Implementing a DevOps Pipeline with Kubernetes, Jenkins, Maven, Tomcat, and Spring Boot on Windows via Docker Desktop

A Project Based Learning Report Submitted in partial fulfilment of the requirements for the award of the degree

of

Bachelor of Technology

in The Department of

Computer Science And Engineering

CI/CD And Cloud DevOPS

Submitted by

2310030012: Ruthwik PVN

2310030111: Aniket OJHA

230031263: Ayush Kumar

Under the guidance of

Dr M Trinath Basu

Department of Electronics and Communication Engineering

Koneru Lakshmaiah Education Foundation, Aziz Nagar

Aziz Nagar – 500075

FEB - 2025.

Introduction

In the current era of software engineering, delivering applications quickly and reliably has become a top priority. Organizations no longer depend on traditional software development lifecycles, which often involved long release cycles, manual testing, and complex deployment processes. Instead, the focus has shifted towards automation, scalability, and continuous delivery. DevOps is the cultural and technical movement that bridges the gap between development teams and operations teams. It ensures that software can be developed, tested, and deployed more frequently and with fewer errors. The goal of DevOps is not only to increase the speed of delivery but also to maintain high reliability and performance in production environments.
To implement DevOps successfully, a set of tools and technologies are required. Each tool addresses a different stage of the software delivery pipeline:

- Docker provides lightweight, portable containers that ensure applications run consistently across environments.
- Kubernetes manages and orchestrates these containers at scale, handling tasks such as scaling, failover, and load balancing.
- Apache Tomcat acts as the application server for running Java-based web applications, ensuring a reliable environment for deployment.
- Apache Maven automates the build and dependency management process, reducing manual effort and improving reproducibility.
- Jenkins acts as the automation engine for continuous integration (CI) and continuous delivery (CD), integrating seamlessly with the above tools to automate the entire lifecycle.

The combination of these tools forms a mini DevOps ecosystem on a single machine. While large enterprises use complex, distributed systems across cloud platforms, setting up this environment on a Windows PC provides an excellent starting point for learning DevOps principles in practice.

Objectives of this Project:

1. To install and configure Docker Desktop on Windows.
2. To enable Kubernetes through Docker Desktop and verify cluster functionality.
3. To set up Apache Tomcat as a local web server for Java applications.
4. To install Apache Maven for building and managing Java projects.
5. To install Jenkins and configure it as an automation server for continuous integration and delivery.
6. To demonstrate a working environment that integrates these tools into a coherent DevOps workflow.

Literature Review/ Application Survey

The adoption of DevOps practices has grown rapidly in recent years as organizations seek to improve the speed, reliability, and scalability of software delivery. Several studies and industry reports highlight the benefits of combining tools such as Docker, Kubernetes, Jenkins, Maven, and Tomcat to create a complete automation environment.

I. 1. Evolution of DevOps

According to Humble & Farley (2010), Continuous Integration (CI) and Continuous Delivery (CD) are core practices that reduce integration issues and accelerate delivery cycles. Jenkins, one of the most widely used CI/CD servers, emerged from the Hudson project in 2011 and has since become a cornerstone in DevOps pipelines.

II. 2. Containerization and Orchestration

Docker introduced containerization in 2013, which solved the classic problem of "it works on my machine" by ensuring applications run consistently across environments. Studies by Pahl (2015) emphasize how containerization reduces overhead compared to traditional virtual machines.

However, running containers at scale required orchestration. Kubernetes, initially developed by Google and later donated to the Cloud Native Computing Foundation (CNCF), has become the industry standard. Literature by Burns et al. (2016) describes Kubernetes as an enabler for fault-tolerant, scalable, and self-healing systems.

III. 3. Application Servers (Tomcat)

Tomcat, maintained by the Apache Software Foundation, has been widely adopted since the early 2000s. According to Apache's own usage reports, Tomcat powers a significant portion of Java web applications due to its lightweight and reliable architecture. Its simplicity and compatibility with Java Servlets and JSP make it a preferred choice in both academic and enterprise environments.

IV. 4. Build Automation (Maven)

Build tools play a crucial role in modern software engineering. Maven, introduced in 2004, standardized the build lifecycle through the Project Object Model (POM). Research by Kivistö (2012) highlights how Maven's dependency management reduced complexity in large-scale Java projects. Compared to Ant, Maven brought better automation, reproducibility, and easier integration with CI/CD pipelines.

V. 5. Jenkins in DevOps Pipelines

Jenkins provides flexibility through its plugin ecosystem, supporting integration with Docker, Kubernetes, and Maven. Studies (Shahin et al., 2017) on DevOps practices emphasize Jenkins' role in reducing manual intervention, increasing test automation, and improving software quality. Jenkins pipelines further allow infrastructure as code (IaC), making build and deployment processes repeatable and reliable.

VI. 6. Industry Trends and Case Studies

- Google, Netflix, and Amazon have showcased the benefits of container orchestration using Kubernetes, achieving high availability and scalability.
- Many open-source projects (e.g., Spring, Hibernate) use Maven for dependency management, reflecting its wide adoption.
- Jenkins remains the most used CI/CD tool in surveys by Stack Overflow and DevOps Research & Assessment (DORA).

System Requirements

Before starting the installation, the following requirements were ensured:

- **Operating System**: Windows 10/11
- **Hardware**: Minimum 8 GB RAM, i5 or higher processor
- **Software Dependencies**:
    - Docker Desktop
    - Kubernetes (enabled through Docker Desktop)
    - Java (JDK 8 or above)
    - Apache Tomcat
    - Apache Maven
    - Jenkins (LTS version)

Tools and Technologies

1. **Docker Desktop with Kubernetes**
    - **Docker** provides containerization, ensuring applications run uniformly across environments.

2. **Kubernetes**
    - **Kubernetes** orchestrates containers, enabling scaling, load balancing, and fault tolerance.
3. **Apache Tomcat**
    - Open-source servlet container.
    - Used for deploying Java-based applications in .war format.
4. **Apache Maven**
    - Build automation tool for Java projects.
    - Manages dependencies and project lifecycles.
5. **Jenkins**
    - Automation server for CI/CD.
    - Integrates with GitHub, Maven, Docker, and Kubernetes for automated workflows.

Installation and Setup

1. Installing Docker Desktop & Enabling Kubernetes

- Download Docker Desktop from the official website.
- Install Docker Desktop on Windows.
- Enable **Kubernetes** from Docker Desktop settings.
- Verify installation using commands:
- docker --version
- kubectl version

2. Running a Basic Kubernetes Deployment

The running of Kubernetes Deployment involves executing the following commands

- kubectl create deployment hello-node --image=k8s.gcr.io/echoserver:1.4
- kubectl expose deployment hello-node --type=LoadBalancer --port=8080
- kubectl get pods

3. Installing Tomcat

- Download Apache Tomcat.
- Extract and set environment variables (CATALINA_HOME).
- Start Tomcat by running startup.bat.
- Verify by visiting http://localhost:8080.

4. Installing Maven

- Download Apache Maven.
- Extract and set environment variables (MAVEN_HOME, PATH).
- Verify installation using mvn -v.

5. Installing Jenkins

- Download Jenkins .war file.
- Run Jenkins using:
- java -jar jenkins.war --httpPort=9090
- Unlock Jenkins with the initial admin password.
- Install suggested plugins.
- Create the first admin user.

6. Installing Spring Boot

- Installed **Spring Initializr** via Maven/STS.
- Created a simple Spring Boot application.
- Ran the application with:
- mvn spring-boot:run
- Application started on an embedded Tomcat server.

Workflow Demonstration

To demonstrate the integration of all the tools, a complete workflow was implemented that simulates how modern DevOps pipelines function. The process begins when a developer

commits new code to a GitHub repository. This code commit automatically triggers Jenkins through a webhook configured between GitHub and Jenkins. Once Jenkins detects a new change in the repository, it starts the build process by invoking Maven. Maven is responsible for compiling the Java application, running the necessary tests, and generating a build artifact in the form of a .war file.

After the build is successfully completed, Jenkins proceeds to create a Docker image containing the newly built application. This Docker image is tagged with a version identifier to maintain version control and then pushed to a container registry such as Docker Hub. Once the image is available in the registry, Kubernetes takes over the deployment process. Kubernetes pulls the Docker image from the registry and deploys it as a pod within the Kubernetes cluster. This pod contains a Tomcat server where the application runs seamlessly. Finally, the application is made accessible to end-users through a Kubernetes service, which provides a stable endpoint for accessing the running application.

By combining these tools, the workflow showcases a complete Continuous Integration and Continuous Deployment (CI/CD) pipeline. The process highlights how automation reduces manual intervention and accelerates the delivery of applications. Additionally, with the inclusion of Spring Boot after Jenkins, the process became even simpler. Spring Boot’s auto-configuration feature made it easier to create REST APIs and microservices, which could then be directly containerized and deployed. This integration demonstrated how microservice-based architectures benefit greatly from DevOps practices, as each microservice can independently follow the same pipeline for deployment.

Advantages of the Setup

The integration of Docker, Kubernetes, Jenkins, Maven, Tomcat, and Spring Boot provides a highly efficient environment for software development and deployment. One of the most significant advantages is **faster deployment cycles**. Traditional approaches to software delivery involve lengthy manual steps for building, testing, and deploying, which often introduce delays and human errors. With Jenkins managing the CI/CD pipeline, these steps are automated, allowing teams to deliver code changes continuously and reliably. The 2019 DORA State of DevOps Report highlights this impact, stating that organizations using automated pipelines deploy 46 times more frequently and recover from failures 96 times faster than those using manual methods. This ability to release small updates quickly gives organizations a strong competitive advantage in today's fast-paced software market.

Another important benefit is **portability**. Docker ensures that applications packaged into containers will run consistently across different systems, whether it is a developer's laptop, a testing server, or a production environment. This solves the common "it works on my machine" problem. According to a 2021 CNCF survey, more than 60% of organizations already combine Docker and Kubernetes to streamline their deployments. This widespread adoption proves that containerization has become the industry standard for delivering modern applications.

Kubernetes extends this advantage by introducing **scalability and orchestration**. Instead of manually provisioning servers or scaling applications, Kubernetes can automatically increase or decrease the number of running containers based on traffic and resource usage. This ensures high availability even under heavy loads. For example, an application running on a single container can be scaled up to hundreds of replicas within seconds during peak demand and then scaled back down during off-hours, ensuring cost efficiency and optimal performance.

Another strength of this setup lies in **automation and reduced errors**. Jenkins, when combined with Maven, creates a seamless workflow where source code changes trigger automatic builds, tests, and deployments.

Conclusion

In conclusion, this project successfully created a mini DevOps environment on a personal Windows machine using Docker Desktop, Kubernetes, Jenkins, Maven, Tomcat, and Spring Boot. Each tool played a crucial role in building a continuous integration and continuous deployment pipeline. Docker provided the containerization layer, Kubernetes handled orchestration, Jenkins automated the build and deployment process, Maven ensured build consistency, Tomcat served as the application server, and Spring Boot simplified application development.

The project provided hands-on experience with real-world DevOps tools, demonstrated the integration of multiple technologies, and offered practical exposure to solving configuration issues. It also emphasized the advantages of DevOps over traditional development approaches, showing why automation, scalability, and reliability are essential in modern software engineering. Overall, the setup forms a strong foundation for future DevOps projects and can be further extended by integrating advanced features such as monitoring, cloud deployment, or security tools to create a production-ready DevOps pipeline.