Docker CI/CD With Jenkins

Objective: This chapter explains how to use Jenkins and Docker to run continuous integration and continuous delivery<br>
Preparation: Open up two instances of your favorite shell, and a browser window. Make absolute sure you are in the appropriate lab folder. Confimr you are in the Part B subfolder.<br>
Outcome: Each student will replicate a highlight demo of some basic functionality<br>
Data Files: Ask Instructor<br>

There are several possible approaches to run Docker builds with Jenkins:

- Install Jenkins on your host machine, where Docker is also installed, and run Docker commands from your build, either using one of the several Jenkins Docker plugins, or by running Docker commands from a build step
- Install Jenkins on your host machine and have a Jenkins slave machine with Docker installed to run your Docker builds
- Run Jenkins on Docker and use the underlying Docker installed on the host to run Docker commands.

> *NOTE: Another option is running Jenkins on Docker and do a complete Docker installation inside the Jenkins Docker container. This technique is called Docker in Docker and it is usually a bad idea. There are several discussions about the problems with this*

> *approach, like this one: http://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/ . A better approach is using Docker outside of Docker, as explained here: http://container-solutions.com/running-docker-in-jenkins-in-docker/*

Step 1. Run Jenkins on Docker

In this example, we will run Jenkins on Docker and use the underlying Docker installed on the host to run Docker commands. This technique is known as Docker outside of Docker.

1. First, clone the project at `https://github.com/fabianenardon/jenkins-docker-demo`, however you should already have it.

2. Then, in the project folder, run:

```
docker-compose up
```

3. Wait for jenkins to start and then go to the browser and open `http://localhost:8081`. Jenkins should be running.

The Jenkins installation on this lab comes pre-configured. To login use the username `jenkins` and the password `jenkins`.

Step 2. Running Integration Tests with Docker and Jenkins

For this Continuous Integration demo, we will run a simple application that saves data on MongoDB. We will then run integration tests to check if the data was correctly saved on the database.

When running integration tests, you want to test your application in an environment as close to production as possible, so you can test interactions between the several components, services, databases, network communication, etc. Fortunately, docker can help you a lot with integration tests. There are several strategies to run integration tests, but in this application we are going to use the following:

- Start the services with a `docker-compose.yml` file created for testing purposes. This file won’t have any volumes mapped, so when the test is over, no state will be saved. The test `docker-compose.yml` file won’t publish any port on the host machine, so we can run simultaneous tests.
- Run the application, using the services started with the `docker-compose.yml` test file.
- Run Maven integration tests to check if the application execution produced the expected results. This will be done by checking what was saved on the MongoDB database.
- Stop the services. No state will be stored, so next time you run the integration tests, you will have a clean environment.

Create a new job on jenkins:

1. Select Freestyle project

image::docker-ci-cd-01.png[]

2. In Source Code Management, select Git and add the repository URL: `https://github.com/fabianenardon/mongo-docker-demo.git`

image::docker-ci-cd-02.png[]

3. In Build, select Add build step and select Execute shell

image::docker-ci-cd-03.png[]

4. In the shell Command, add these instructions:

```
$ cd sample

# Generates the images
$ sudo /var/jenkins_home/tools/hudson.tasks.Maven_MavenIn
stallation/maven/bin/mvn clean install -Papp-docker-image

# Starts the mongo service. The -p option allows multiple
 builds to run at the same time,
# since we can start multiple instances of the containers
$ sudo docker-compose -p app-$BUILD_NUMBER --file src/tes
t/resources/docker-compose.yml up -d mongo

# Waits for containers to start
sleep 30

# Run the application
$ sudo docker-compose -p app-$BUILD_NUMBER --file src/tes
```

```
t/resources/docker-compose.yml \
     run mongo-docker-demo \
     java -jar /maven/jar/mongo-docker-demo-1.0-SNAPSHOT-
jar-with-dependencies.jar mongo

# Run the integration tests
$ sudo docker-compose -p app-$BUILD_NUMBER --file src/tes
t/resources/docker-compose.yml \
     run mongo-docker-demo-tests \
     mvn -f /maven/code/pom.xml -Dmaven.repo.local=/m2/re
pository \
     -Pintegration-test verify
```

5 . Click on Add post-build action and select Execute a set of scripts

image::docker-ci-cd-04.png[]

6 . In Post-build Actions, select Execute shell

image::docker-ci-cd-05.png[]

7 . In the Command box, add:

```
$ cd sample
$ sudo docker-compose -p app-$BUILD_NUMBER --file src/tes
t/resources/docker-compose.yml down
```

8 . Uncheck the `Execute script only if build succeeds` and `Execute script only if build fails` options, so this script will run always when the build ends. This way, we make sure to always stop the services.

> *NOTE:*

- The `-p app-$BUILD_NUMBER` option allows multiple builds to run at the same time, since we can start multiple instances of the containers. We are using Jenkins $BUILD_NUMBER variable to isolate the containers. This way, each set of services will run on its own network.
- We are running the commands with sudo because we are actually running the Docker socket on the host. Jenkins runs with the jenkins user and we added the jenkins user to the sudoers list in our image. Obviously, this can have security consequences in a production environment, since one could create a build that would access root level services on the host. You can avoid this by configuring the jenkins user on the host, so it will have access to the Docker socket and then run the commands without sudo.

9 . Save the build and then click on `Build now` to run it. You should see a progress bar when the build is running. You can click on the progress bar to see the build console output.

image::docker-ci-cd-06.png[]

10 . If the build is successful, you should see this on the build console output:

image::docker-ci-cd-07.png[]

Running and Debugging Integration Tests Outside Jenkins

1. When creating integration tests, it is useful to be able to run and debug them outside Jenkins. In order to do that, you can simply run the same commands you ran in the Jenkins build:

```
# Generates the images
mvn clean install -Papp-docker-image


# Starts mongo service
docker-compose --file src/test/resources/docker-compose.y
ml up -d mongo


# Waits for services do start
sleep 30


# Run our application
docker-compose --file src/test/resources/docker-compose.y
ml \
               run mongo-docker-demo \
               java -jar /maven/jar/mongo-docker-demo-1.0
-SNAPSHOT-jar-with-dependencies.jar mongo
```

```

# Run our integration tests
docker-compose --file src/test/resources/docker-compose.y
ml \
               run mongo-docker-demo-tests mvn -f /maven/
code/pom.xml \
               -Dmaven.repo.local=/m2/repository -Pintegr
ation-test verify

# Stop all the services
docker-compose --file src/test/resources/docker-compose.y
ml down
```

2. If you wanted to debug your integration tests, you would run the tests with this command:

```
# Run integration tests in debug mode
docker run -v ~/.m2/repository:/m2/repository \
       -p 5005:5005 --link mongo:mongo \
       --net resources_default mongo-docker-demo-tests \
       mvn -f /maven/code/pom.xml \
       -Dmaven.repo.local=/m2/repository \
       -Pintegration-test verify -Dmaven.failsafe.debug
```

3. This will make your test wait for a connection on port 5005 for debugging. You can then attach your IDE to this port and debug. Here is how this is done on Netbeans:

image::docker-ci-cd-08.png[]

image::docker-ci-cd-09.png[]

Conclusion:

Continuous Delivery strategies depend greatly on the application architecture. With a dockerized application like the one in our demo, the continuous delivery strategy could be to publish a new version of the application image if the tests passed. This way, next time the application runs on production, the new image will be downloaded and automatically deployed. You can publish images with Jenkins just like you invoked all the other docker commands in the build.