# Comprehensive Documentation: Docker and Jenkins

This documentation combines both Docker and Jenkins pipeline configurations to provide a holistic view of the Continuous Integration (CI) and Continuous Deployment (CD) process. The project consists of the following services, orchestrated using Docker Compose and managed via a Jenkins pipeline:

1. **Backend Service**: A Flask-based application that provides stock-related data and metrics for monitoring.
2. **Frontend Service**: An Angular application, served using Nginx.
3. **Prometheus**: A monitoring tool that scrapes metrics exposed by the backend.
4. **Grafana**: A visualization tool that connects to Prometheus to create monitoring dashboards.

This documentation covers:

* Dockerfile configuration for each service (Backend, Frontend).
* The docker-compose.yml file for managing services.
* Jenkins Pipeline integration for building, tagging, and pushing Docker images to Docker Hub.

**Table of Contents**

1. [Docker Configuration](#docker-configuration)
   * [Backend Dockerfile](#backend-dockerfile)
   * [Frontend Dockerfile](#frontend-dockerfile)
2. [Docker Compose Configuration](#docker-compose-configuration)
3. [Jenkins Pipeline Configuration](#jenkins-pipeline-configuration)
4. [Using Jenkins and Docker Together](#using-jenkins-and-docker-together)
5. [Post Actions](#post-actions)
6. [Troubleshooting](#troubleshooting)

**Docker Configuration**

**Backend Dockerfile**

**File Path**: ./backend/Dockerfile

This Dockerfile sets up a lightweight Python environment for running the Flask backend service.

A screenshot of a computer program

Description automatically generated

**Key Points:**

* **Base Image**: Uses the lightweight python:3.9-slim image to reduce the image size.
* **Working Directory**: All files are placed inside /app in the container.
* **Exposing Port**: The Flask app listens on port 8001.
* **Flask Application**: The environment variable FLASK\_APP=app.py ensures that Flask knows which file to execute when starting the application.

**Frontend Dockerfile**

**File Path**: ./frontend/Dockerfile

This is a multi-stage build for the Angular frontend application.

A screenshot of a computer program

Description automatically generated

**Key Points:**

* **Multi-Stage Build**: The first stage builds the Angular app, and the second stage uses Nginx to serve the static files.
* **Angular CLI**: The CLI is installed globally for building the project.
* **Nginx**: Nginx is used to serve the Angular app via port 80.

**Docker Compose Configuration**

**File Path**: docker-compose.yml

Docker Compose is used to manage the backend, frontend, Prometheus, and Grafana services. Each service is configured with build contexts, environment variables, and port mappings.

A screenshot of a computer screen

Description automatically generated**Key Points:**

* **Backend**: Flask app is exposed on port 8001.
* **Frontend**: Angular app is exposed on port 3001, serving through Nginx.
* **Prometheus**: Monitors metrics and is accessible via port 9090.
* **Grafana**: Visualization tool accessible via port 3000.
* **Network**: All services are connected through the monitoring network for internal communication.

**Jenkins Pipeline Configuration**

The Jenkins pipeline automates building, tagging, and pushing Docker images for both the backend and frontend services. It fetches the code from GitHub, builds the Docker images, and pushes them to Docker Hub.

**Key Points:**

* **Docker Credentials**: Docker Hub credentials are securely stored in Jenkins and referenced via DOCKER\_CREDENTIALS\_ID.
* **Build and Tagging**: Both the backend and frontend Docker images are built and tagged with the Jenkins build number.
* **Push to Docker Hub**: Once built, the images are pushed to Docker Hub.
* **Clean Workspace**: After each pipeline run, the workspace is cleaned up to ensure no leftover files from previous builds.

**Using Jenkins and Docker Together**

**Steps:**

1. **Create a Jenkins Job**:
   * In Jenkins, create a new **Pipeline** job.
   * Paste the provided Jenkins pipeline script in the pipeline configuration section.
2. **Set up Docker Hub Credentials**:
   * In Jenkins, go to **Manage Jenkins** > **Manage Credentials** and add Docker Hub credentials (username and password).
3. **Set up GitHub Credentials**:
   * In Jenkins, add GitHub credentials for access to the repository.
4. **Run the Pipeline**:
   * Trigger the pipeline manually or set up a webhook to trigger it upon commits to the main branch.

**Post Actions**

* **Clean Workspace**: After the pipeline run (whether successful or not), the Jenkins workspace is cleaned up using the cleanWs() command. This ensures that the workspace is fresh for the next pipeline run.

**Troubleshooting**

1. **Docker Command Not Found**:
   * Ensure that Docker is installed on the machine where Jenkins is running, and the Jenkins user has permission to run Docker commands.
2. **Git Checkout Issues**:
   * If Jenkins fails to clone the repository, verify that the GitHub credentials and repository URL are correct.
3. **Docker Authentication Failure**:
   * Ensure that Docker Hub credentials are correctly stored in Jenkins, and the credentials ID is properly referenced in the pipeline.

**Conclusion**

By combining Docker and Jenkins, this setup provides a streamlined Continuous Integration (CI) and Continuous Deployment (CD) pipeline. It automates the process of building, tagging, and pushing Docker images for the backend and frontend services while ensuring that metrics are monitored with Prometheus and visualized through Grafana. The Docker Compose configuration allows easy management of all the services within the project, and the Jenkins pipeline ensures consistent delivery of new builds to Docker Hub.

Top of Form

Bottom of Form