Jenkins CI/CD with Docker Lab Report

Amo Samuel

Introduction

This report outlines the Continuous Integration/Continuous Deployment (CI/CD) pipeline I configured using Jenkins, along with a detailed explanation of the deployment process. The pipeline is designed to automate the process of building, testing, and deploying a Java-based application managed through a Docker Compose setup. My approach ensures efficient and reliable deployment of updates to the application, minimizing manual intervention and reducing the risk of human error.

Jenkins Setup

Before creating the pipeline, I installed Jenkins on my local environment to serve as the CI/CD tool. Jenkins is a powerful automation server that supports building, deploying, and automating software development workflows. After setting up Jenkins, I created a new pipeline project through the Jenkins web interface, where I specified my GitHub repository and linked the Jenkinsfile to the pipeline.

Pipeline Overview

The Jenkins pipeline I created is structured as follows:

Checkout Stage

Objective: This stage clones the code from the specified GitHub repository.

Details: The pipeline checks out the main branch of the repository to ensure that the latest version of the code is used for the build and deployment process.

Build Stage

Objective: The application is built using Maven.

Details: The ./mvnw clean package command is executed to clean the project, compile the source code, run tests, and package the compiled code into a deployable artifact. This ensures that the application is always built from a clean state.

Deploy with Docker Compose Stage

Objective: The application is deployed using Docker Compose.

Details: In this stage, the pipeline brings down any existing Docker containers, rebuilds the Docker images, and then brings up the containers in detached mode. This process ensures that the latest version of the application is always running in the Docker environment.

Post-Execution Actions

The pipeline is configured with post-execution actions to handle the outcome of the pipeline run:

Success Notification: If the pipeline completes successfully, Jenkins outputs a message indicating that the pipeline has been completed without issues.

Failure Notification: If any stage of the pipeline fails, Jenkins outputs a failure message, providing immediate feedback on the pipeline status.

Takeaways and Lessons Learned

Efficiency Through Automation: By automating the build and deployment process with Jenkins, I was able to significantly reduce the time and effort required to deploy updates to the application. This automation ensures consistency and reliability, as the same steps are followed every time the pipeline is executed.

Docker Compose Integration: Using Docker Compose for deployment simplifies the management of containerized applications. The ability to bring down, rebuild, and bring up containers within a single pipeline stage makes the deployment process seamless and reduces downtime.

Scalability Considerations: While the current setup is sufficient for smaller projects, scaling this approach for larger, more complex applications might require additional considerations, such as integrating Kubernetes for orchestration or implementing advanced CI/CD strategies like blue-green or canary deployments.

Continuous Improvement: This pipeline serves as a strong foundation for continuous integration and deployment. However, there's always room for improvement. Future enhancements could include adding automated tests, integrating code quality checks, or setting up notifications to alert the team of pipeline statuses.

Conclusion

The Jenkins pipeline I set up successfully automates the CI/CD process for the Java application in my GitHub repository. This setup not only ensures that the application is always up-to-date but also streamlines the entire build and deployment process. Going forward, I plan to iterate on this pipeline, adding more sophisticated features and refining the deployment process based on the lessons learned.