GitLab CI/CD Documentation

1. Introduction

GitLab CI/CD is an integrated tool within GitLab that automates the software development lifecycle. It simplifies and accelerates the processes of building, testing, and deploying applications. This documentation covers the essentials of GitLab CI/CD, comparing it with Jenkins, and provides a step-by-step guide to getting started.

2. GitLab CI/CD vs. Jenkins

2.1 Overview

GitLab CI/CD and **Jenkins** are popular CI/CD tools that automate software development workflows. Both have their strengths and cater to different needs:

- **GitLab CI/CD**: An integrated part of GitLab, providing a seamless experience from version control to deployment. It offers built-in CI/CD features, making it easier to set up and manage pipelines directly within the GitLab interface.
- **Jenkins**: A standalone open-source automation server that supports a wide range of plugins to extend its capabilities. It requires more configuration and setup compared to GitLab CI/CD but offers greater flexibility and customization options.

2.2 Key Comparisons

Feature	GitLab CI/CD	Jenkins
Integration	Built into GitLab	Standalone with various integrations
Setup	Easier setup within GitLab	Requires separate setup and configuration
User Interface	Integrated UI with GitLab	Separate UI with extensive plugin options
Configuration	`.gitlab-ci.yml` file	Jenkinsfile and plugin configurations
Scalability	Scales with GitLab	Highly scalable with plugins and agents
Pipeline Visualization	Visual pipeline graphs and status	Customizable dashboard with plugins
Pricing	Included in GitLab (free and paid plans)	Free, with costs for plugins and infrastructure

3. Getting Started with GitLab CI/CD

3.1 Prerequisites

1. **GitLab Account**: Ensure you have a GitLab account and access to a project repository.
2. **GitLab Runner**: Install and register GitLab Runners to execute jobs. GitLab Runner Installation Guide.

What is a GitLab Runner?

A GitLab Runner is an application that works with GitLab CI/CD to run jobs in a pipeline. It processes jobs by executing scripts defined in `.gitlab-ci.yml`. Runners can be configured to run on different platforms and are essential for automating the build, test, and deployment processes of your project.

Utility of a GitLab Runner

1. **Execution of CI/CD Pipelines**:
   - Runners execute the various stages of your CI/CD pipeline, such as building the application, running tests, and deploying code.
2. **Environment Isolation**:
   - Runners often use Docker containers to provide isolated environments for each job, ensuring that jobs run in clean, reproducible environments without interference from previous jobs.
3. **Load Distribution**:
   - Runners distribute the workload across multiple machines, which helps to speed up the CI/CD process and manage resources more effectively.
4. **Cross-Platform Support**:
   - Runners can be configured to run on different operating systems (Windows, Linux, macOS), which is useful for testing and building applications in various environments.

Types of GitLab Runners

1. **Shared Runners**:
   - These runners are available to all projects in a GitLab instance. They are managed by the GitLab administrator and are useful for general purposes.
2. **Specific Runners**:
   - These are dedicated to a specific project or group. They can be configured to meet the particular needs of the project, such as using certain tools or libraries.
3. **Group Runners**:
   - These runners are shared across all projects in a group. They allow centralized runner management for multiple projects within a group.

Workflow with GitLab Runners

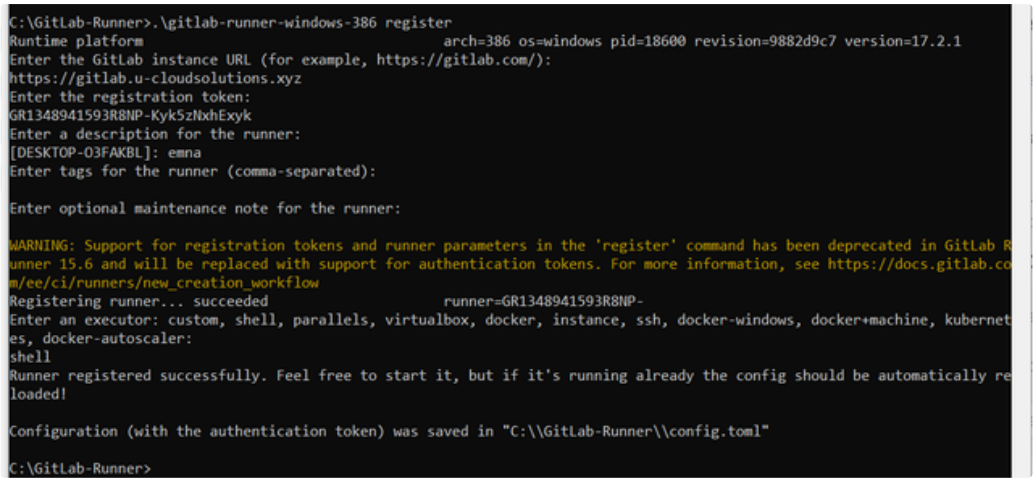
1. **Triggering a Pipeline**:
   - A pipeline is triggered by an event, such as a commit or a merge request.
2. **Job Assignment**:
   - GitLab assigns jobs to available runners. If you have configured tags, GitLab uses those tags to match jobs with appropriate runners.
3. **Job Execution**:
   - The runner fetches the code, runs the scripts defined in `.gitlab-ci.yml`, and sends back the results (console logs, artifacts, etc.) to GitLab.
4. **Results Reporting**:
   - GitLab displays the results of each job in the UI, allowing developers to see successes or failures and take appropriate actions.

Importance of GitLab Runners

- **Automation**: Runners automate the entire software development lifecycle, reducing manual errors and increasing efficiency.
- **Reproducibility**: By using containers and isolated environments, runners ensure that builds and tests are consistent and reproducible.
- **Continuous Integration**: Runners facilitate continuous integration by automatically testing and validating each code change.
- **Continuous Deployment**: Runners also enable continuous deployment by automatically deploying applications once tests pass, ensuring quick and reliable software delivery.

Generating a Registration Token

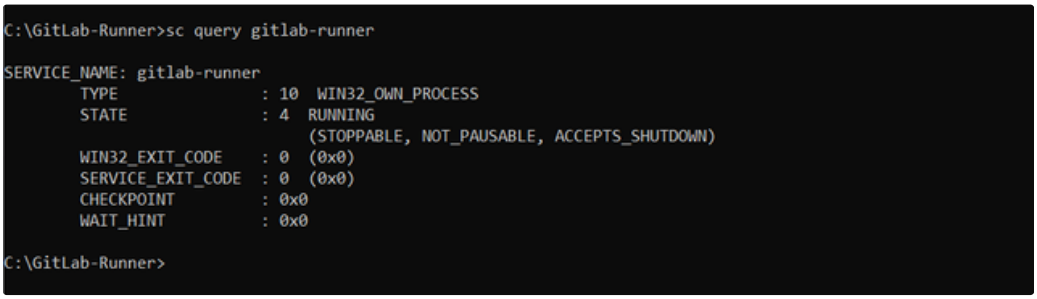
1. **Log in to GitLab**
2. **Navigate to Project : the Lumen Microservice**:
   - **For a specific project**:
     i. Go to the project for which I want to register the runner.
     ii. Navigate to **Settings > CI/CD**.
     iii. Expand the **Runners** section. I found a **Registration Token**



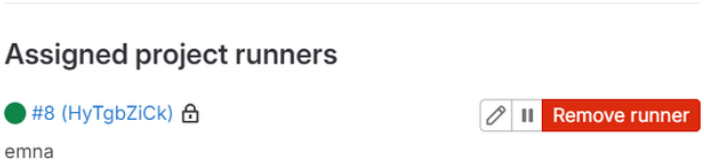
`.\gitlab-runner-windows-386 install`

`.\gitlab-runner-windows-386 start`

and here's the service of gitlab runner is running



and here it's connected :

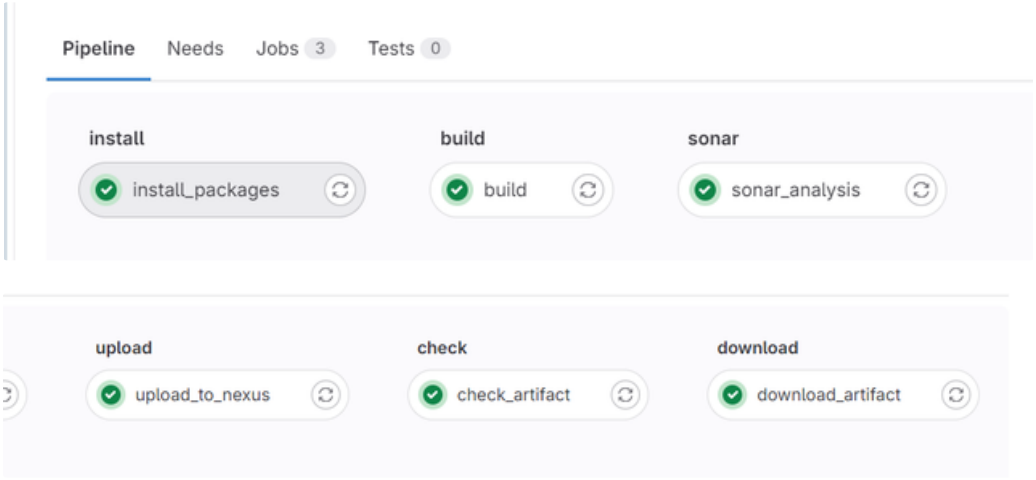


GitLab CI/CD Pipeline Configuration

1. Stages

The `.gitlab-ci.yml` file is organized into stages. Each stage contains jobs that are executed in a specific order. The stages defined are:

- `install`
- `build`
- `sonar`



2. Stage: Install Packages

Purpose: This stage is responsible for installing necessary dependencies required for your project.

Job: `install_packages`

Configuration:

```
install_packages:
  stage: install
  script:
    - echo "Installing packages..."
    - composer install
  tags:
    - shell

```

- **Stage:** Specifies the pipeline stage (`install`).
- **Script:** Commands to be executed for the job.
- **Tags:** Identifies the GitLab Runner with the `shell` tag that should pick this job.

3. Stage: Build

Purpose: This stage is used to build or prepare your project for analysis and deployment.

Job: `build`

Configuration:

```
build:
  stage: build
  script:
    - echo "This is a Lumen project. Build stage executed."
  tags:
    - shell

```

- **Stage:** Specifies the pipeline stage (`build`).

- **Script:** Commands to be executed for the job.
- **Tags:** Identifies the GitLab Runner with the `shell` tag that should pick this job.

SonarQube Stage Configuration

1. GitLab Runner Registration and Setup

You registered a GitLab Runner with the following key points:

- **GitLab Runner Registration:**

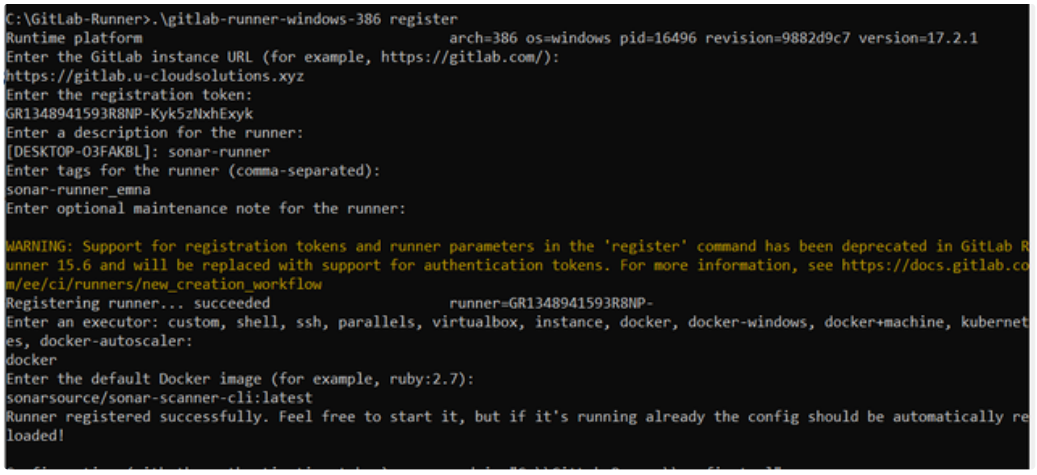
```
.\gitlab-runner-windows-386 register
```

  This command is used to register a new runner with your GitLab instance.
- **GitLab Instance URL and Registration Token:**
  - **GitLab Instance URL:** `https://gitlab.u-cloudsolutions.xyz`
  - **Registration Token:** You provided this token during the registration process.
- **Description and Tags:**
  - **Description:** `sonar-runner`
  - **Tags:** `sonar-runner_emna`
- **Executor Choice:**

```
docker
```

  You chose the Docker executor, which allows the runner to execute jobs within Docker containers.
- **Default Docker Image:**

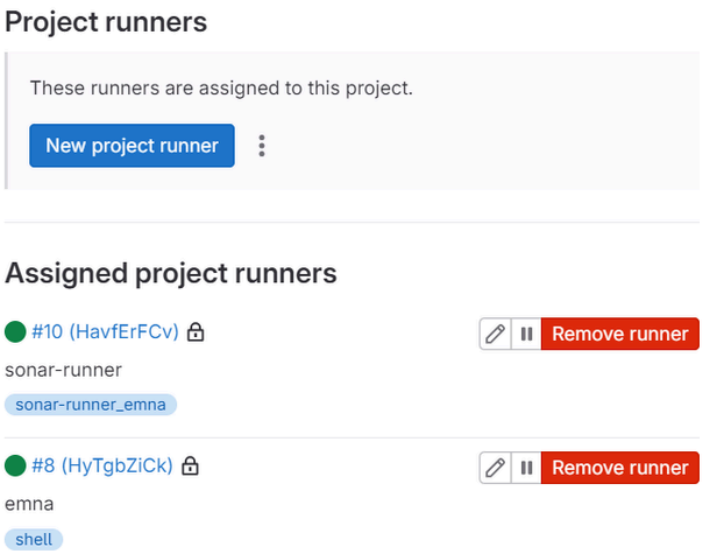
```
sonarsource/sonar-scanner-cli:latest
```



Steps Taken:

1. **Registering the Runner:** You registered the GitLab Runner and associated it with the `sonar-runner_emna` tag. This tag is used to identify this runner for specific jobs in your pipeline.
2. **Docker Image Configuration:** The runner is configured to use the Docker executor with the `sonarsource/sonar-scanner-cli:latest` image. This image contains the necessary tools to perform SonarQube analysis.

3. **Runner Configuration File:** After registration, the runner's configuration was saved in `C:\GitLab-Runner\config.toml`. This file contains details about the runner's settings, including the Docker image and tags.



2. SonarQube Analysis Stage in `.gitlab-ci.yml`

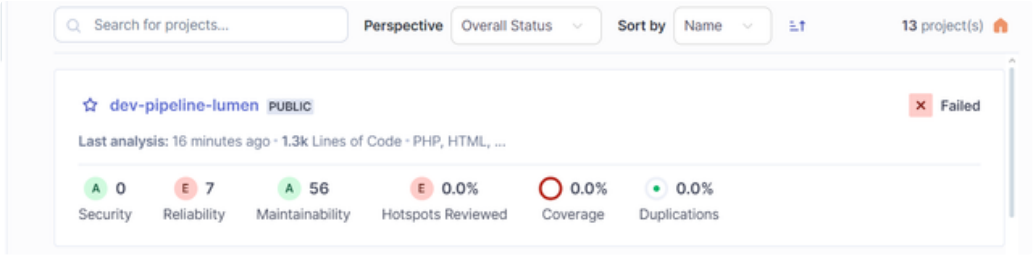
Configuration:

```
sonar_analysis:
  stage: sonar
  script:
    - echo "Running SonarQube analysis..."
    - sonar-scanner -Dsonar.projectKey=dev-pipeline-lumen -Dsonar.projectName=dev-pipeline-lumen -Dsonar.sources=
  tags:
    - sonar-runner_emna

```

Explanation:

1. **Stage:**
   - The `sonar_analysis` job is part of the `sonar` stage, which is specifically set up for SonarQube analysis.
2. **Script:**
   - **Echo Statement:** Outputs a message indicating that SonarQube analysis is starting.
   - **SonarQube Scanner Command:**
     - `sonar-scanner`: Command to run SonarQube analysis.
     - **Parameters:**
       - `-Dsonar.projectKey`: Unique identifier for the project in SonarQube.
       - `-Dsonar.projectName`: Display name of the project.
       - `-Dsonar.sources`: Directory containing source code to analyze.
       - `-Dsonar.host.url`: URL of the SonarQube server.
       - `-Dsonar.login`: Authentication token for accessing SonarQube.
       - `-Dsonar.exclusions`: Specifies files or directories to exclude from analysis.
3. **Tags:**

- `sonar-runner_emna`: Ensures that this job is picked up by the runner registered with this tag.



Package Artifact

In this stage, the project is packaged into a ZIP file. This is a critical step before uploading the artifact to Nexus or any other repository.

Configuration:

- **Stage Name:** `package`
- **Purpose:** To create a compressed archive of the project files.
- **Script:**

```
package_project:
  stage: package
  script:
    - echo "Packaging project..."
    - mkdir -p artifacts
    - zip -r artifacts/project.zip .
  artifacts:
    paths:
      - artifacts/project.zip
  tags:
    - shell
```

Explanation:

1. **Create Artifacts Directory:** The `mkdir -p artifacts` command ensures that the `artifacts` directory exists where the ZIP file will be stored.
2. **Create ZIP Archive:** The `zip -r artifacts/project.zip .` command creates a ZIP file named `project.zip` that includes all files in the project directory.
3. **Artifacts Path:** The `artifacts` section specifies that the created ZIP file should be kept as an artifact, making it available for subsequent stages like upload.

Upload to Nexus:

The `upload_to_nexus` stage in the GitLab CI/CD pipeline handles the uploading of the packaged artifact (`project.zip`) to the Nexus repository. This stage ensures that the build artifacts are stored in a central repository for versioning and further use in deployment pipelines.

Pipeline Configuration:

1. **Variables Configuration:** The following variables are defined for the `upload_to_nexus` stage:
   - `NEXUS_URL`: The base URL of the Nexus repository.
   - `NEXUS_REPOSITORY`: The target repository in Nexus where artifacts will be uploaded.
   - `MAVEN_GROUP_ID`: The Maven group ID of the project, which is used to structure the repository path.
   - `MAVEN_ARTIFACT_ID`: The Maven artifact ID of the project.

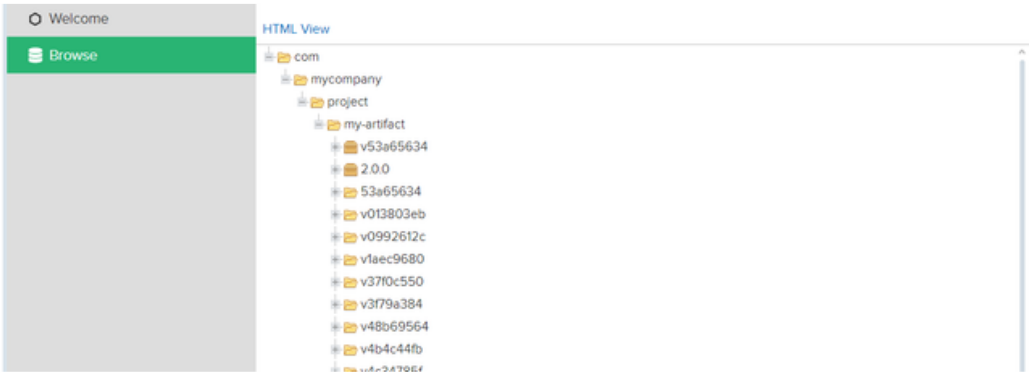
YAML Configuration:

Here is the YAML configuration for the `upload_to_nexus` stage:

```
upload_to_nexus:
  stage: upload
  variables:
    NEXUS_URL: 'https://nexus.u-cloudsolutions.xyz'
    NEXUS_REPOSITORY: 'student-repository'
    MAVEN_GROUP_ID: 'com.gitlab.project'
    MAVEN_ARTIFACT_ID: 'my-artifact'
  script:
    - echo "Uploading artifacts/project.zip to Nexus..."
    - |
      CURL_CMD="curl -u ${NEXUS_USERNAME}:${NEXUS_PASSWORD} --upload-file artifacts/project.zip"
      UPLOAD_URL="${NEXUS_URL}/repository/${NEXUS_REPOSITORY}/$(echo ${MAVEN_GROUP_ID} | tr '.' '/')/${MAVEN_ART
      echo $CURL_CMD $UPLOAD_URL
      $CURL_CMD $UPLOAD_URL
    - echo "Upload complete with commit ID ${CI_COMMIT_SHA} and tag ${CI_COMMIT_REF_NAME}"
  tags:
    - shell


```

Notes:

- The `MAVEN_GROUP_ID` is transformed to replace dots with slashes to fit the Nexus repository path structure.
- The `curl` command uploads the `project.zip` file to the constructed URL using the provided Nexus credentials.
- The stage ensures the proper handling and storage of build artifacts in the Nexus repository, facilitating artifact management and version control.



Stage: Check Artifact in Nexus

Purpose: Checks if a specific artifact exists in the Nexus Repository by making a request and inspecting the HTTP response code.

Script:

```
check_artifact:
  stage: check
  script:
    - echo "Checking if artifact exists in Nexus..."
    - |
      # Replace dots with slashes in MAVEN_GROUP_ID
```

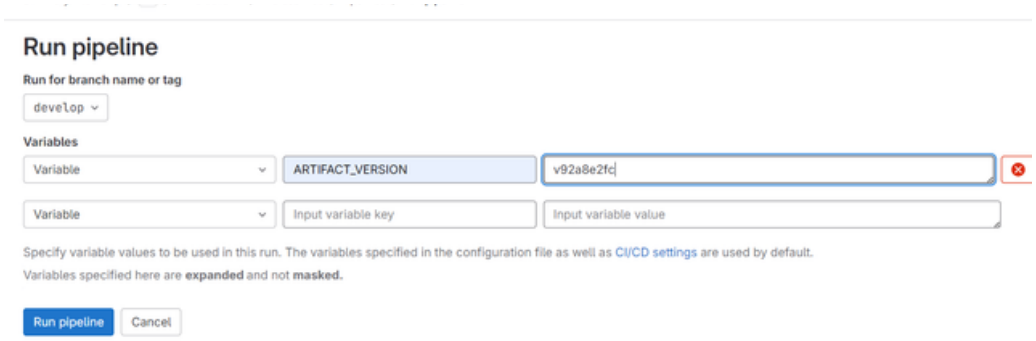
```
      $MAVEN_GROUP_ID_PATH = $env:MAVEN_GROUP_ID -replace '\.', '/'

      # Use the parameter ARTIFACT_VERSION for versioning
      $VERSION_TAG = $env:ARTIFACT_VERSION

      # Construct the Nexus URL to check the artifact
      $NEXUS_CHECK_URL = "${env:NEXUS_URL}/repository/${env:NEXUS_REPOSITORY}/${MAVEN_GROUP_ID_PATH}/${env:MAVEN

      # Print URL for debugging
      echo "Checking URL: $NEXUS_CHECK_URL"

      # Check if the artifact is available using curl
      $RESPONSE_CODE = curl -s -o /dev/null -w "%{http_code}" -u "$env:NEXUS_USERNAME:$env:NEXUS_PASSWORD" "$NEX

      if ($RESPONSE_CODE -eq "200") {
        echo "Artifact found in Nexus."
      } else {
        echo "Artifact not found. HTTP response code: $RESPONSE_CODE"
      }
  tags:
    - shell
  rules:
    - if: '$ARTIFACT_VERSION'
      when: always
```

Explanation:

- **Replace Dots with Slashes:** Converts Maven group ID into a path format.
- **VERSION_TAG:** Fetches the version tag from the pipeline environment variables.
- **NEXUS_CHECK_URL:** Constructs the URL to check if the artifact exists.
- **Curl Command:** Performs a request to check the HTTP response code, indicating artifact presence.

Setting VERSION_TAG Manually: To manually set the `ARTIFACT_VERSION` (which represents the `VERSION_TAG`) when running the pipeline:

1. **Go to GitLab:**
   - Navigate to your project in GitLab.
2. **Navigate to CI/CD Pipelines:**
   - Click on `CI/CD` in the left sidebar.
   - Select `Pipelines`.
3. **Trigger a New Pipeline:**
   - Click on the `Run pipeline` or `Create pipeline` button.
4. **Add Pipeline Variables:**
   - In the "Variables" section, add a new variable:
     - **Key:** `ARTIFACT_VERSION`
     - **Value:** Specify the version tag you want to check, for example, `v92a8e2fc` in our case .
5. **Run the Pipeline:**
   - Click `Create pipeline` or `Run pipeline` to start the pipeline with the specified variables.



Download Artifact

Purpose: To download a specific version of the artifact from Nexus and save it to a defined path.

Configuration:

Stage Name: download

Purpose: To retrieve the packaged artifact from the Nexus repository for further use or verification.

Script:

```
download_artifact:
  stage: download
  script:
    - echo "Downloading artifact from Nexus..."
    - |
      # Convert Maven group ID from dots to slashes
      $MAVEN_GROUP_ID_PATH = $env:MAVEN_GROUP_ID -replace '\.', '/'

      # Retrieve the artifact version
      $VERSION_TAG = $env:ARTIFACT_VERSION

      # Build the Nexus URL for downloading the artifact
      $NEXUS_DOWNLOAD_URL = "${env:NEXUS_URL}/repository/${env:NEXUS_REPOSITORY}/${MAVEN_GROUP_ID_PATH}/${env:MA

      # Print URL for debugging
      echo "Downloading from URL: $NEXUS_DOWNLOAD_URL"

      # Download the artifact using curl
      curl -o "$env:DOWNLOAD_PATH" "$NEXUS_DOWNLOAD_URL"

      echo "Artifact downloaded to ${env:DOWNLOAD_PATH}"
  tags:
    - shell
  rules:
    - if: '$ARTIFACT_VERSION'
      when: always

```

Explanation:

1. **Format Maven Group ID:**
   - Converts `MAVEN_GROUP_ID` from dots to slashes to match Nexus URL format.
2. **Retrieve Version Tag:**
   - Uses `ARTIFACT_VERSION` to specify which version of the artifact to download.
3. **Construct URL:**

- Builds the Nexus URL for the artifact using Nexus base URL, repository name, Maven group ID path, artifact ID, and version tag.

4. **Download Artifact:**
   - Uses `curl` to download the artifact from the constructed URL and saves it to `DOWNLOAD_PATH`.
5. **Print Download Path:**
   - Confirms where the artifact was saved for verification.

Unzip Stage in CI/CD Pipeline

Stage Name: `unzip_artifact`

Purpose: Extract the contents of a ZIP file for further processing.

Configuration

```
# Stage: Unzip Artifact
unzip_artifact:
  stage: unzip
  script:
    - echo "Unzipping artifact..."
    - |
      # Create a directory for unzipped files
      $EXTRACT_PATH = "artifacts/unzipped"
      mkdir -p $EXTRACT_PATH

      # Determine which ZIP file to use
      if ($env:ARTIFACT_VERSION) {
        $ZIP_FILE = "$env:DOWNLOAD_PATH"  # Use downloaded artifact if version is specified
      } else {
        $ZIP_FILE = "$env:FALLBACK_ZIP_PATH"  # Fallback to packaged artifact
      }

      # Unzip the chosen artifact
      unzip "$ZIP_FILE" -d $EXTRACT_PATH

      echo "Artifact extracted to $EXTRACT_PATH"
  tags:
    - shell
  rules:
    - if: '$ARTIFACT_VERSION || $CI_COMMIT_REF_NAME'
      when: always

```

Explanation

- **Stage:** Defined as `unzip`.
- **Create Directory:** Ensures `artifacts/unzipped` directory exists.
- **Determine ZIP File:**
  - If `ARTIFACT_VERSION` is set, use the downloaded artifact.
  - Otherwise, use the fallback ZIP file.
- **Unzip Artifact:** Extracts the chosen ZIP file to `artifacts/unzipped`.
- **Tags and Rules:**
  - Tagged with `shell`.
  - Runs if `ARTIFACT_VERSION` or `CI_COMMIT_REF_NAME` is set.

Build Docker Image

Stage Name: `build_image`
Job Name: `build_docker_image`
Purpose: This stage builds the Docker image from the source code in the repository. The Docker image is tagged using the commit ID to ensure that each image is uniquely identified.

Script:

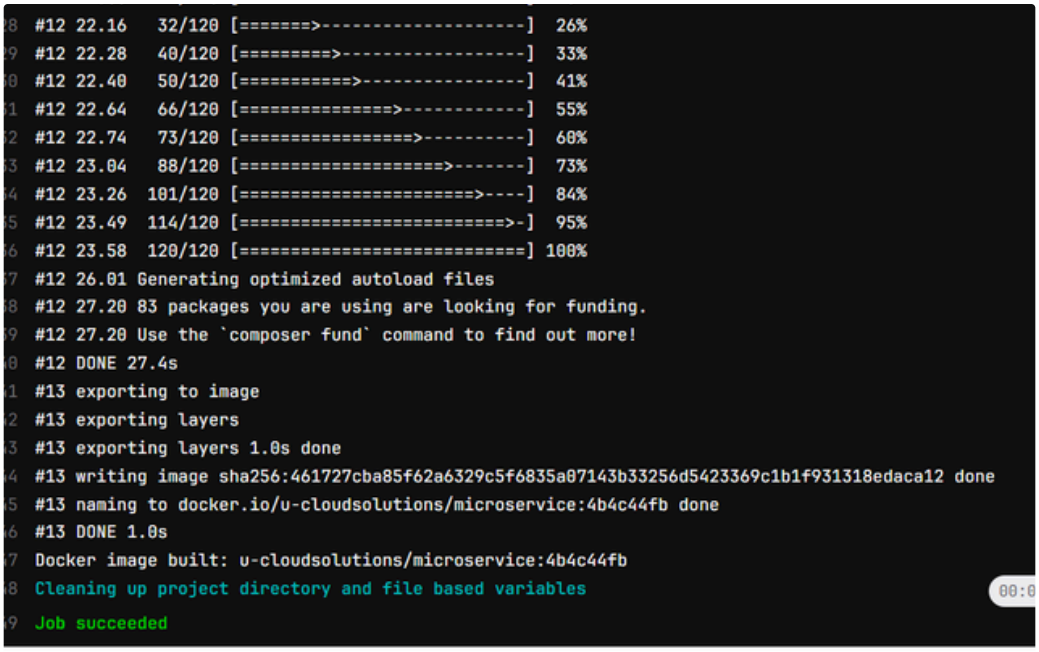
```
build_docker_image:
  stage: build_image
  script:
    - echo "Building Docker image..."
    - |
      # Define Docker image name and tag using the commit ID
      $IMAGE_TAG = $env:CI_COMMIT_SHA.Substring(0, 8)
      $IMAGE_NAME = "u-cloudsolutions/microservice:${IMAGE_TAG}"

      # Build the Docker image
      docker build -t $IMAGE_NAME .

      echo "Docker image built: $IMAGE_NAME"
  tags:
    - shell
  rules:
    - if: '$CI_COMMIT_SHA'
      when: always

```

Details:

- **Docker Image Name:** Formed using the environment variable `AA` and the short commit ID.
- **Docker Image Tag:** Consists of the commit ID's first 8 characters.
- **Commands:**
  - `docker build -t $IMAGE_NAME .` builds the Docker image.
  - The image is tagged with the specified name and commit ID.

Configuration:

- Ensure that the Docker daemon has the necessary permissions to build images and access the source code.



---

2. Push Docker Image to Nexus

Stage Name: `push_image`

Job Name: `push_docker_image`

Purpose: This stage publishes the Docker image built in the previous stage to the Nexus Docker registry. The image is tagged appropriately for the Nexus repository before being pushed.

Script:

```
push_docker_image:
  stage: push_image
  script:
    - echo "Publishing Docker image to Nexus..."
    - |
      # Define Docker registry URL and image name
      $DOCKER_REGISTRY_URL = "${env:NEXUS_DOCKER_URL}/${env:NEXUS_DOCKER_REPOSITORY}"
      $IMAGE_TAG = $env:CI_COMMIT_SHA.Substring(0, 8)
      $IMAGE_NAME = "u-cloudsolutions/microservice:${IMAGE_TAG}"
      $IMAGE_TAGGED = "${DOCKER_REGISTRY_URL}/u-cloudsolutions/microservice:${IMAGE_TAG}"

      # Log in to Docker registry securely
      echo "${env:NEXUS_PASSWORD}" | docker login $env:NEXUS_DOCKER_URL --username $env:NEXUS_USERNAME --passwor

      # Tag and push Docker image
      docker tag $IMAGE_NAME $IMAGE_TAGGED
      docker push $IMAGE_TAGGED

      echo "Docker image $IMAGE_TAGGED published successfully."
  tags:
    - shell
  rules:
    - if: '$CI_COMMIT_SHA'
      when: always

```

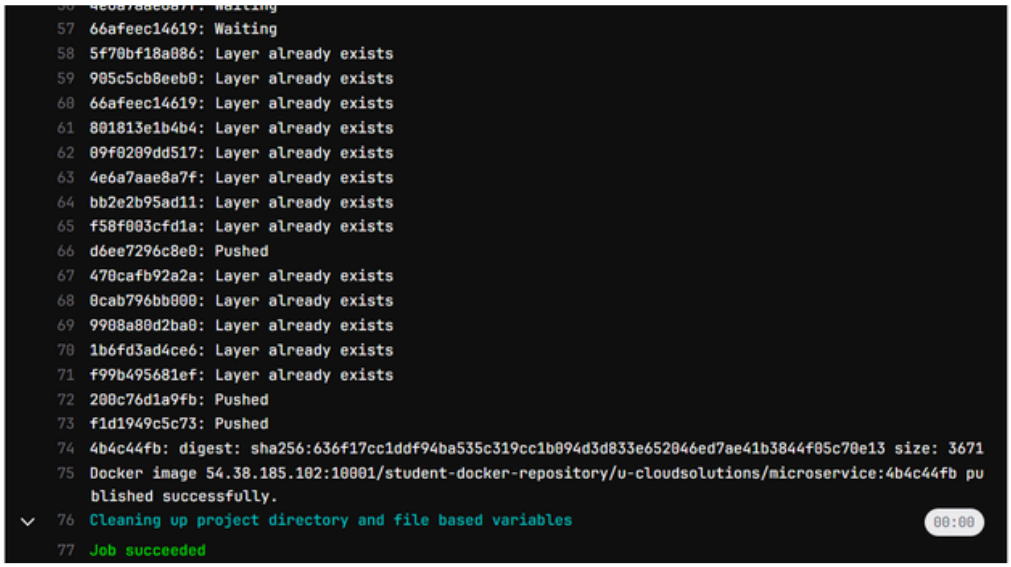
Details:

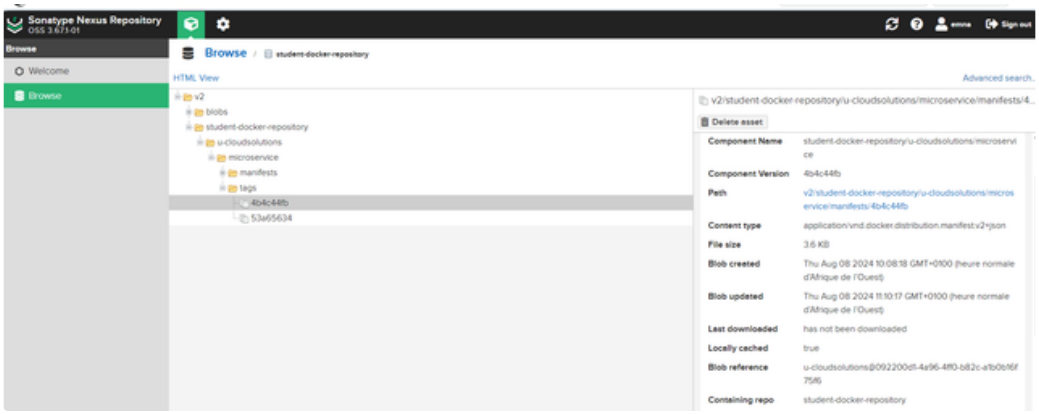
- **Docker Registry URL:** Defined by the environment variables `NEXUS_DOCKER_URL` and `NEXUS_DOCKER_REPOSITORY`.
- **Docker Image Name:** The name of the Docker image with the tag from the commit ID.
- **Commands:**
  - `docker login` logs in to the Nexus Docker registry.
  - `docker tag` tags the image with the Nexus registry URL.
  - `docker push` pushes the tagged image to the Nexus registry.

Configuration:

- Ensure that the Nexus Docker registry is accessible from the build environment and configured to accept incoming requests.
- Verify that the Docker registry URL, username, and password are correct and securely managed.

added "u-cloudsolutions.xyz:10001" in the insecure-registries in docker engine





Build JMeter Docker Image

Stage: `build_jmeter_image`

Purpose: Build a Docker image specifically for running JMeter tests.

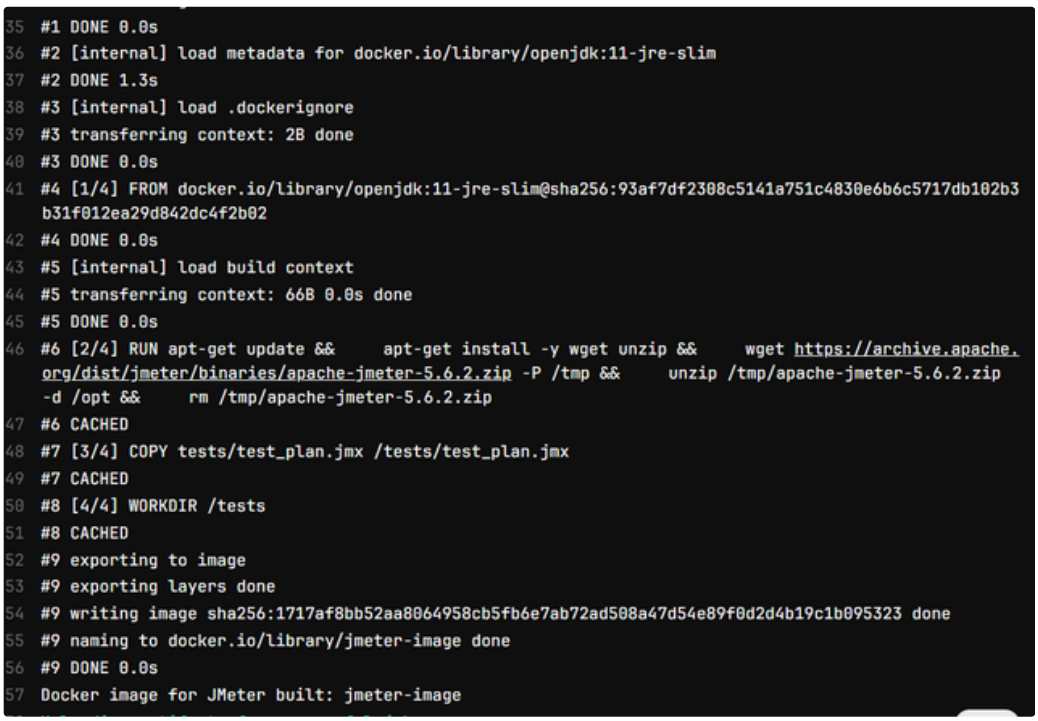
Script:

```
build_jmeter_image:
  stage: build_jmeter_image
  script:
```

```
    - echo "Building Docker image for JMeter..."
    - |
      # Build Docker image for JMeter
      docker build -f tests/Dockerfile.jmeter -t $JMETER_IMAGE_NAME .
      echo "Docker image for JMeter built: $JMETER_IMAGE_NAME"
      echo $JMETER_IMAGE_NAME > img.txt
  tags:
    - shell
  artifacts:
    paths:
      - img.txt
  rules:
    - if: '$CI_COMMIT_SHA'
      when: always

```

Details:

- **Dockerfile Location**: `tests/Dockerfile.jmeter`
- **Image Name**: `$JMETER_IMAGE_NAME`
- **Artifact**: `img.txt` containing the name of the built JMeter image.



---

2. Deploy Application Container

Stage: `deploy`

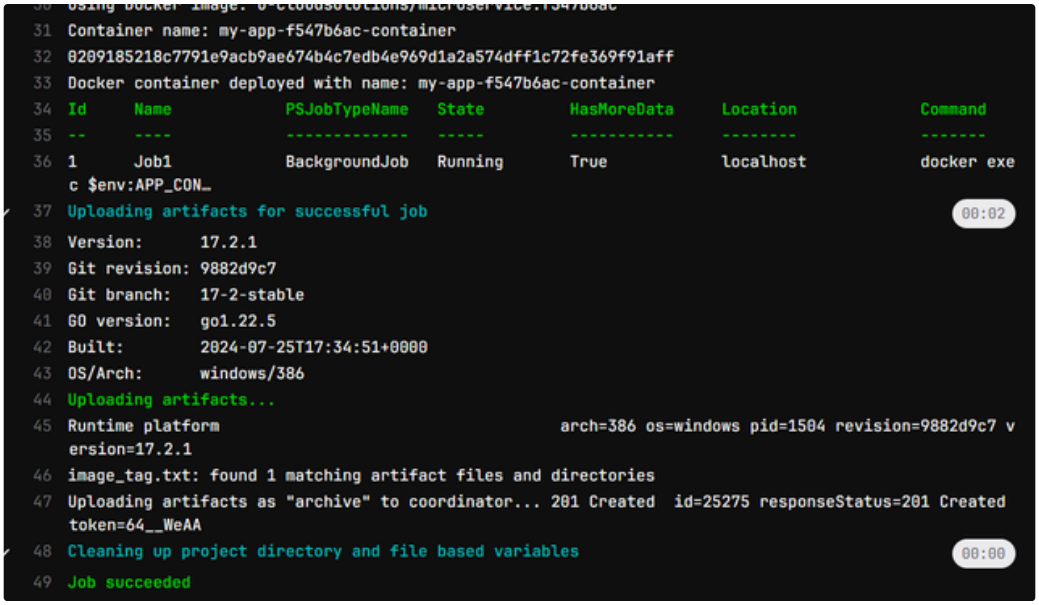
Purpose: Deploy the Docker container with the application for JMeter testing.

Script:

```
deploy_app_container:
  stage: deploy
  script:
    - echo "Deploying Docker container..."
    - |
      $env:IMAGE_TAG = (Get-Content image_tag.txt).Trim()
      $env:APP_CONTAINER_NAME = "my-app-$($env:IMAGE_TAG)-container"
      $env:DOCKER_IMAGE_NAME = "u-cloudsolutions/microservice:$($env:IMAGE_TAG)"
      Write-Host "Using Docker image: $env:DOCKER_IMAGE_NAME"
      Write-Host "Container name: $env:APP_CONTAINER_NAME"

      $ErrorActionPreference = "Stop"

      function Cleanup {
        Write-Host "Cleaning up application container..."
        if (docker ps -q -f "name=$env:APP_CONTAINER_NAME") {
          docker stop $env:APP_CONTAINER_NAME | Out-Null
          docker rm $env:APP_CONTAINER_NAME | Out-Null
          Write-Host "Application container $env:APP_CONTAINER_NAME has been removed."
        }
      }

      try {
        docker run -d --name $env:APP_CONTAINER_NAME -p 10088:80 $env:DOCKER_IMAGE_NAME
        Write-Host "Docker container deployed with name: $env:APP_CONTAINER_NAME"
        Start-Sleep -Seconds 10
        docker exec $env:APP_CONTAINER_NAME php -S localhost:8000 > $null 2>&1 &
      } finally {
        # Cleanup function will be called only if necessary
      }
  tags:
    - shell
  dependencies:
    - build_jmeter_image
  artifacts:
    paths:
      - image_tag.txt
  rules:
    - if: '$CI_COMMIT_SHA'
      when: always

```

Details:

- **Docker Image**: `u-cloudsolutions/microservice:$IMAGE_TAG`
- **Container Name**: `my-app-$IMAGE_TAG-container`
- **Ports**: Mapping container port 80 to host port 10088
- **Cleanup**: Function to stop and remove the container if it exists.



---

3. Run JMeter Tests

Stage: `run_jmeter_tests`

Purpose: Execute JMeter tests against the deployed application container and collect results.

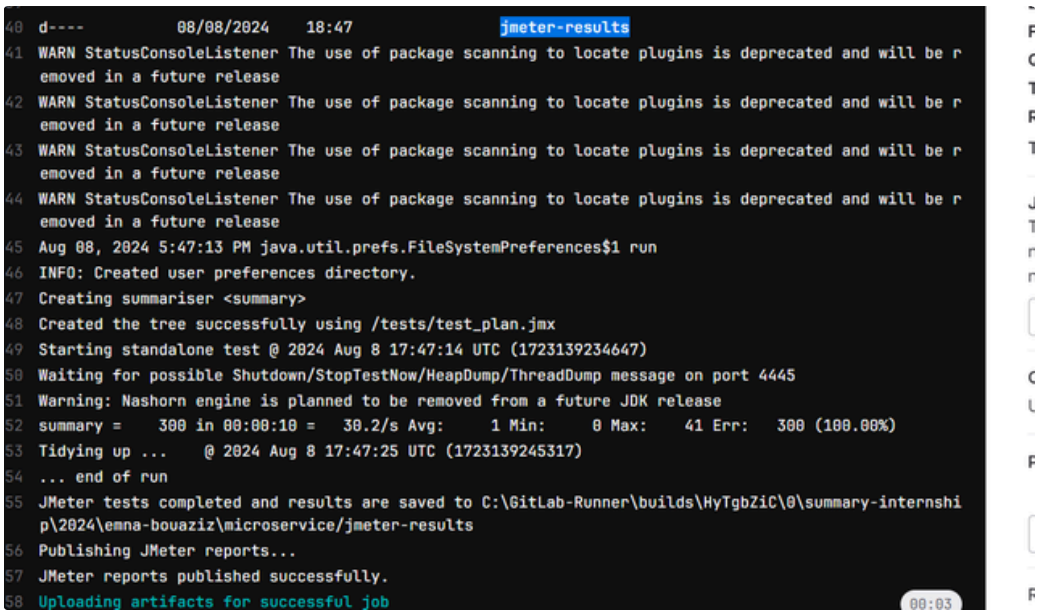
Script:

```
run_jmeter_tests:
  stage: run_jmeter_tests
  script:
    - echo "Running JMeter tests..."
    - |
      $env:IMAGE_TAG = (Get-Content image_tag.txt).Trim()
      $env:APP_CONTAINER_NAME = "my-app-$($env:IMAGE_TAG)-container"
      $env:JMETER_IMAGE_NAME = (Get-Content img.txt).Trim()
      $env:JMETER_RESULTS_DIR = "$(pwd)/jmeter-results"

      $ErrorActionPreference = "Stop"

      function Cleanup {
        Write-Host "Cleaning up application container..."
        if (docker ps -q -f name=$env:APP_CONTAINER_NAME) {
          docker stop $env:APP_CONTAINER_NAME | Out-Null
          docker rm $env:APP_CONTAINER_NAME | Out-Null
        }
        Write-Host "Application container $env:APP_CONTAINER_NAME has been removed."
      }

      try {
        if (-not (Test-Path -Path $env:JMETER_RESULTS_DIR)) {
          New-Item -ItemType Directory -Path $env:JMETER_RESULTS_DIR
        }

        docker run --rm `
          --link $env:APP_CONTAINER_NAME:my-app `
```

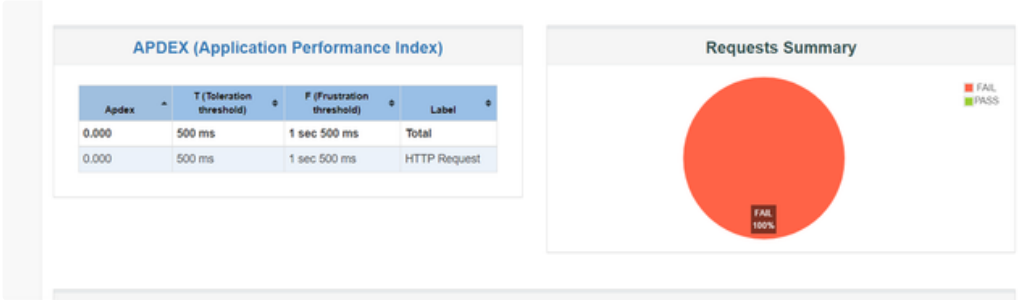
```
          -v "$env:JMETER_RESULTS_DIR:/tests/results" `
          $env:JMETER_IMAGE_NAME `
          -n -t /tests/test_plan.jmx -l /tests/results/results.jtl -e -o /tests/results -Jhostname=my-app -Jport

        Write-Host "JMeter tests completed and results are saved to $env:JMETER_RESULTS_DIR"

        Write-Host "Publishing JMeter reports..."
        Write-Host "JMeter reports published successfully."
      } catch {
        Write-Host "An error occurred: $_"
        exit 1
      }
  tags:
    - shell
  artifacts:
    paths:
      - jmeter-results/
      - image_tag.txt
      - img.txt
    when: always
  rules:
    - if: '$CI_COMMIT_SHA'
      when: always

```

Details:

- **JMeter Image**: Obtained from `img.txt`
- **Results Directory**: `$JMETER_RESULTS_DIR`
- **JMeter Command**: Runs JMeter tests using `test_plan.jmx` and stores results in `results.jtl`



I added the reports as artifacts in gitlab :



---

4. Cleanup

Stage: `cleanup`

Purpose: Clean up resources to ensure no leftover containers or artifacts remain.

Script:

```
cleanup:
  stage: cleanup
  script:
    - echo "Cleaning up application container..."
    - |
      $env:IMAGE_TAG = (Get-Content image_tag.txt).Trim()
      $env:APP_CONTAINER_NAME = "my-app-$($env:IMAGE_TAG)-container"
      Write-Host "Using container name: $env:APP_CONTAINER_NAME"

      # Stop and remove the container using the correct variable
      docker stop $env:APP_CONTAINER_NAME
      docker rm $env:APP_CONTAINER_NAME

      Write-Host "Application container $env:APP_CONTAINER_NAME has been removed."
  tags:
    - shell
  artifacts:
    paths:
      - image_tag.txt
  dependencies:
    - run_jmeter_tests
  rules:
    - if: '$CI_COMMIT_SHA'
      when: always
```

```
25
```

Details:

- **Cleanup Action**: Stops and removes the application container if it exists.
- **Artifact**: `image_tag.txt` to maintain context across stages.