CLOUDBEES UNIVERSITY

CloudBees Jenkins Platform: Pipeline with Docker

Labs

Table of Contents

Introduction

This workbook is designed to supplement your CloudBees Jenkins training. It consists of a sequence of lab exercises that will introduce Jenkins Pipeline and Docker concepts and best practices.

The Project - Part 1: Exercise

The solution to all tasks is located at the end. Please try to solve them by yourself and look at the solutions only if you get stuck or want to validate your work.

The exercise consists of performing the whole delivery lifecycle of an application using Jenkins Pipeline and Docker. We'll test the application, and build and push a Docker container. The service we'll use is written in Go. Since this training is language-agnostic, you are not expected to know the language. You will not need any additional tools. All the tasks should be completed with Docker.

Before diving into the exercise, please SSH into the machine you are assigned and enter the */mnt/training-books-ms/exercises* directory. All the code you'll need is inside.

Task: Create a new Pipeline job

The requirements for this task are as follows.

- Create a new **Pipeline** job called **docker-flow-proxy**.

For additional information, please consult Getting Started tutorial.

Task: Clone the code from the GitHub repository

The requirements for this task are as follows.

- Everything should run inside the node called **cd**.
- Clone of **pipeline** branch of the **https://github.com/cloudbees/training-books-ms** repository.

For additional information, please consult node and git documentation.

Task: Run the tests and build the binary

The requirements for this task are as follows.

- Everything should run inside the node called **cd**.

- Define the stage called **test**.
- Run the tests and build the binary inside the **golang** container.
- Set user and group to **0**.
- Run the following commands inside the container.

```
ln -s $PWD /go/src/docker-flow
cd /go/src/docker-flow && go get -t && go test --cover -v
cd /go/src/docker-flow && go build -v -o docker-flow-proxy
```

For additional information, please consult withDockerContainer and stage documentation.

Task: Build the image and push it to the local registry

The requirements for this task are as follows.

- Everything should run inside the node called **cd**.
- Define the stage called **build**.
- Build the container called **localhost:5000/docker-flow-proxy**.
- Build the container called **localhost:5000/docker-flow-proxy**.

Task: Archive the binary

The requirements for this task are as follows.

- Everything should run inside the node called **cd**.
- Archive the binary **docker-flow-proxy**.

For additional information, please consult archive documentation.

Task: Run latest version of the container

The requirements for this task are as follows.

- Define the checkpoint called **deploy**
- Everything should run inside the node called **production**.
- Define the stage called **deploy**.
- Delete the container called **docker-flow-proxy**. Make sure that the pipeline does not fail if the

container is not running.

- Run the **localhost:5000/docker-flow-proxy** container.
- The name of the container should be **docker-flow-proxy**.
- The internal port **80** should be exposed to the host as **8081**.
- The internal port **8080** should be exposed to the host as **8082**.

Solution: Create a new Pipeline job

- Open the Jenkins UI.
- Click the **New Item** link from the left-hand menu.
- Type **docker-flow-proxy** in the **Item Name** field, select the **Pipeline** job type, and click the **OK** button.

Solution: Clone the code from the GitHub repository

Write the following script inside the **Pipeline Script** field.

```
node("cd") {
    git branch: 'pipeline', url: 'https://github.com/cloudbees/training-books-ms'
}
```

The explanation of the snippet is as follows.

- The steps executed inside the `node` block will run in one of the slaves named or labeled **cd**.

Solution: Run the tests and build the binary

Add the following snippet below the `git` instruction inside the **Pipeline Script** field.

```
    stage 'test'
    docker.image("golang").inside('-u 0:0') {
        sh 'ln -s $PWD /go/src/docker-flow'
        sh 'cd /go/src/docker-flow && go get -t && go test --cover -v'
        sh 'cd /go/src/docker-flow && go build -v -o docker-flow-proxy'
    }
```

The explanation of the snippet is as follows.

- The `stage` instruction defines a group of steps.
- The `docker.image` instruction specifies the name of the image that will be used.

- The arguments set for the `inside` function will be passed as Docker `run` arguments.
- The steps executed inside the `docker.image.inside` block will run inside the specified container.

Solution: Build the image and push it to the local registry

Add the following snippet below the `docker.image("golang").inside('-u 0:0')` block inside the **Pipeline Script** field.

```
    stage 'build'
    docker.build('localhost:5000/docker-flow-proxy')
    docker.image('localhost:5000/docker-flow-proxy').push()
```

The explanation of the snippet is as follows.

- The `stage` instruction defines a group of steps.
- The `docker.build` instruction builds the image called `localhost:5000/docker-flow-proxy`. The name itself follows the **<REGISTRY>/<IMAGE_NAME>** format.
- The `docker.image` instruction returns **Container**. The `push` command is given to the container **localhost:5000/docker-flow-proxy**.

Solution: Archive the binary

Add the following snippet below the `docker.image('localhost:5000/docker-flow-proxy').push()` instruction inside the **Pipeline Script** field.

```
    archive 'docker-flow-proxy'
```

The explanation of the snippet is as follows.

- The `archive` instruction stores all the files that match the specified pattern.

Solution: Run latest version of the container

```
checkpoint 'deploy'

node('production') {
    stage 'deploy'
    try {
        sh 'docker rm -f docker-flow-proxy'
    } catch(e) { }
    docker.image('localhost:5000/docker-flow-proxy').run('--name docker-flow-
proxy -p 8081:80 -p 8082:8080')
}
```

The explanation of the snippet is as follows.

- In case of a system failure, the `checkpoint` instruction allows restart of the job from that point.
- The steps executed inside the `node` block will run in one of the slaves named or labeled **production**.
- The `sh` step is used to run the Docker `rm` command that removes the container. To prevent possible failure in case such a container does not exist, the step is wrapped inside an `try/catch` statement.

Solution: The complete Pipeline script

The complete script is listed below.

```
node("cd") {
    git branch: 'pipeline', url: 'https://github.com/cloudbees/training-books-ms'

    stage 'test'
    docker.image("golang").inside('-u 0:0') {
        sh 'ln -s $PWD /go/src/docker-flow'
        sh 'cd /go/src/docker-flow && go get -t && go test --cover -v'
        sh 'cd /go/src/docker-flow && go build -v -o docker-flow-proxy'
    }

    stage 'build'
    docker.build('localhost:5000/docker-flow-proxy')
    docker.image('localhost:5000/docker-flow-proxy').push()
    archive 'docker-flow-proxy'
}

checkpoint 'deploy'

node('production') {
    stage 'deploy'
    try {
        sh 'docker rm -f docker-flow-proxy'
    } catch(e) { }
    docker.image('localhost:5000/docker-flow-proxy').run('--name docker-flow-
proxy -p 8081:80 -p 8082:8080')
}
```

Go back to slides