Project Report: CI/CD Pipeline for Flask Application

Date: April 28, 2025

Abstract

This report details the setup and implementation of a Continuous Integration and Continuous Deployment (CI/CD) pipeline for a sample Python Flask web application. The primary objective was to automate the build, testing (optional), and containerization process using Docker, triggered by code changes pushed to a GitHub repository. The pipeline leverages GitHub Actions for automation and Docker Hub for image storage. Deployment was demonstrated locally using both direct Docker commands and Minikube, simulating a development or testing environment without reliance on cloud infrastructure. The project successfully established an automated workflow, enhancing development efficiency and deployment consistency.

Introduction

The modern software development lifecycle emphasizes speed, reliability, and automation. CI/CD pipelines are fundamental to achieving these goals by automating the integration of code changes and their subsequent deployment. This project aimed to create a practical, self-contained CI/CD pipeline for a simple web application. The focus was on using readily available tools like GitHub Actions and Docker, demonstrating how such a pipeline can be implemented even for local deployment scenarios (e.g., development, testing, or small-scale internal use) without requiring complex cloud service configurations. The outcome is a repeatable process that automatically builds and packages the application into a Docker container upon code commits, making deployment straightforward.

Tools Used

The following tools and technologies were utilized in this project:

Version Control: Git

Code Repository: GitHub

Web Framework: Python Flask (for the sample application)

Containerization: Docker, Dockerfile

Container Registry: Docker Hub

CI/CD Automation: GitHub Actions

Local Orchestration/Deployment:

* Docker Engine (direct container running)
* Minikube (local Kubernetes cluster simulation)
* kubectl (Kubernetes command-line tool)

Configuration: YAML (for GitHub Actions workflow, docker-compose, Kubernetes manifests)

Steps Involved in Building the Project

The construction of the CI/CD pipeline followed these key steps:

1. Application Development: A simple Python Flask web application (`app.py`) was created, along with its dependencies listed in `requirements.txt`.

2. Dockerization: A `Dockerfile` was written to define the steps for building a container image for the Flask application. This included specifying the base Python image, installing dependencies, copying application code, and defining the container's entry point. A `docker-compose.yml` was also included for simplified local development runs.

3. GitHub Repository Setup: A GitHub repository was initialized to host the application code, Dockerfile, and CI/CD workflow configuration.

4. GitHub Actions Workflow Creation: A workflow file (`.github/workflows/main.yml`) was created to define the CI/CD pipeline logic. This workflow was configured to trigger on pushes to the `main` branch.

5. Workflow Configuration: The workflow included steps for:

   * Checking out the source code.

   * Setting up Docker Buildx for efficient image building.

   * Logging into Docker Hub using securely stored credentials (username and access token)

     configured as GitHub Actions secrets (`DOCKERHUB_USERNAME`, `DOCKERHUB_TOKEN`).

   * Building the Docker image using the `Dockerfile`.

   * Tagging the image with `latest` and the unique Git commit SHA for versioning.

   * Pushing the tagged image to the specified Docker Hub repository.

   * Logging out of Docker Hub.

6. Secrets Management: Docker Hub credentials were securely stored as secrets within the GitHub repository settings to avoid hardcoding sensitive information in the workflow file.

7. Triggering and Monitoring: Pushing code changes to the `main` branch automatically triggered the workflow. Progress and logs were monitored via the "Actions" tab in the GitHub repository.

8. Local Deployment: Instructions were provided for deploying the application locally after the image was successfully pushed to Docker Hub:

   * Using Docker: Pulling the image from Docker Hub (`docker pull`) and running it as a container (`docker run`), mapping ports for access.

   * Using Minikube: Starting a local Minikube cluster, creating Kubernetes `Deployment` and `Service` manifest files (referencing the Docker Hub image), applying these manifests using `kubectl`, and accessing the application via the Minikube service URL.

Conclusion

This project successfully demonstrated the creation and operation of a complete CI/CD pipeline using GitHub Actions and Docker for a Python Flask application. The pipeline effectively automates the process of building and distributing the application as a container image upon code changes. By integrating standard tools like Docker, Docker Hub, and GitHub Actions, the project provides a practical template for automating development workflows. The inclusion of local deployment methods using both Docker and Minikube highlights the versatility of containerization for various environments, including development and testing, without necessitating immediate cloud deployment. This automated approach significantly improves developer productivity, ensures consistent builds, and simplifies the deployment process.