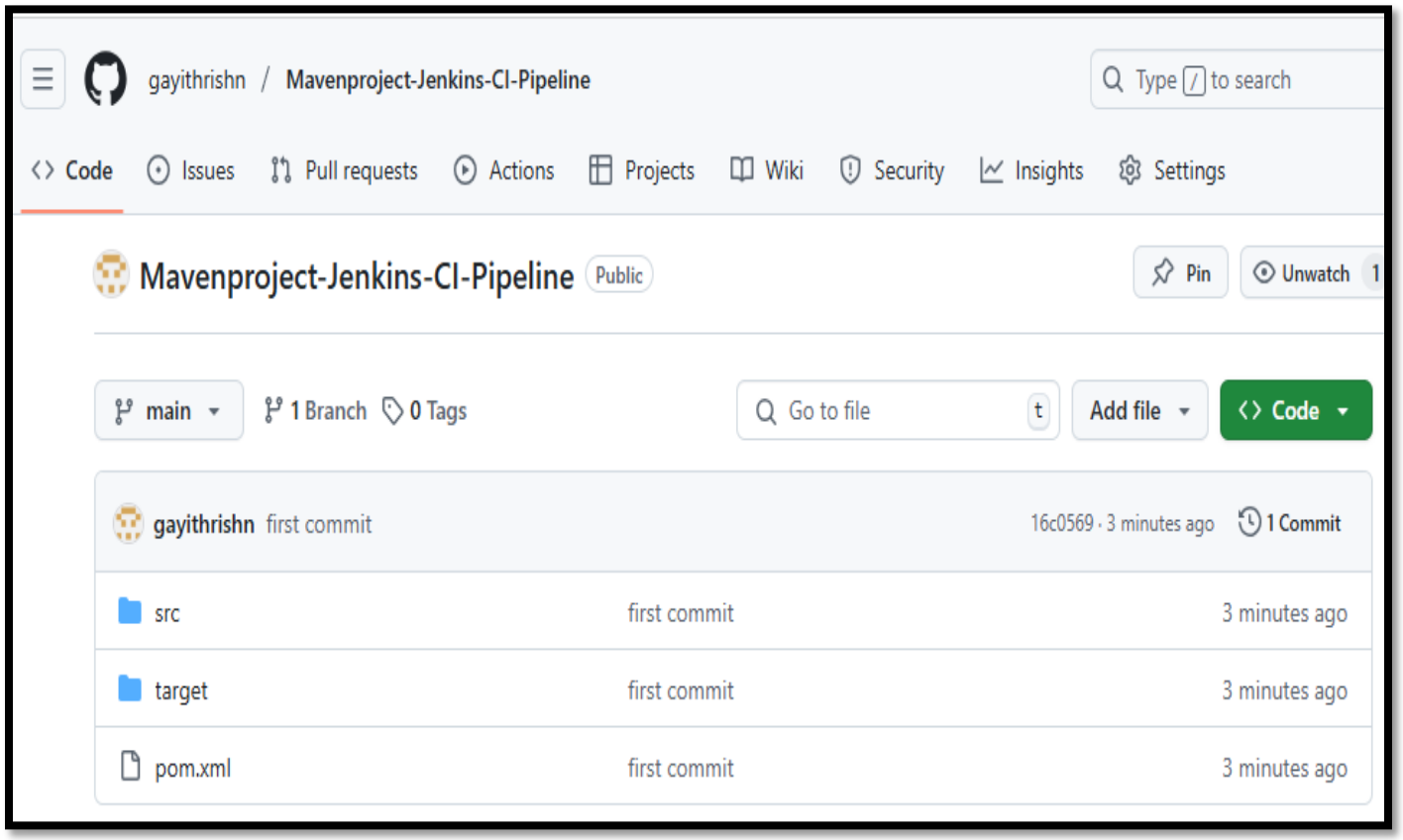
- Source code is pulled from GitHub
- Project builds successfully
- Unit tests run
- JAR file is archived

Output Screenshots:

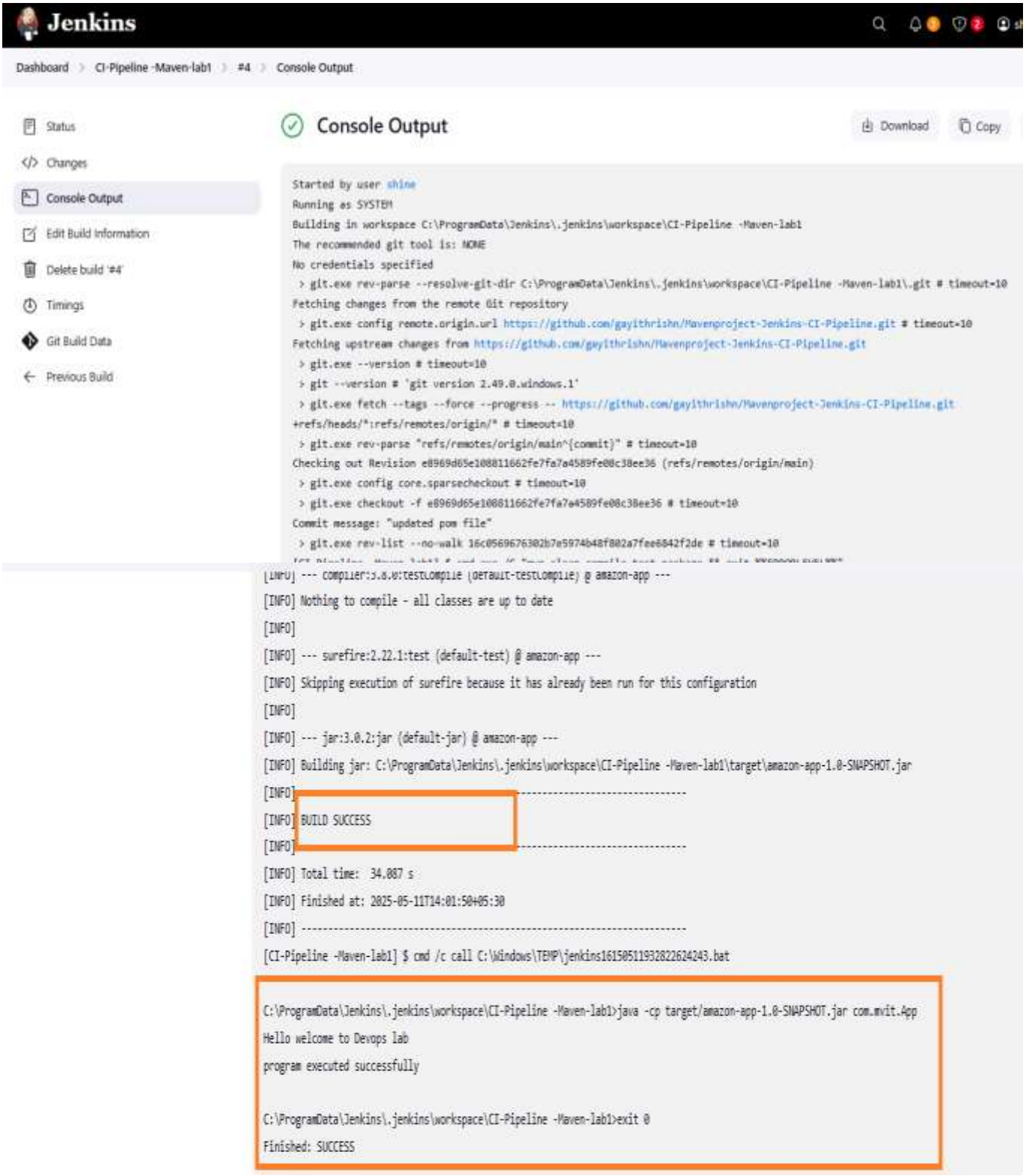
Pushing Maven Project to GitHub Using Git Bash



GitHub Repository After Pushing Maven Project



Jenkins Console Output After Build Execution



Program - 7

Configuration Management with Ansible: Basics of Ansible: Inventory, Playbooks, and Modules, Automating Server Configurations with Playbooks, Hands-On: Writing and Running a Basic Playbook.

Configuration Management :

Configuration Management means setting up and maintaining systems in a consistent and automated way. It ensures systems work reliably across multiple servers. In DevOps, tools like Ansible help automate these tasks to manage large numbers of systems easily.

Ansible:

Ansible is an open-source automation tool used for **configuration management**, **application deployment**, and **task automation**. It allows you to manage systems and software configurations across multiple servers simultaneously.

Ansible uses simple **human-readable YAML (Yet Another Markup Language)** files to describe automation tasks, known as **Playbooks**.

Components of Ansible :

- **Inventory**: Ansible uses an inventory file to define the list of hosts (servers) that it will manage.
- **Playbooks**: A Playbook is a file where tasks are defined using YAML syntax. A playbook specifies the actions that should be carried out on a target server or set of servers.
- **Modules**: Ansible uses modules to perform specific tasks (such as installing packages, starting services, etc.). Some of the most commonly used modules include the **apt** module for package management and the **service** module for service management.

Lab Setup:

Step 1: Install WSL on Windows :

- Install **Windows Subsystem for Linux (WSL)** to provide a Linux environment on a Windows system.

  - ❖ Open **PowerShell** as Administrator.
  - ❖ Run the command:

    wsl - - install

Step 2: Install Ubuntu on WSL

1. Open **Microsoft Store**
2. Search for **"Ubuntu App"** (choose **Ubuntu 22.04 LTS**)
3. Click **Install**
4. After installation, open Ubuntu from Start Menu
5. It will ask to create a UNIX username and password

Step 3: Update Ubuntu Packages

- Run the following command: **sudo apt update -y**

Step 4: Install Ansible inside WSL (Ubuntu)

- Run the command : **sudo apt install ansible -y**
- Verify the installation by checking the Ansible version : **ansible - -version**

Lab Execution : Configuration Management with Ansible

Step 5: Create a Working Directory

- Run the following commands to create and navigate into the directory:

mkdir ansible-lab
cd ansible-lab

Step 6: Create the Inventory File

- Create a file that lists the target hosts for Ansible (in this case, the localhost).
- Use `vi` or any text editor to create the **inventory.ini** file:

vi inventory.ini

- Add the following content to the file:

```
[local]
localhost ansible_connection=local
```

- Save and exit by pressing **Esc**, typing **:wq**, and pressing **Enter**.

Step 7: Test Connectivity with Ansible

- Ensure that Ansible can communicate with the localhost.
- Run the following Ad-hoc command to test connectivity

ansible all -i inventory.ini -m ping

- Expected Output:

```
localhost | SUCCESS => {
"changed": false,
 "ping": "pong"
 }
```

This confirms that Ansible can communicate with the localhost.

Step 8: Create a Basic Playbook

- Write a playbook to automate the installation of the **htop** package.
- Create a playbook install_htop.yml:

vi install_htop.yml

- Add the following content:

```
---
- name: Install htop system monitor tool
  hosts: local
  become: true

  tasks:
    - name: Install htop package
      ansible.builtin.package:
        name: htop
        state: present
```

Save and exit by pressing **Esc**, typing **:wq**, and pressing **Enter**.

Step 9: Run the Playbook

- Run the playbook to execute the automated tasks.
- Run the playbook with the following command:

ansible-playbook -i inventory.ini install_htop.yml

- Expected Output:

**TASK [Ensure htop is installed] **
changed: [localhost]**

Step 10: Verify the Installation

- Verify that the **htop** package is installed correctly.
- Type the command

htop

- This should open the interactive `htop` system monitor (press `q` to exit it).

Output Screenshots :

Ansible Inventory File with Localhost Entry

```
prishitha@DELL-PC:~/devops-lab$ vi inventory.ini
[local]
localhost ansible_connection=local
```

Ansible Connectivity Check Using Ping Module

```
prishitha@DELL-PC:~/devops-lab$ vi inventory.ini
prishitha@DELL-PC:~/devops-lab$ ansible all -i inventory.ini -m ping
localhost | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
prishitha@DELL-PC:~/devops-lab$
```

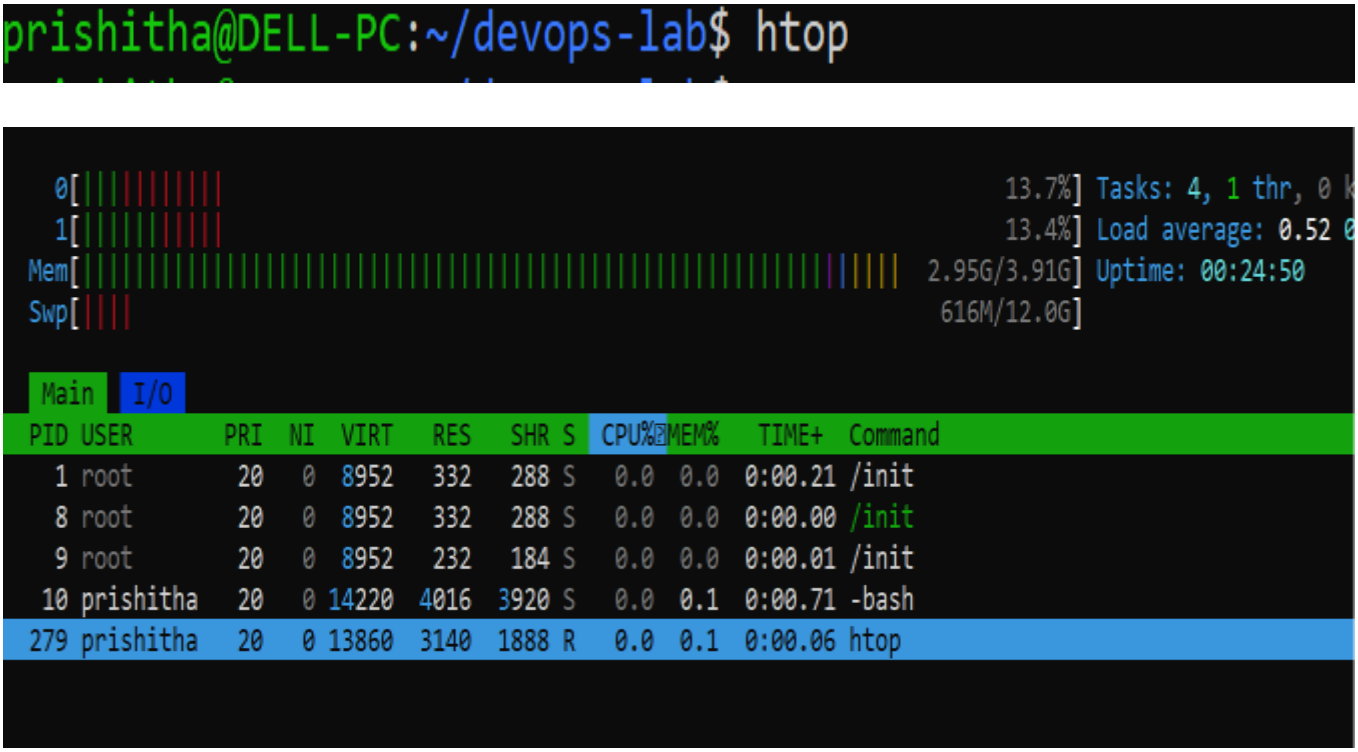
Playbook File Created to Install htop (install_htop.yml)

```
prishitha@DELL-PC:~/devops-lab$ vi install_htop.yml
---
- name: Install htop system monitor tool
  hosts: local
  become: true
  tasks:
    - name: install htop package
      ansible.builtin.package:
        name: htop
        state: present
```

Playbook Execution Output for Installing htop

```
prishitha@DELL-PC:~/devops-lab$ vi install_htop.yml
prishitha@DELL-PC:~/devops-lab$ ansible-playbook -i inventory.ini install_htop.yml

PLAY [Install htop system monitor tool] ****************************************

TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [install htop package] ****************************************************
ok: [localhost]

PLAY RECAP *********************************************************************
localhost                  : ok=2    changed=0    unreachable=0    failed=0    skipped=0
```

htop Command Execution Showing System Monitor



Program – 8

Practical Exercise: Set Up a Jenkins CI Pipeline for a Maven Project, Use Ansible to Deploy Artifacts Generated by Jenkins.

Aim : To configure a Jenkins CI pipeline to build a Maven project and use Ansible (CD) to deploy the generated .jar artifact to a local server.

PART A: Jenkins CI Setup for Maven Project

Step 1. Create a Maven Project

- Use mvn archetype:generate or create using IDE (like IntelliJ/Eclipse).
- Ensure it contains pom.xml and source code.
- Test locally: mvn compile test package

Step 2. Push Project to GitHub:

- Initialize Git, add remote, commit and push:

```
git init
git add .
git commit -m "Initial commit"
git branch -M main
git remote add origin <GitHub_Repo_URL>
git push -u origin main
```

Step 3. Install and Configure Jenkins:

- Start Jenkins and install required plugins: Maven Integration, Git, Ansible.
- Configure Maven and JDK paths in **Manage Jenkins > Global Tool Configuration**.

Step 4. Set Up Jenkins CI Job

- Open Jenkins → New Item → Select "Freestyle project".
- Under **Source Code Management**, choose Git and paste your GitHub repo URL.
- Under **Build Triggers**, enable **Poll SCM** (e.g., H/5 * * * *).
- Under **Build** section:
  - Select "Invoke top-level Maven targets"
  - Goal: clean compile test package
- Add a **Windows Batch Command** to test run the jar:

java -cp target/your-app-1.0-SNAPSHOT.jar com.multit.App

- Save and **Build the job** — confirm that `.jar` is generated in `target/`.

Archive the Artifact:

- Post-build action: Select **Archive the artifacts**, and give:

  target/*.jar

PART B: Set Up Ansible (CD) for deploying the artifact ie .JAR file generated by Jenkins

Step 1. Install Ansible in WSL (Ubuntu):

- **sudo apt update –y**
- **sudo apt install ansible –y**

Step 2. Configure Ansible Inventory and Playbook to deploy the JAR file.

- mkdir ansible-lab
- cd ansible-lab
- vi inventory.ini

  - Add the following into the Inventory file

    [local]
    localhost ansible_connection=local

  - Create the **deploy.yml** playbook:

```
---
- name: Deploy JAR
 hosts: local
 become: true

 tasks:
   - name: Copy JAR file
     copy:
        src: /mnt/c/ProgramData/Jenkins/.jenkins/workspace/your-job/target/your-app.jar
       dest: /home/your-username/ansible-lab/app.jar
       mode: '0755'

    - name: Run JAR file
      shell: nohup java -jar /home/your-username/ansible-lab/app.jar > app.log 2>&1 &
```

Step 3. Run the Playbook manually in WSL to verify the deployment:

ansible-playbook -i inventory.ini deploy.yml

Step 4: Automate the Deployment in Jenkins (optional)

1. Go back to the Jenkins job configuration and add a **post-build action**.
2. Add a **Windows Batch Command** to run the Ansible playbook automatically:

   wsl ansible-playbook -i /home/your-user/ansible-lab/inventory /home/youruser/ansible-lab/deploy.yml

   Trigger the Jenkins Build: When Jenkins builds the Maven project, it will automatically trigger the deployment using Ansible.

Step 5: Verify the Application Runs

- Check the **deployment logs** on your target machine to ensure the JAR file was deployed successfully.

  Using this command **:** **cat app.log**

Step 6: Encounter the Manifest Error

- **During Deployment**, if you try to run the JAR and see the following error:

  no main manifest attribute, in your-app-1.0-SNAPSHOT.jar

This error occurs because the `Main-Class` is missing from the `META-INF/MANIFEST.MF` file inside the JAR.

Step 7: Resolve the Manifest Error

- Fix the Manifest Error by Updating pom.xml

  - **Edit `pom.xml`** in the root directory of your Maven project to add the **`maven-jar-plugin`** configuration under the `<build>` section:

```
<build>
<plugins>
<plugin>
<artifactId>maven-jar-plugin</artifactId>
<version>3.1.0</version>
<configuration>
      <archive>
      <manifest>
            <addClasspath>true</addClasspath>
            <mainClass>com.multit.App</mainClass>  <!-- Replace with main class
      </manifest>
     </archive>
  </configuration>
 </plugin>
</plugins>
</build>
```

Step 8: Redeploy the Artifact

- Push Changes to GitHub**:**

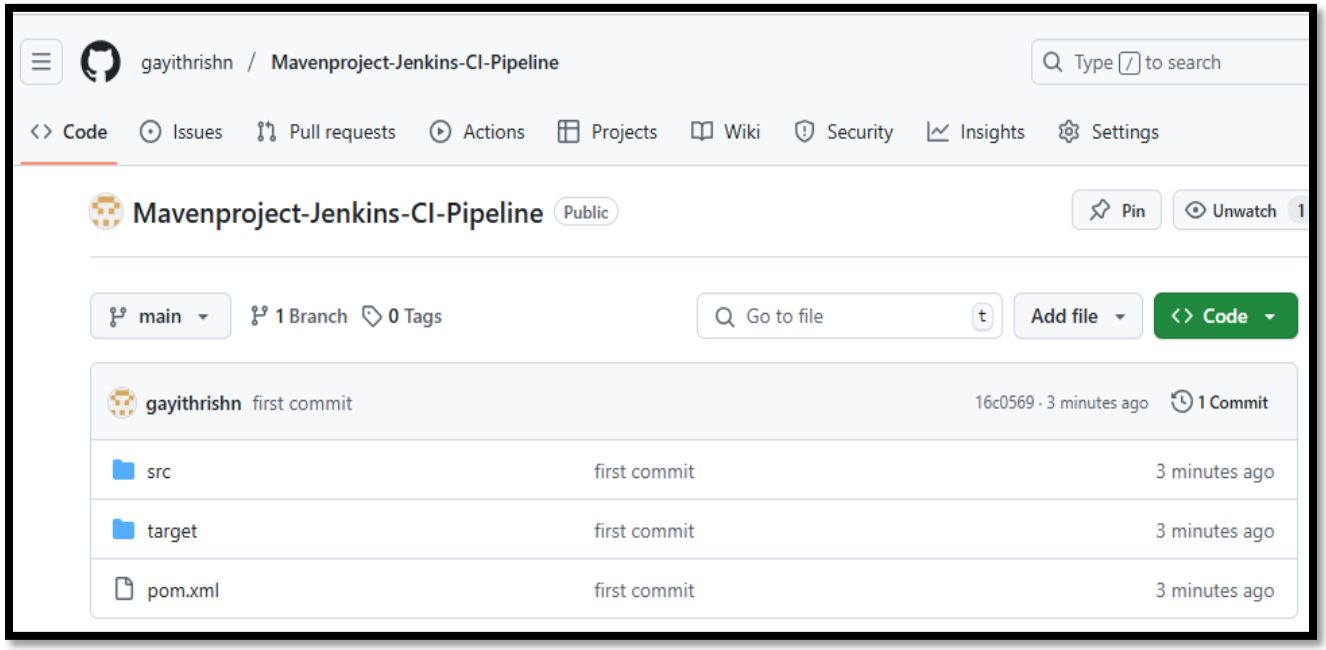
```
git add pom.xml
git commit -m "Fix manifest error by adding Main-Class"
git push  -u  origin main
```

Jenkins will automatically trigger the build **and deploy the new JAR file, which now includes the correct `Main-Class`.**

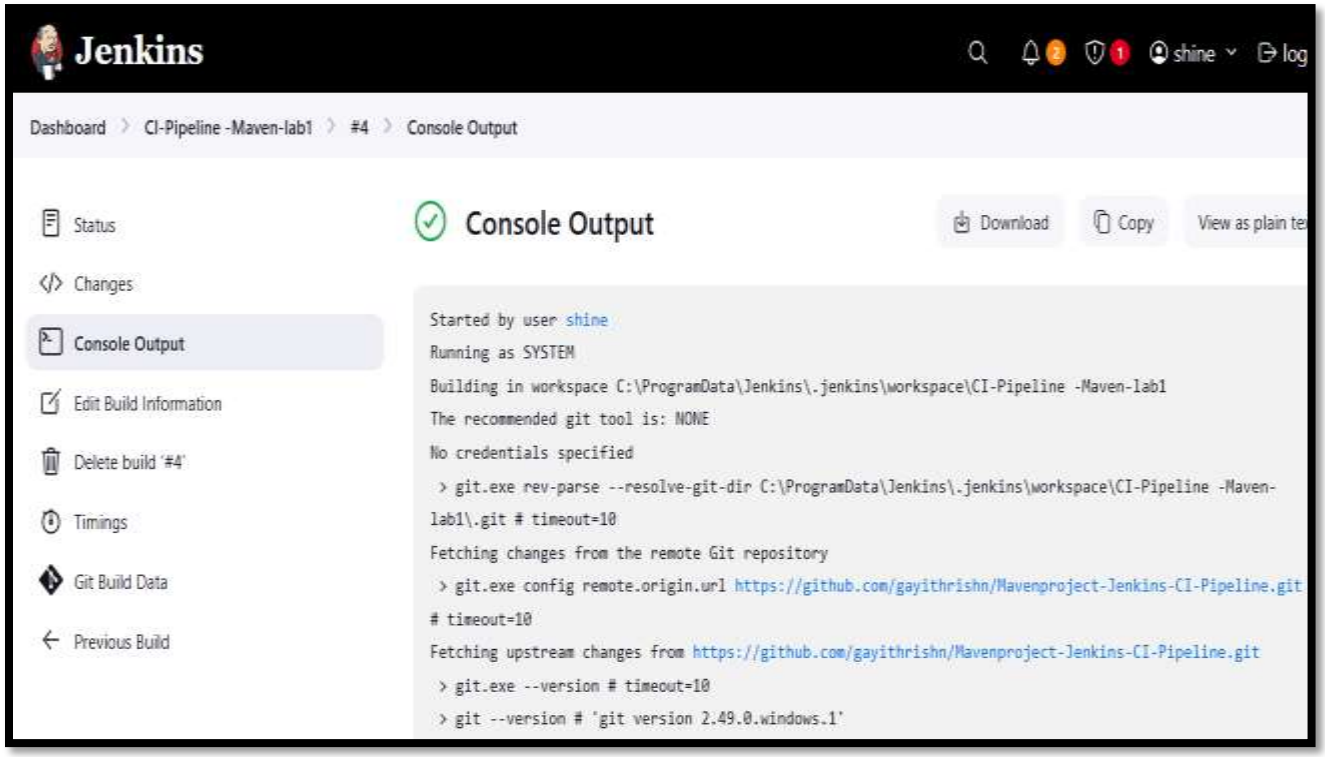
Step 9: Rerun Application

- Using this command : **cat app.log**
- now the application running without the **manifest error**.

Maven Project Hosted on GitHub



Jenkins Console Output (Build Success)



Ansible Playbook creation and Execution

```
prishitha@DELL-PC:~/Ansible-lab$ vi deploy.yml
```

```
---
- name: deploy JAR from jenkins
  hosts: local
  become: true

  tasks:
   - name: copy .jar from jenkins workspace
     copy:
       src: /mnt/c/ProgramData/Jenkins/.jenkins/workspace/CI-Pipeline -Maven-lab1/target/amazon-app-1.0-SNAPSHOT.jar
       dest: /home/prishitha/Ansible-lab/amazon-app-1.0-SNAPSHOT.jar
       mode: 0755

   - name: Run the Application
     shell: nohup java -jar /home/prishitha/Ansible-lab/amazon-app-1.0-SNAPSHOT.jar > app.log 2>&1 &
~
~
~
```

```
prishitha@DELL-PC: ~/Ansible-lab
prishitha@DELL-PC:~/Ansible-lab$ vi deploy.yml
prishitha@DELL-PC:~/Ansible-lab$ ansible-playbook -i inventory.ini deploy.yml

PLAY [deploy JAR from jenkins] ************************************************************************************************************************

TASK [Gathering Facts] ********************************************************************************************************************************
ok: [localhost]

TASK [copy .jar from jenkins workspace] ***************************************************************************************************************
changed: [localhost]

TASK [Run the Application] ****************************************************************************************************************************
changed: [localhost]

PLAY RECAP ********************************************************************************************************************************************
localhost                  : ok=3    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

prishitha@DELL-PC:~/Ansible-lab$
```

Application Running Confirmation (Log Output)

```
prishitha@DELL-PC:~/Ansible-lab$ cat app.log
Hello welcome to Devops lab
program executed successfully
prishitha@DELL-PC:~/Ansible-lab$
```

Manifest Attribute Error During JAR Execution

```
prishitha@DELL-PC:~/Ansible-lab$ cat app.log
no main manifest attribute, in /home/prishitha/Ansible-lab/amazon-app-1.0-SNAPSHOT.jar
prishitha@DELL-PC:~/Ansible-lab$
```

"Fixing Manifest Error: Updating pom.xml to Add Main-Class"

```
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>3.0.2</version>
  <configuration>
        <archive>
            <manifest>
                <addClasspath>true</addClasspath>
                <mainClass>com.mvit.App</mainClass>
            </manifest>
        </archive>
    </configuration>
</plugin>
```

Program - 9

Introduction to Azure DevOps: Overview of Azure DevOps Services, Setting Up an Azure DevOps Account and Project

Azure DevOps is a cloud-based suite of development tools provided by Microsoft to support the complete software development lifecycle (SDLC). It includes a set of services that helps teams plan, develop, test, and deliver applications efficiently. Azure DevOps is designed to support both continuous integration (CI) and continuous delivery (CD), and it integrates seamlessly with various development platforms.

Overview of Azure DevOps Services

Azure DevOps offers several key services, each catering to different parts of the software development lifecycle:

1. Azure Repos : A set of version control tools (Git or Team Foundation Version Control - TFVC) that enables you to manage your code repositories, track changes, and collaborate with your team.

2. Azure Pipelines: A continuous integration and continuous delivery (CI/CD) service that automates the process of building, testing, and deploying code to different environments (e.g., development, staging, production).

3. Azure Boards: A tool for agile project management that allows teams to plan, track, and discuss work. It includes features like Kanban boards, Scrum boards, user stories, and backlog management.

4. Azure Test Plans: Provides tools for manual and exploratory testing. It helps in tracking defects and managing test cases to ensure high-quality code.

5. Azure Artifacts: A service that enables teams to host and share packages (like NuGet, npm, and Maven) within their organization, promoting reuse and easier dependency management.

6. Azure DevOps Services for Collaboration : Features like dashboards, Wikis, and collaboration tools help teams work together effectively by providing visibility into the status of projects and workflows.

Setting Up Azure DevOps Account

1. Go to Azure DevOps Portal:

https://aex.dev.azure.com

2. Sign in with Microsoft Account

- Use your email ID to sign in
- If you don't have one, create a Microsoft account

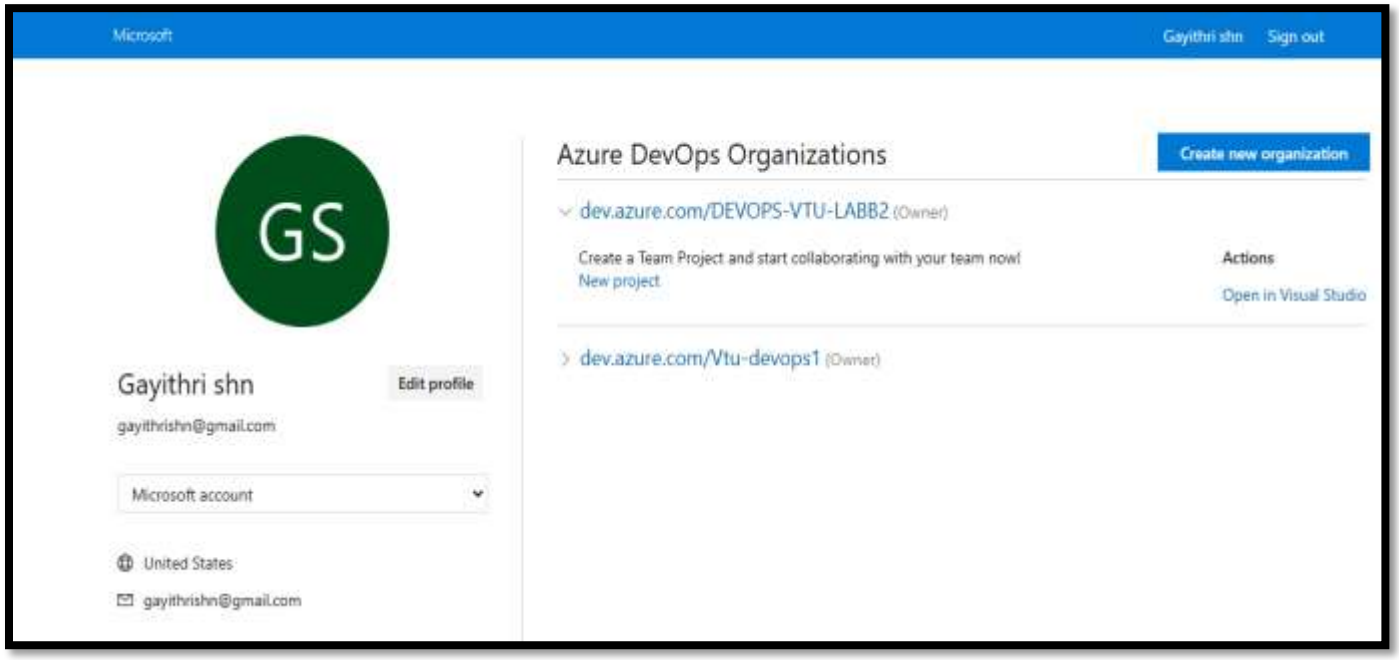
3. Create a New Azure DevOps Organization

- On first login, it asks to **create an organization**.
- Click **"New organization"**
- Enter details:
  - Organization name: `Vtu-Devops-org`
  - Location: Choose nearest (like **South India**)
- Click **Continue**

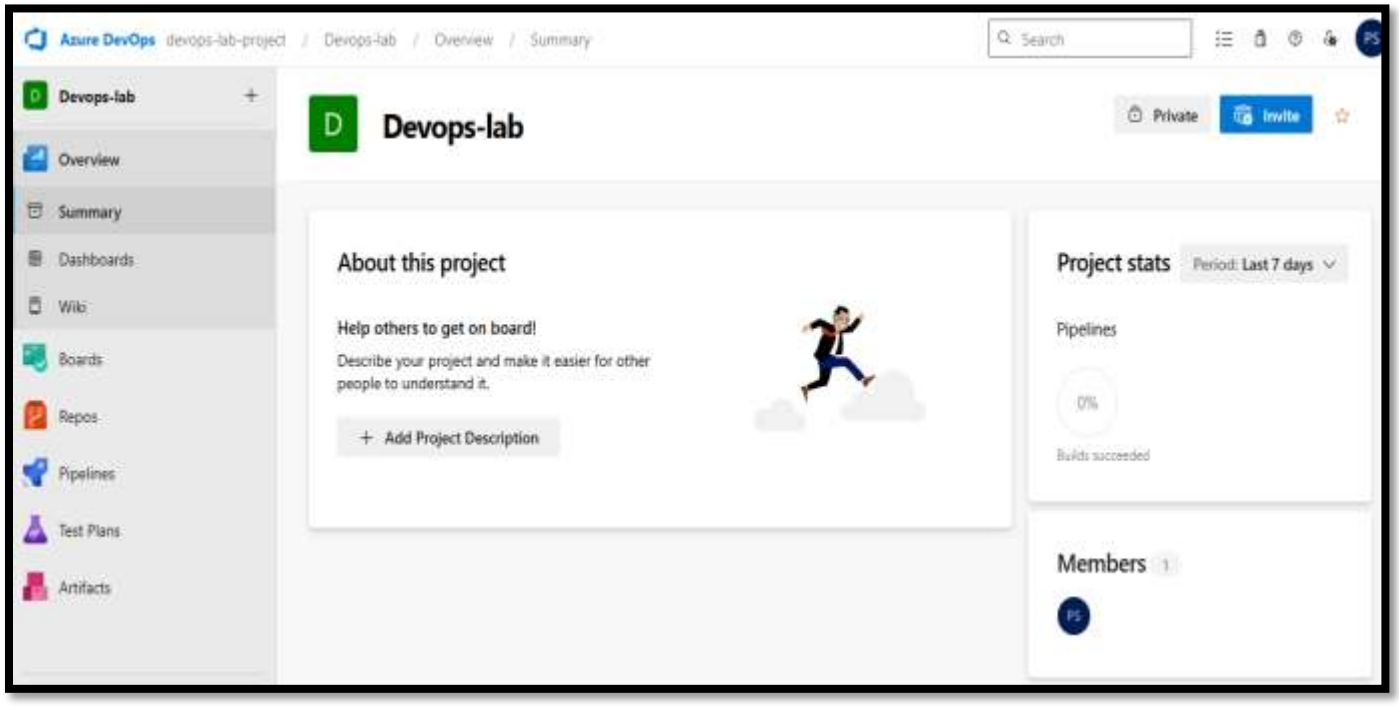
4. Create a New Azure DevOps Project

- Once organization is created, you'll see a **dashboard**.
- Click on **New project**
- Fill the details:
  - Project name: MyFirstDevOpsProject
  - Description: This is my first Azure DevOps lab project
  - Visibility: Choose **Private**
- Click **Create**

Azure DevOps Account and Organization Setup



Azure DevOps Project Dashboard – Overview Page



Program - 10

Creating Build Pipelines: Building a Maven/Gradle Project with Azure Pipelines, Integrating Code Repositories (e.g., GitHub, Azure Repos), Running Unit Tests and Generating Reports

Aim: To set up a CI build pipeline for a Maven/Gradle project using Azure Pipelines with a self-hosted agent, integrate it with a code repository (e.g., GitHub), run unit tests, and generate reports.

1. Continuous Integration (CI): CI is a DevOps practice where developers frequently integrate code into a shared repository, triggering automated builds and tests to catch issues early.

2. Azure Pipelines : A cloud service that supports CI/CD to automatically build, test, and deploy code using pipelines defined in YAML or classic UI.

3. **Self-Hosted Agent** : Instead of using Microsoft-hosted agents, you can configure your own build machine (Windows/Linux) to run Azure DevOps pipelines. This is useful when you need specific tools, environments, or want to save cost.

4. Build Tools :

- **Maven/Gradle**: Used to build the Java project.
- **YAML**: A configuration format for defining pipeline stages, tasks, and agents.

Procedure :

Step 1: Creat a Maven/Gradle Project

Step 2: Push the project to GitHub

Step 3: Create Azure DevOps Project

- Go to https://aex.dev.azure.com
- Click **New Project**
  Project name – Lab program-10
  Visibility (private/public)
- Click **Create**

Step 4: Set Up a Self-Hosted Agent

1. In Azure DevOps, go to **project Settings > Agent Pools**
2. Click on Agent Pool - > **Add Pool**
   - Name: MyLocalPool
   - Selct self – hosted
   - Click create

3. Click on the created pool and click **New agent**

➢ For Download Agent

- Choose OS: Windows/Linux
- Download the ZIP file
- Extract it in a local folder (e.g., `C:\azureagent`)

4. Configure the Agent

- Open command prompt as Administrator
- Navigate to extracted folder cd C:\azureagent
- Run the following command in the extracted folder

config.cmd

Enter the following when prompted:

❖ Azure DevOps URL: `https://dev.azure.com/<your-org>`
❖ Authentication type: **PAT (Personal Access Token)**

How to create PAT

➢ Goto Azure devops -> Click on your **user profile** icon (top-right corner)
➢ In the **Personal Access Tokens** section, click **"New Token"**
➢ Fill in the token details:

- **Name**: (e.g., `MyAgentToken`)
- **Organization**: Select your Azure DevOps organization
- **Expiration**: Choose duration (e.g., 30 days, 90 days)
- **Scopes**:
  - Select **Custom defined**
  - Enable **Agent Pools → Read & Manage**
    Build → Read and execute
- Click **Create**
- Once generated, **copy the token immediately**

❖ Enter the PAT token you created from Azure DevOps
❖ Agent Pool: Select the one created above - MyLocalPool
❖ Agent Name: Choose a unique name (e.g Azure agent)

5. Run the Agent as service

run.cmd

Step 5 : **Create a YAML Build Pipeline**

- Go to **Pipelines** > **New Pipeline**
- Choose your repo (GitHub or Azure Repos)
- Choose Starter pipeline **YAML**

Step 6: Write the YAML Configuration

```
trigger:
  - main

pool:
    name: 'MyLocalPool'  # Your self-hosted agent pool name

steps:
  # Step 1: Checkout the Code from GitHub
  - checkout: self
    displayName: 'Checkout Code from GitHub'

  # Step 2: Build and Run Unit Tests
  - script: mvn clean test
    displayName: 'Build and Run Unit Tests'

  # Step 3: Publish Test Results (JUnit)
  - task: PublishTestResults@2
    inputs:
      testResultsFiles: '**/target/surefire-reports/TEST-*.xml'
      testResultsFormat: 'JUnit'
      failTaskOnMissingResultsFile: true
    displayName: 'Publish Maven Test Results'

  # Step 4: Publish Build Artifacts
  - task: PublishBuildArtifacts@1
    inputs:
      PathtoPublish: 'target'
      ArtifactName: 'drop'
      publishLocation: 'Container'
    displayName: 'Publish Build Artifacts'
```

Step 7: Run Pipeline

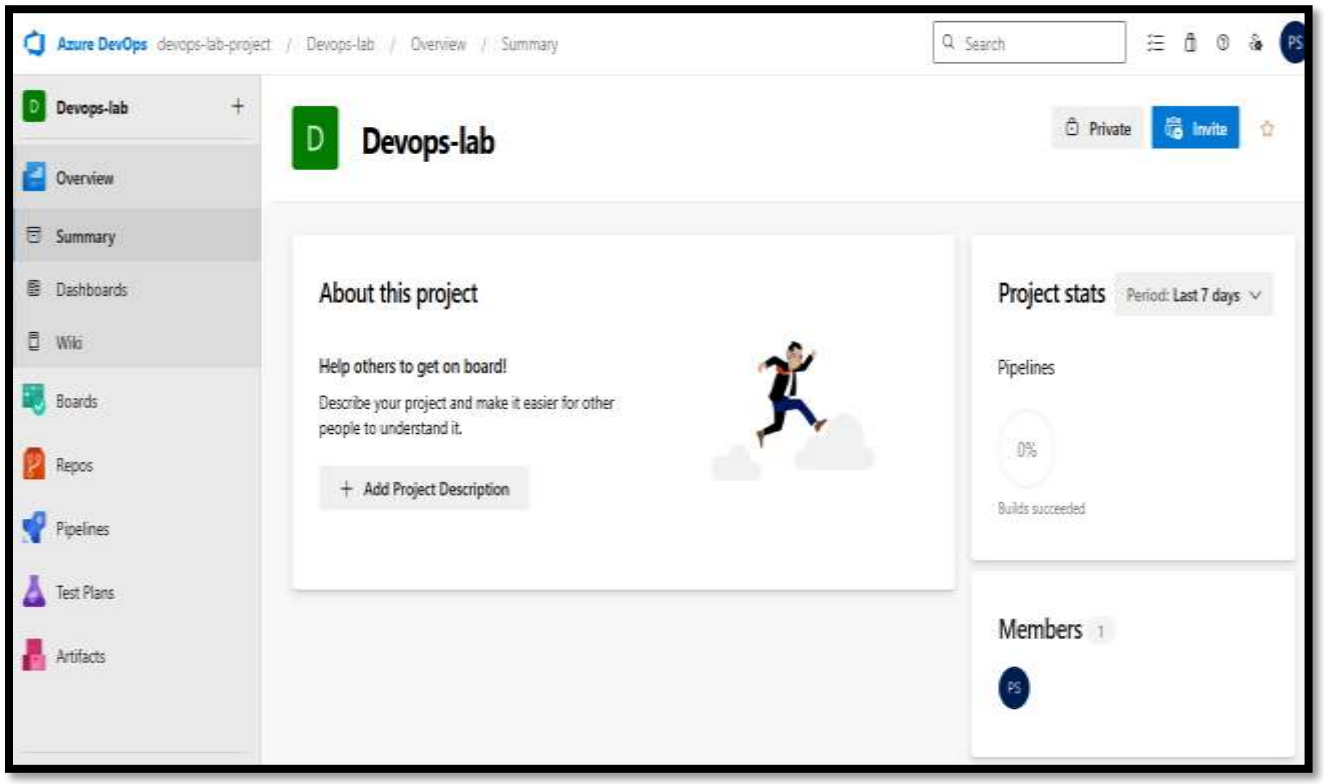
- Click **Run** to trigger the build
- Check logs for:
  - Build success/failure
  - Test results

Step 8: View Test Reports

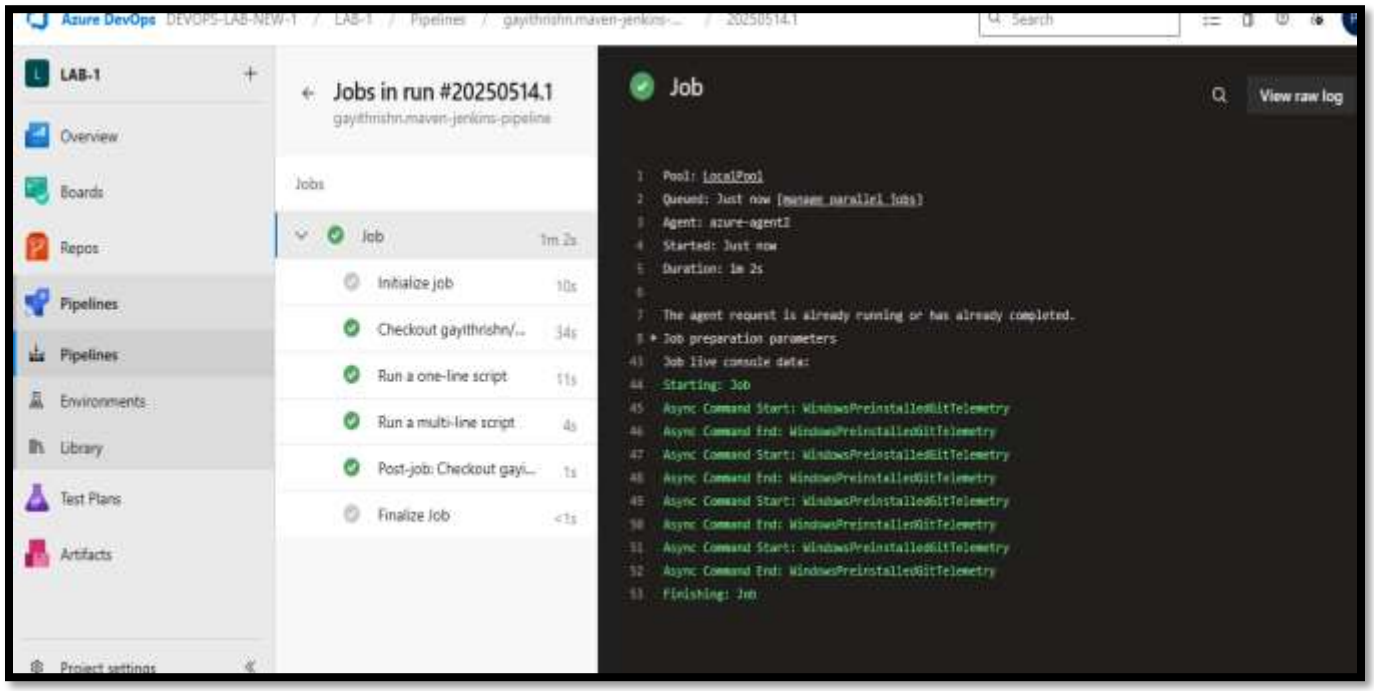
- Go to **Pipelines > [Your Pipeline] > Runs**
- Click the latest run
- Navigate to **Tests** tab to view passed/failed unit tests

Output Screenshots

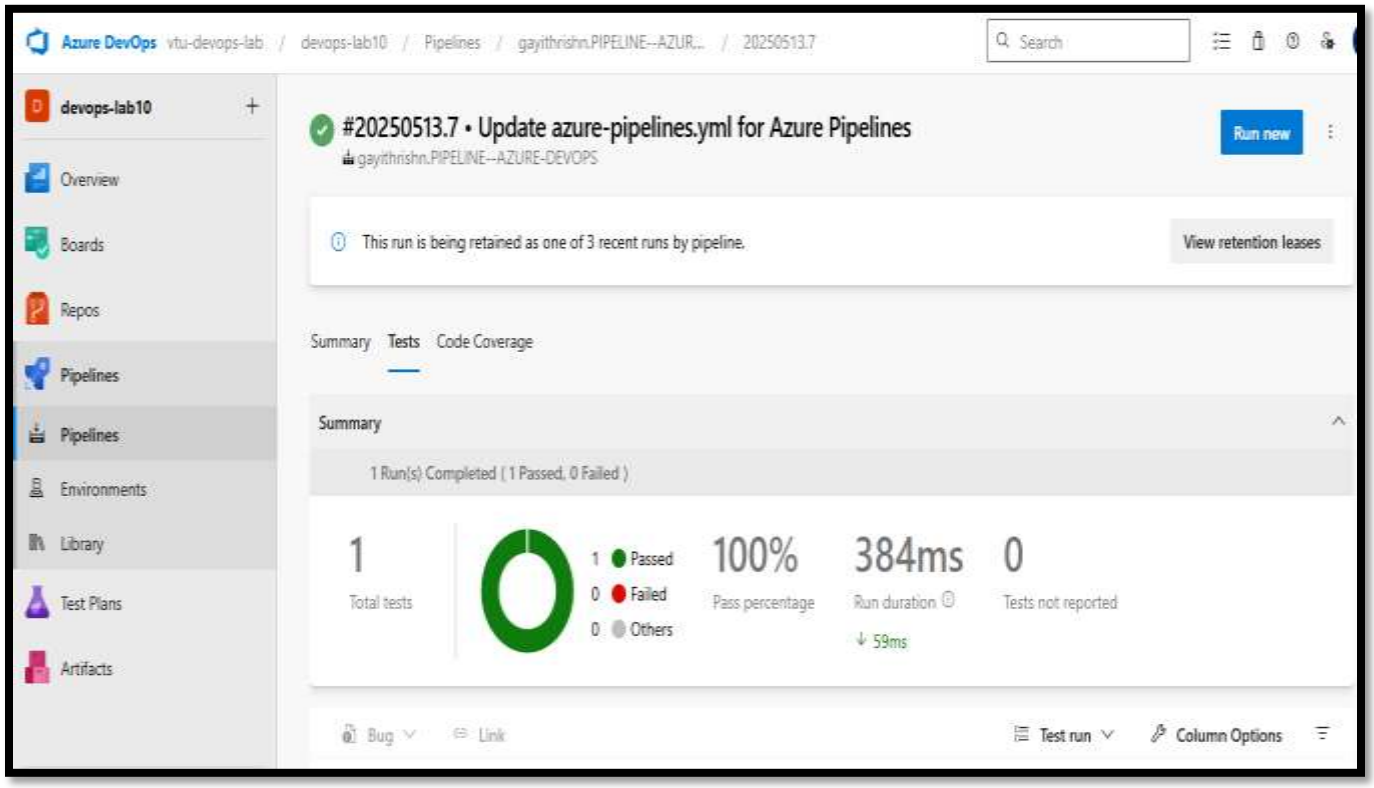
Azure DevOps Dashboard – Accessing Pipelines and Repositories



Successful Pipeline Run – Job Execution and Logs



Unit Test Results – All Test Cases Passed in Azure Pipeline



Program - 11

Creating Release Pipelines: Deploying Applications to Azure App Services, Managing Secrets and Configuration with Azure Key Vault, Hands-On: Continuous Deployment with Azure Pipelines

Aim: Creating CI/CD Pipelines using Azure DevOps with Simulated Deployment on Self-Hosted Agent (Alternative to Azure App Services)

To simulate application deployment using Azure Pipelines when Azure App Services and Classic Release Pipelines are unavailable, by using a self-hosted agent and YAML-based pipeline.

Note: Due to limitations in the Azure DevOps Free Tier (such as unavailability of Azure App Services and Classic Release Pipelines), the deployment was simulated using a local folder and a self-hosted agent. This simulation mimics the actual release pipeline behavior and fulfills the learning objective of continuous deployment using Azure Pipelines.

Tools and Technology Used:

- Azure DevOps (Free Tier)
- GitHub (for code repository)
- Maven (for build automation)
- YAML (pipeline configuration)
- Windows OS (for self-hosted agent)

Steps Followed:

1. **Create a Maven Project:**
   - Create a basic Java Maven project locally or on GitHub.
2. **Push to GitHub:**
   - Upload the project to a public GitHub repository.
3. **Create Azure DevOps Project:**
   - Sign in to Azure DevOps (https://aex.dev.azure.com).
   - Create a new project and link it to the GitHub repository.
4. **Configure Self-Hosted Agent:**
   - Download and configure a self-hosted agent on your Windows system.
   - Register the agent with Azure DevOps under Organization Settings > Agent Pools.

5. Create YAML Pipeline (`azure-pipelines.yml`):

```
trigger:
- main

pool:
   name: MYLOCALPOOL     # Name of your self-hosted agent pool

steps:
-  task: Maven@3
   inputs:
      mavenPomFile: 'pom.xml'
      goals: 'package'

- script: |
     echo "Simulating deployment..."
     mkdir deployed
     copy target\*.jar deployed\
     echo "Deployment successful!"
  displayName: 'Simulate Deployment'

- script: |
     echo "Contents of deployed folder:"
     dir deployed
  displayName: 'Verify Deployment Output'
```

6. **Run Pipeline:**

- Commit the YAML file to GitHub.
- The pipeline will be triggered and executed by the self-hosted agent.

7. **Verify Deployment:**

1. Check the deployed folder on the self-hosted agent system.

- Open the folder where your **self-hosted agent is installed** (e.g.C:\Users\<YourUsername>\Downloads\AZURE-AGENT-NEW1_work\1\s\
- This folder contains the source code and pipeline files during the run.

2. Look inside:

- C:\Users\<YourUsername>\Downloads\AZURE-AGENTNEW1_work\1\s\deployed
- That the `deployed` folder exists
- It contains the correct `.jar` file copied from the `target/` folder.
- This confirms that the **deployment simulation worked successfully**.

Program - 12

Practical Exercise and Wrap-Up: Build and Deploy a Complete DevOps Pipeline, Discussion on Best Practices and Q&A.

To implement a complete DevOps pipeline that integrates source code management, continuous integration using Maven, and deployment using Azure Pipelines on a self-hosted agent.

Software and Tools Required:

- Azure DevOps (Free Tier Account)
- Self-Hosted Agent on Windows
- Java (JDK 17 or later)
- Apache Maven
- Git and GitHub
- Command Prompt (cmd) for Agent Execution

Step 1: Create a Simple Java Maven Project

- Create a pom.xml file for Maven with the following mainClass configuration:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>3.2.2</version>
      <configuration>
        <archive>
          <manifest>
            <mainClass>HelloWorld</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

- Initialize a Git repository and push the project to GitHub.

Step 2. Configure Azure DevOps Project

- Login to Azure DevOps
- Create a new project (e.g., `DevOpsFinalProject`).
- Link your GitHub repository under **Repos**.

Step 3: Set Up a Self-Hosted Agent

- Go to **Project Settings → Agent Pools → Add Pool (e.g., `MYLOCALPOOL`)**.
- Download the agent from Azure DevOps.
- Extract it and configure with:

  config.cmd --url https://dev.azure.com/<your-org>

  --auth pat --token <your-PAT-token>

- Start the agent

Step 4: Create azure-pipelines.yml

```
trigger:
  - main

pool:
      name: MYLOCALPOOL

steps:
-   task: Maven@3
    inputs:
        mavenPomFile: 'pom.xml'
        goals: 'clean package'

- script: |
      echo "Simulating deployment..."
      mkdir deployed
      copy target\*.jar deployed\
      echo "Deployment successful!"
    displayName: 'Simulate Deployment'

- script: |
      echo "Running the JAR file..."
      java -jar deployed\*.jar
    displayName: 'Run Application'
```

This YAML performs CI (build) and CD (deployment to local machine) and **runs the JAR file**, so the output will be visible in the terminal where run.cmd is running.

Step 5. Run and Verify the Pipeline

- Commit and push the YAML to GitHub
- Azure DevOps triggers the pipeline on `main` branch
- The self-hosted agent executes the pipeline:
    - Compiles and packages the Java program.
    - Copies the `.jar` to `deployed/` folder.
    - Executes the `.jar` file.

You should see the following output in the command prompt:

Hello from DevOps Final Lab!

Expected Output:
Simulating deployment...
Deployment successful!
Running the JAR file...
Hello from DevOps Final Lab!

Discussion on Best Practices in CI/CD:

1. **Use Source Control**
    - Store all code in a Git repository (GitHub/Azure Repos).
2. **Automate Builds**
    - Use Maven/Gradle for reliable, repeatable builds.
3. **Use YAML for Pipelines**
    - YAML is infrastructure-as-code and version-controlled.
4. **Keep Build and Deploy Steps Separate**
    - Use separate stages (optional) to isolate concerns.
5. **Agent Management**
    - Use self-hosted agents for controlled environments and debugging.
    - Use Microsoft-hosted agents for quick builds on clean machines.
6. **Secrets Management (Optional)**
    - Use Azure Key Vault to store passwords, API tokens securely.
7. **Monitoring and Logs**
    Check pipeline logs and agent output for debugging