CI/CD Pipeline and Deployment Process with Jenkins

1. Setting up Jenkins

To begin with, I installed and configured Jenkins on a dedicated server. Jenkins is an essential tool in the CI/CD pipeline, automating the entire process from code build to deployment. After successfully installing Jenkins, I accessed the web interface to complete the initial setup.

I configured Jenkins to work with both JDK and Maven:

- **JDK**: Version 21, ensuring compatibility with the Java version used in the application.
- **Maven**: Version 3.9.9, allowing Jenkins to handle the build and testing phases for the Java application.

2. Creating a Jenkins Job

After setting up Jenkins, I created a new Jenkins job specifically for this pipeline. I configured the job to fetch the source code from the `main` branch of the GitHub repository: `https://github.com/ojAsare910/songapp.git`.

By doing this, Jenkins is always in sync with the latest code updates, ensuring that any new commits to the `main` branch trigger the pipeline process.

3. Building the Java Application

In the pipeline, the first significant stage is building the Java application. Using Maven, I integrated the `mvn clean package` command into the Jenkins pipeline. This command compiles the source code and packages it into a deployable JAR file.

If there are any issues during the build (e.g., compilation errors), the pipeline will halt, preventing any further stages from executing, which is critical in maintaining code integrity.

4. Running Unit Tests

Once the build is successful, the next step involves running unit tests. This is handled using the Maven command `mvn test`, which executes the predefined test cases in the project.

Running tests early in the pipeline ensures that no broken code gets deployed. Should any test fail, the pipeline immediately stops, and the build is marked as unsuccessful. This step is essential to maintaining the quality and stability of the application.

5. Building the Docker Image

After ensuring that the application has been built and tested successfully, I move to the next stage: containerizing the application. This involves creating a Docker image of the application.

The Jenkins pipeline uses the `docker.build("${DOCKER_IMAGE}")` command, where `${DOCKER_IMAGE}` is the identifier for the Docker image, in this case: `ojasare910/songapp:latest`. This command builds the Docker image, packaging the application along with its dependencies and configurations inside a container. The resulting image can be deployed consistently across multiple environments, ensuring that there are no environment-specific issues.

6. Pushing the Docker Image to Docker Hub

This stage of the pipeline is critical in making the Docker image accessible across various environments and teams. Once the Docker image is successfully built, it must be stored in a registry to be pulled later for deployment. For this pipeline, I chose Docker Hub as the registry.

Detailed Breakdown of Docker Push Stage in Jenkins Pipeline:

In the Jenkins pipeline, I configured the following stage to push the Docker image to Docker Hub:

```
stage('Push Docker Image') {
    steps {
        withCredentials([string(credentialsId: 'docker-hub', variable:
'dockerhubpwd')]) {
            sh 'docker login -u ojasare910 -p ${dockerhubpwd}'
        }
        sh 'docker push ${DOCKER_IMAGE}'
    }
}
```

Here's how this process works step-by-step:

Authentication with Docker Hub: The first task is logging into Docker Hub. To ensure security, I used Jenkins' credential management system. This allows me to securely store sensitive information such as my Docker Hub password. In the pipeline, I retrieve this password using the `withCredentials` block:

```
withCredentials([string(credentialsId: 'docker-hub', variable: 'dockerhubpwd')])
```

The `credentialsId` (`docker-hub`) refers to the secret I previously configured in Jenkins for Docker Hub. This secret includes my Docker Hub credentials. During the build, Jenkins retrieves the password and assigns it to the environment variable `dockerhubpwd`.
After that, the following shell command logs me into Docker Hub:

```
docker login -u ojasare910 -p ${dockerhubpwd}
```

1.
    - `ojasare910`: This is my Docker Hub username.
    - `${dockerhubpwd}`: The retrieved password.

2. This command logs Jenkins into Docker Hub so it can push the Docker image. By handling credentials in this way, I prevent sensitive information from being exposed in the pipeline logs.

Pushing the Docker Image: Once logged in, the pipeline executes the command:

```
docker push ${DOCKER_IMAGE}
```

3. `${DOCKER_IMAGE}` is the image tag, which in this case is set to `ojasare910/songapp:latest`. The `docker push` command uploads the image to Docker Hub under my account (`ojasare910`). The image is tagged as `latest`, meaning it represents the most up-to-date version of the application.
   After this stage is completed, the Docker image is publicly available on Docker Hub. From this point on, anyone (or any deployment pipeline) can pull the image from Docker Hub to deploy the application. This approach centralises the storage of Docker images and enables seamless access across different deployment environments.

7. Configuring the Deployment Environment

Before deploying the application, the target environment must be prepared. In this case, the target environment is a virtual machine where the application will run. The VM is set up with the necessary tools and services, such as Docker, Postgres, and Redis.

In addition, I made sure that the correct network configurations, like opening port `8083`, were in place. This ensures that the application, database, and Redis cache can communicate with each other and external clients without any issues.

8. Deploying to the Target Environment

Finally, I configured Jenkins to deploy the Docker image to the virtual machine. This is achieved by running the Docker container on the target machine. The following stage in the Jenkins pipeline handles the deployment:

```
stage('Deploy') {
    steps {
        sh '''
            docker run -d \
            -p ${SERVER_PORT}:${SERVER_PORT} \
            --name songapp-api \
            -e SPRING_DATASOURCE_URL=${POSTGRES_URL} \
            -e SPRING_DATASOURCE_USERNAME=${POSTGRES_USER} \
            -e SPRING_DATASOURCE_PASSWORD=${POSTGRES_PASSWORD} \
            -e SPRING_DATA_REDIS_HOST=${REDIS_HOST} \
            -e SPRING_DATA_REDIS_PORT=${REDIS_PORT} \
            -e SERVER_PORT=${SERVER_PORT} \
            -v /path/on/host/logs:/app/logs \
            ${DOCKER_IMAGE}
```

```
        '''
    }
}
```

This script pulls the latest Docker image from Docker Hub and runs it on the target VM with the correct environment variables. The environment variables, such as `SPRING_DATASOURCE_URL`, `SPRING_DATASOURCE_USERNAME`, and others, are used to configure the connection to Postgres and Redis at runtime.

The command `docker run -d` ensures the application runs in detached mode, meaning it runs in the background. The container is also mapped to port `8083`, making the application accessible on this port.

Takeaway Notes

1. **Docker Image Push to Docker Hub**: This is a crucial step in the pipeline. By pushing the image to Docker Hub, I ensure that my application's container is centralized and easily accessible across different environments.
2. **Security in CI/CD**: I learned the importance of managing credentials securely using Jenkins' credential management system to avoid leaking sensitive information.
3. **Automation Efficiency**: Automating the entire process, from build to deployment, allows for quicker releases with minimal manual intervention.
4. **Environment Preparation**: A well-prepared deployment environment with all necessary services (Postgres, Redis) and configurations ensures smooth deployment.
5. **Containerization Benefits**: Docker ensures that my application can be deployed consistently across any environment, reducing the chances of environment-specific issues.

This entire CI/CD process ensures that every change in the codebase is automatically built, tested, containerized, and deployed with minimal manual effort, increasing overall productivity and reliability.