DOCKER | PART-2 [Hands-on]

Installing Docker & Docker-compose

Follow these steps to install Docker:

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

```
# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update -y
```

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Follow these steps to install Docker-Compose:

```
sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

```
sudo chmod +x /usr/local/bin/docker-compose
```

Dockerfile Explanation

You have a Dockerfile that uses the eclipse-temurin:17-jdk-alpine base image. Here's what each line does:

```
# Base image with Java 17 (Alpine version)
FROM eclipse-temurin:17-jdk-alpine
```

```
# Expose port 8080 (application will listen on this port)
EXPOSE 8080
```

```
# Set an environment variable for the application home directory
ENV APP_HOME /usr/src/app
```

```
# Copy the JAR file from the host (from the 'target' directory) into the container's app directory
COPY target/*.jar $APP_HOME/app.jar

# Set the working directory inside the container
WORKDIR $APP_HOME

# Command to run the JAR file when the container starts
CMD ["java", "-jar", "app.jar"]
```

Build the Docker Image

1. Save the Dockerfile in the same directory where you have the target directory (which contains the built JAR file from your Java project).

2. Run the following command to build the Docker image:

```
docker build -t <image_name>:<tag> .
```

Replace <image_name> and <tag> with your desired image name and tag.

Example:

```
docker build -t myjavaapp:1.0 .
```

In this command:

- myjavaapp: The name of the Docker image.
- 1.0: The tag version.
- .: Specifies the current directory as the location of the Dockerfile.

Tag the Docker Image

After building the image, you might want to tag it with a repository name for pushing it to a Docker registry (e.g., Docker Hub or a private registry).

1. If you have a Docker Hub account, log in:

```
docker login
```

2. Tag your Docker image for the repository.

```
docker tag <image_name>:<tag> <dockerhub_username>/<repository_name>:<tag>
```

Example:

```
docker tag myjavaapp:1.0 yourdockerhubusername/myjavaapp:1.0
```

This tags the image for the repository <dockerhub_username>/myjavaapp with the version 1.0.

Push the Docker Image

Now, push the tagged Docker image to Docker Hub (or another Docker registry).

```
docker push <dockerhub_username>/<repository_name>:<tag>
```

Example:

```
docker push yourdockerhubusername/myjavaapp:1.0
```

This will upload the image to your Docker Hub account.

Create a Container from the Image

After pushing the image, you can create and run a container from it.

1. Use the docker run command to create a new container from the image and expose the necessary port.

Command:

```
docker run -d -p <host_port>:<container_port> --name <container_name>
<dockerhub_username>/<repository_name>:<tag>
```

Example:

```
docker run -d -p 8080:8080 --name myjavacontainer yourdockerhubusername/myjavaapp:1.0
```

- -d: Runs the container in detached mode (in the background).
- -p 8080:8080: Maps port 8080 on the host to port 8080 in the container.
- --name myjavacontainer: Assigns the name myjavacontainer to the container.

Step 7: Verify the Running Container

1. To check if the container is running:

```
docker ps
```

This will list all running containers and should display your myjavacontainer container.

2. To see the application in action, open a browser and go to:

```
http://VM_IP:8080
```

This assumes your application is running on port 8080 inside the container, and you mapped it to port 8080 on your host.

Step 8: Managing the Container

You can manage the container using the following commands:

- **Stop the container**:

```
docker stop myjavacontainer
```

- **Start the container**:

```
docker start myjavacontainer
```

- **View the container logs**:

```
docker logs myjavacontainer
```

- **Remove the container**:

```
docker rm -f myjavacontainer
```

This will forcefully stop and remove the container.

MongoDB and Mongo Express Project

Step 1: Create a Docker Network

Before running the containers, you need to create a Docker network to allow MongoDB and Mongo Express to communicate with each other.

Command:

```
docker network create my-network
```

This command creates a Docker network named my-network so the containers can communicate with each other by name.

Step 2: Running MongoDB Container

You can now run the MongoDB container and connect it to the network.

Command:

```
docker run -d \
  --name mongodb \
  --network my-network \
  -e MONGO_INITDB_ROOT_USERNAME=admin \
  -e MONGO_INITDB_ROOT_PASSWORD=password \
  -v mongo-data:/data/db \
  mongo:latest
```

Explanation:

- **-d**: Runs the container in detached mode (in the background).
- **--name mongodb**: Names the container mongodb.
- **--network my-network**: Connects the container to the network my-network.
- **-e MONGO_INITDB_ROOT_USERNAME=admin**: Sets the environment variable for MongoDB's root username.
- **-e MONGO_INITDB_ROOT_PASSWORD=password**: Sets the environment variable for MongoDB's root password.
- **-v mongo-data:/data/db**: Creates a named volume mongo-data to persist MongoDB data in the /data/db directory inside the container.
- **mongo:latest**: Uses the latest version of the official MongoDB image from Docker Hub.

Step 3: Running Mongo Express Container

Mongo Express is a web-based MongoDB administration tool. You can run this container and connect it to the same network as MongoDB.

Command:

```
docker run -d \
  --name mongo-express \
  --network my-network \
  -e ME_CONFIG_MONGODB_ADMINUSERNAME=admin \
  -e ME_CONFIG_MONGODB_ADMINPASSWORD=password \
  -e ME_CONFIG_MONGODB_SERVER=mongodb \
  -e ME_CONFIG_BASICAUTH_USERNAME=admin \
  -e ME_CONFIG_BASICAUTH_PASSWORD=pass123 \
  -p 8081:8081 \
  mongo-express:latest
```

Explanation:

- **-d**: Runs the container in detached mode.
- **--name mongo-express**: Names the container mongo-express.
- **--network my-network**: Connects the container to the my-network network (shared with the MongoDB container).
- **-e ME_CONFIG_MONGODB_ADMINUSERNAME=admin**: Sets MongoDB admin username (same as in MongoDB).
- **-e ME_CONFIG_MONGODB_ADMINPASSWORD=password**: Sets MongoDB admin password (same as in MongoDB).
- **-e ME_CONFIG_MONGODB_SERVER=mongodb**: Tells Mongo Express to connect to the MongoDB container by name (mongodb).
- **-e ME_CONFIG_BASICAUTH_USERNAME=admin**: Sets a basic authentication username for Mongo Express.
- **-e ME_CONFIG_BASICAUTH_PASSWORD=pass123**: Sets a basic authentication password for Mongo Express.
- **-p 8081:8081**: Maps port 8081 on the host to port 8081 in the container, allowing you to access Mongo Express via http://localhost:8081.
- **mongo-express:latest**: Uses the latest version of the Mongo Express image from Docker Hub.

Using Docker Compose

Rather than running these containers manually with docker run, you can automate the setup using Docker Compose. Docker Compose allows you to define and manage multiple containers in a single YAML file.

Step 1: Creating a docker-compose.yml File

Create a file named docker-compose.yml in your project directory and add the following content:

```
version: '3.8'

services:
  mongodb:
    image: mongo:latest
    container_name: mongodb
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password
    volumes:
      - mongo-data:/data/db
    networks:
      - my-network
    restart: unless-stopped

  mongo-express:
    image: mongo-express:latest
    container_name: mongo-express
    environment:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
      - ME_CONFIG_MONGODB_ADMINPASSWORD=password
      - ME_CONFIG_MONGODB_SERVER=mongodb
      - ME_CONFIG_BASICAUTH_USERNAME=admin
      - ME_CONFIG_BASICAUTH_PASSWORD=pass123
    ports:
      - "8081:8081"
    networks:
      - my-network
    restart: unless-stopped
    depends_on:
      - mongodb

networks:
  my-network:

volumes:
  mongo-data:
```

Explanation of the docker-compose.yml File:

- **version: '3.8'**: Specifies the version of the Docker Compose file format.
- **services:**: Defines the services (containers) to be run.

    - **mongodb:**: Defines the MongoDB container.
        - **image: mongo:latest**: Uses the latest MongoDB image.
        - **container_name: mongodb**: Names the MongoDB container mongodb.
        - **environment:**: Defines environment variables for MongoDB.
        - **volumes:**: Defines a named volume mongo-data to persist data.
        - **networks:**: Connects the MongoDB container to the my-network network.
        - **restart: unless-stopped**: Restarts the container unless it is manually stopped.
    - **mongo-express:**: Defines the Mongo Express container.
        - **image: mongo-express:latest**: Uses the latest Mongo Express image.
        - **container_name: mongo-express**: Names the Mongo Express container mongo-express.
        - **environment:**: Defines environment variables for Mongo Express.
        - **ports:**: Exposes port 8081 on the host and maps it to port 8081 in the container.
        - **networks:**: Connects the Mongo Express container to the my-network network.
        - **depends_on:**: Ensures MongoDB starts before Mongo Express.
        - **restart: unless-stopped**: Restarts the container unless it is manually stopped.
- **networks:**: Defines the my-network network for communication between services.
- **volumes:**: Defines a named volume mongo-data for MongoDB data persistence.

Step 2: Running Docker Compose

After creating the docker-compose.yml file, you can use Docker Compose to bring up the entire stack (MongoDB and Mongo Express).

Run the following command in the same directory as the docker-compose.yml file:

```
docker-compose up -d
```

Explanation:

- **up**: Builds, (re)creates, and starts the services defined in the docker-compose.yml file.
- **-d**: Runs the containers in detached mode, in the background.

Step 3: Verifying the Setup

1. **Check Running Containers**: Run the following command to see the running containers:

```
docker ps
```

You should see both mongodb and mongo-express containers running.

2. **Access Mongo Express**: Open your browser and go to http://localhost:8081. You should see the Mongo Express web UI.

3. **Stop the Containers**: If you want to stop the containers, use:

```
docker-compose down
```

This will stop and remove the containers, but the data will remain in the mongo-data volume.

Step 4: Managing the Volumes and Networks

With Docker Compose, you don’t have to manually manage networks and volumes as they are defined in the docker-compose.yml file.

- **To list volumes**:

```
docker volume ls
```

This will list the mongo-data volume created by Docker Compose.

- **To list networks**:

```
docker network ls
```

You should see the my-network network created by Docker Compose.

Detailed Steps to Integrate Docker with Jenkins

Integrating Docker with Jenkins allows you to automate the building, testing, and deployment of applications within Docker containers. This integration streamlines your CI/CD pipeline and ensures consistency across development, testing, and production environments.

Below is a comprehensive guide on how to set up Docker with Jenkins, including installing Docker, configuring Jenkins, and setting up a Jenkins pipeline that builds and deploys a Docker image.

Prerequisites

- A server or machine running **Ubuntu** (or a Debian-based Linux distribution).
- **Jenkins** installed and running on the server.
- **Administrative access** to install packages and modify system configurations.

Step 1: Install Docker on the Jenkins Server

First, you need to install Docker Engine on the server where Jenkins is running.

Commands:

1. **Update the package index:**

```
sudo apt-get update
```

2. **Install packages to allow apt to use a repository over HTTPS:**

```
sudo apt-get install -y ca-certificates curl
```

3. **Add Docker's official GPG key:**

```
sudo install -m 0755 -d /etc/apt/keyrings

sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc

sudo chmod a+r /etc/apt/keyrings/docker.asc
```

4. **Add the Docker repository to APT sources:**

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

5. **Update the package index again:**

```
sudo apt-get update
```

6. **Install Docker Engine and related packages:**

```
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

7. **Verify Docker installation:**

```
docker --version
```

You should see the Docker version information, indicating that Docker is installed correctly.

Step 2: Configure Jenkins to Use Docker

To allow Jenkins to execute Docker commands, the Jenkins user must have the appropriate permissions.

Commands:

1. **Add the jenkins user to the docker group:**

```
sudo usermod -aG docker jenkins
```

2. **Apply the new group membership:**

Since Jenkins runs as a service, you need to restart the Jenkins service for the group changes to take effect.

```
sudo systemctl restart jenkins
```

Step 3: Install Necessary Plugins in Jenkins

For Jenkins to interact with Docker and perform other pipeline tasks, you need to install several plugins.

Plugins to Install:

1. **Docker Pipeline Plugin**
   - Allows Jenkins to interact with Docker containers and images.
2. **SonarQube Scanner for Jenkins**
   - Integrates SonarQube code analysis into your Jenkins pipeline.
3. **Nexus Artifact Uploader Plugin**
   - Facilitates uploading artifacts to a Nexus repository.
4. **Pipeline Maven Integration Plugin**
   - Simplifies the use of Maven in your Jenkins pipeline.

Installation Steps:

1. **Access Jenkins Dashboard:**

Navigate to http://your_jenkins_server:8080/ in your web browser.

2. **Install Plugins:**
   - Go to **Manage Jenkins** > **Manage Plugins**.
   - Select the **Available** tab.
   - Search for each plugin by name and check the box next to it.
   - Click **Install without restart**.

Step 4: Configure Global Tool Installations

Ensure Jenkins knows where to find the necessary tools like JDK, Maven, and SonarQube Scanner.

Steps:

1. **Go to Global Tool Configuration:**
   - Navigate to **Manage Jenkins** > **Global Tool Configuration**.
2. **Add JDK Installation:**
   - Under **JDK**, click **Add JDK**.
   - **Name:** jdk17
   - **Install automatically:**
     - Select a version from eclipse adoptium installer
3. **Add Maven Installation:**
   - Under **Maven**, click **Add Maven**.
   - **Name:** maven3
   - **Install automatically:** Choose to install a specific version.
4. **Add SonarQube Scanner:**
   - Under **SonarQube Scanner**, click **Add SonarQube Scanner**.
   - **Name:** sonar-scanner
   - **Install automatically:** Configure as needed.
5. **Configure SonarQube Servers:**
   - Go to **Manage Jenkins** > **Configure System**.

    - Find **SonarQube servers** section.
    - Click **Add SonarQube**.
        - **Name:** sonar (must match the name in the pipeline).
        - **Server URL:** Your SonarQube server URL.
        - **Authentication Token:** Add credentials for SonarQube access.

Step 5: Configure Credentials in Jenkins

You'll need to add credentials for Docker registry, Nexus, and potentially SonarQube.

Steps:

1. **Access Credentials:**
    - Go to **Manage Jenkins** > **Credentials** > **System** > **Global credentials (unrestricted)**.
2. **Add Docker Registry Credentials:**
    - Click **Add Credentials**.
    - **Kind:** Username with password.
    - **Username:** Your Docker registry username.
    - **Password:** Your Docker registry password or token.
    - **ID:** docker-cred (must match the ID used in the pipeline).
    - **Description:** Docker Registry Credentials.
3. **Add Nexus Credentials:**
    - **Kind:** Username with password.
    - **Username:** Nexus username.
    - **Password:** Nexus password.
    - **ID:** nx (must match the ID used in the pipeline).
    - **Description:** Nexus Credentials.
4. **Add SonarQube Token (if not using global configuration):**
    - **Kind:** Secret text.
    - **Secret:** Your SonarQube token.
    - **ID:** Use an appropriate ID.

Step 6: Prepare the Jenkins Pipeline Script

Use the provided Jenkinsfile, which defines the stages for your CI/CD pipeline.

Jenkinsfile Overview:

```
pipeline {
    agent any

    tools{
        jdk 'jdk17'
        maven 'maven3'
    }

    environment {
        SONARQUBE_HOME= tool 'sonar-scanner'
    }

    stages {
        stage('Git CheckOut') {
            steps {
                git 'https://github.com/jaiswaladi2468/BoardgameListingWebApp.git'
            }
        }

        stage('Compile') {
            steps {
                sh "mvn compile"
            }
        }

        stage('Unit Tests') {
            steps {
                sh "mvn test"
            }
        }


        stage('Package') {
            steps {
                sh "mvn package"
            }
        }

        stage('SonarQube Analysis') {
            steps {
                withSonarQubeEnv('sonar') {
                    sh ''' $SONARQUBE_HOME/bin/sonar-scanner -Dsonar.projectName=Boardgame -
Dsonar.projectKey=Boardgame \
                          -Dsonar.java.binaries=. '''

                }
            }
        }


      stage('Quality Gate') {
            steps {
                waitForQualityGate abortPipeline: false
            }
        }


        stage('Deploy Artifacts ') {
            steps {
                withMaven(globalMavenSettingsConfig: 'global-maven-settings', jdk: 'jdk17', maven:
'maven3', mavenSettingsConfig: '', traceability: false) {
                     sh "mvn deploy"
                }
            }
        }

        stage('Docker Build Image') {
            steps {
                script {
                withDockerRegistry(credentialsId: 'docker-cred', toolName: 'docker') {

                    sh "docker build -t boardwebapp:latest ."
                    sh "docker tag boardwebapp:latest adijaiswal/boardwebapp:latest"

                }

                }
            }
        }

         stage('trivy Image scan') {
            steps {
                sh " trivy image adijaiswal/boardwebapp:latest "
            }
        }


        stage('Docker Push Image') {
            steps {
                script {
                withDockerRegistry(credentialsId: 'docker-cred', toolName: 'docker') {

                    sh "docker push adijaiswal/boardwebapp:latest"

                }

                }
            }
        }

        stage('Deploy application to container') {
            steps {
                script {
                withDockerRegistry(credentialsId: 'docker-cred', toolName: 'docker') {

                    sh "docker run -d -p 8085:8080 adijaiswal/boardwebapp:latest"


                }

                }
            }
        }

    }
}
```

- **Stages:**
    1. **Git CheckOut:** Clones the Git repository.
    2. **Compile:** Compiles the Java project.
    3. **Unit Tests:** Runs unit tests.
    4. **Package:** Packages the application.
    5. **SonarQube Analysis:** Performs static code analysis.
    6. **Quality Gate:** Checks the quality gate status.
    7. **Deploy Artifacts To Nexus:** Uploads artifacts to Nexus.
    8. **Deploy Artifacts:** Deploys artifacts using Maven.
    9. **Docker Build Image:** Builds a Docker image.
    10. **Trivy Image Scan:** Scans the Docker image for vulnerabilities.
    11. **Docker Push Image:** Pushes the image to a Docker registry.
    12. **Deploy Application to Container:** Runs the Docker container.

Create the Pipeline Job:

1. **In Jenkins Dashboard:**
    - Click **New Item**.
    - Enter **Job Name** (e.g., BoardgameListingPipeline).

- Select **Pipeline**.
- Click **OK**.

2. **Configure the Pipeline:**
   - Under **Pipeline** section, choose **Pipeline script**.
   - Paste the Jenkinsfile content.
   - Click **Save**.

Step 7: Install Trivy on Jenkins Server

Trivy is used for scanning Docker images for vulnerabilities.

Installation Commands:

```
# Install required packages
sudo apt-get install -y wget apt-transport-https gnupg lsb-release

# Add Trivy's official GPG key and repository
wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | sudo apt-key add -
echo deb https://aquasecurity.github.io/trivy-repo/deb $(lsb_release -sc) main | sudo tee -a
/etc/apt/sources.list.d/trivy.list

# Update package index and install Trivy
sudo apt-get update
sudo apt-get install -y trivy
```

Verify Installation:

```
trivy --version
```

You should see the Trivy version information.

Step 8: Ensure Docker Can Run Without Sudo

For Jenkins to execute Docker commands without sudo, the jenkins user must be in the docker group, which we did earlier. Confirm that no permissions issues are present.

Step 9: Run the Pipeline

1. **Start the Build:**
   - In Jenkins, navigate to your pipeline job.

- Click **Build Now**.

2. **Monitor the Build:**
   - Click on the build number to view the build details.
   - Use **Console Output** to monitor the progress and check for errors.

Step 10: Verify Each Pipeline Stage

Stage Explanations and Verifications:

1. **Git CheckOut:**
   - **Action:** Clones the repository from GitHub.
   - **Verification:** Ensure the repository is cloned successfully.
2. **Compile:**
   - **Action:** Compiles the project using Maven.
   - **Verification:** Check for compilation success messages.
3. **Unit Tests:**
   - **Action:** Runs unit tests.
   - **Verification:** Ensure tests pass without failures.
4. **Package:**
   - **Action:** Packages the application into a JAR file.
   - **Verification:** Confirm that the JAR file is created in the target directory.
5. **SonarQube Analysis:**
   - **Action:** Performs static code analysis.
   - **Verification:** Check SonarQube dashboard for analysis results.
6. **Quality Gate:**
   - **Action:** Waits for the Quality Gate status from SonarQube.
   - **Verification:** Ensure the Quality Gate is passed.
7. **Deploy Artifacts To Nexus:**
   - **Action:** Uploads artifacts to Nexus repository.
   - **Verification:** Verify that the artifacts are present in Nexus.
8. **Deploy Artifacts:**
   - **Action:** Uses Maven to deploy artifacts.

    - **Verification:** Confirm deployment success messages.

9. **Docker Build Image:**
    - **Action:** Builds the Docker image using the Dockerfile.
    - **Verification:** Check for successful image build messages.
10. **Trivy Image Scan:**
    - **Action:** Scans the Docker image for vulnerabilities.
    - **Verification:** Review the scan output for any vulnerabilities.
11. **Docker Push Image:**
    - **Action:** Pushes the Docker image to the Docker registry.
    - **Verification:** Ensure the image is available in your Docker registry.
12. **Deploy Application to Container:**
    - **Action:** Runs the Docker container from the image.
    - **Verification:** Access the application via http://your_server_ip:8085/ to confirm it's running.

Step 11: Troubleshooting Common Issues

- **Permission Denied Errors:**
    - Ensure the jenkins user is in the docker group.
    - Restart the Jenkins service after modifying group memberships.
- **Tool Not Found:**
    - Verify that all tools (JDK, Maven, SonarQube Scanner, Trivy) are correctly installed and configured in Jenkins.
- **Credentials Issues:**
    - Double-check that all credentials IDs in the Jenkinsfile match those configured in Jenkins.
- **Network Connectivity:**
    - Ensure the Jenkins server has internet access to download dependencies and push/pull images.
- **Plugin Compatibility:**
    - Make sure all plugins are up to date and compatible with your Jenkins version.

Step 12: Security Considerations

- **Credentials Security:**
  - Use Jenkins Credentials Manager to securely store sensitive information.
  - Limit access to Jenkins configurations and credentials.
- **Docker Socket Exposure:**
  - Be cautious with Docker permissions to prevent unauthorized access.
  - Consider using tools like **Docker-in-Docker** or **Docker agents** for enhanced security.

Step 13: Clean Up and Maintenance

- **Regular Updates:**
  - Keep Jenkins, plugins, and tools updated to the latest versions.
- **Monitor Pipeline Runs:**
  - Regularly check pipeline runs for failures and address issues promptly.
- **Resource Management:**
  - Monitor system resources to ensure the Jenkins server operates efficiently.

Conclusion

By integrating Docker with Jenkins, you enhance your CI/CD pipeline, enabling automated builds, tests, and deployments within Docker containers. This setup ensures consistency across environments and accelerates the delivery process.