Pre-requisite

* A github repo with Java project with a Dockerfile and Dockerrun.aws.json
* A github repo with node.js project a Dockerfile and Dockerrun.aws.json
* A github repo with react.js project a Dockerfile and Dockerrun.aws.json
* Aws credential
* Docker Hub account
* An account on AWS Elastic Beanstalk

Theory:

We usually deploy our code to these environments quite often and sometimes the instances would stop working right in the middle of a deployment. This causes a bit of irritation because it means we have to wait 10 - 15 minutes for it to fail, reboot the instance and try it again. It is a waste of time and it could mean a downtime of 20 minutes or more for the application development process to continue. The reason our instances get overloaded is due to all of the work that it does when a new version is being deployed. That's when Docker Hub can be of use.

Docker Hub: Docker Hub gives us a space to host and manage our Docker containers. This way we can delegate all the hard and tedious work to it.The main purpose of doing this is to separate the container management from the deployment service. Once Docker Hub informs us that our new image has been built correctly, we’re ready to deliver it to whichever service we prefer. This has a huge benefit over traditional methods because separating the image building process from the actual instance means that our application environment will keep running until we do the actual deployment. Docker Hub also comes with many handy features to make our work easier.

**Autobuild**: Docker Hub automatically pulls our code from GitHub or BitBucket, locates the Dockerfile in it and starts building, tagging and pushing the image into the container. (Solution for Assignment Part 1)

**Autotest**: Docker Hub starts running our application tests after building the image and allows us to automatically stop the push process if anything fails. (Solution for Assignment Part 2)

Below is the implementation of this concept.

1. Add a Dockerfile to the project. Docker will use this file to know how to build your image.

* for java sample java docker file will be like below:

# Multi-stage build setup (<https://docs.docker.com/develop/develop-images/multistage-build>/)

# Stage 1 (to create a "build" image, ~140MB)

FROM openjdk:8-jdk-alpine3.7 AS builder

RUN java -version

COPY . /usr/src/easy-shop/

WORKDIR /usr/src/easy-shop/

RUN apk --no-cache add maven && mvn --version

RUN mvn package

# Stage 2 (to create a downsized "container executable", ~87MB)

FROM openjdk:8-jre-alpine3.7

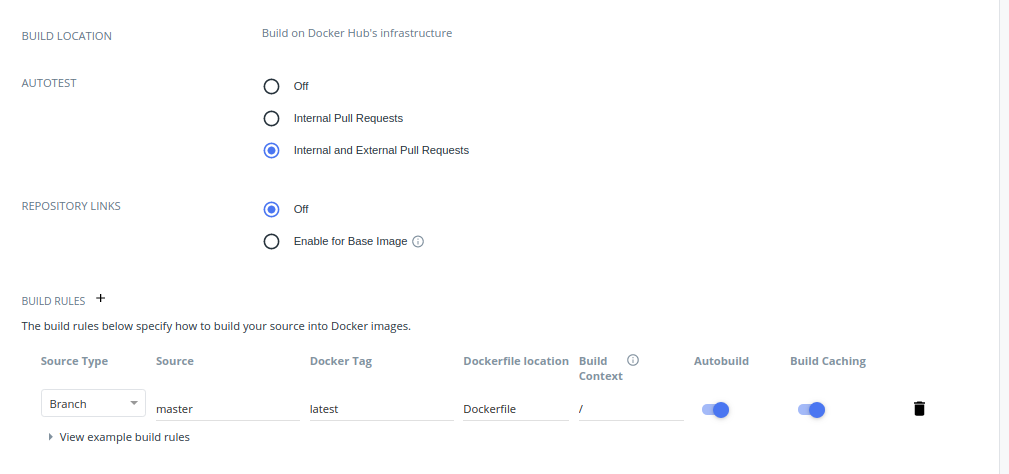
WORKDIR /root/

COPY --from=builder /usr/src/myapp/target/app.jar .

EXPOSE 8123

ENTRYPOINT ["java", "-jar", "./app.jar"]

2. Login into docker hub and configure it for autobuild of docker image with every checkin to master branch and also enable autotest for any internal or external pull requests hence with every checkin an image will be created. This will create a file located at `.docker/config.json` that we will need later



3. Add a `Dockerrun.aws.json` file to the project.

{

"AWSEBDockerrunVersion": "1",

"Ports": [

{

"ContainerPort": "80"

}

],

"Authentication": {

// S3 Bucket where the config.json is stored

"Bucket": "MY-DOCKERCONFIG-S3-BUCKET",

// config.json file created by `docker login` command

"Key": "config.json"

},

"Image": {

// Name of my Docker Hub user and Docker Hub repository

"Name": "MYUSER/MYREPO",

"Update": "true"

},

"Volumes": [],

"Logging": "/var/log/nginx"

}

4.login to AWS web console and create an Elastic Beanstalk application. Next, create an environment selecting Docker as platform and choose the “upload your code” option.

5. Now create a private bucket on AWS S3 to store your Docker credentials. Upload the `config.json` (created in step 2) file which contains docker credentials like below.

{

"auths": {

"https://index.docker.io/v1/": {

"auth": "YWJoaW5hdmRvY2tlcjQ3OmxhanBhdHZpaGFy"

}

},

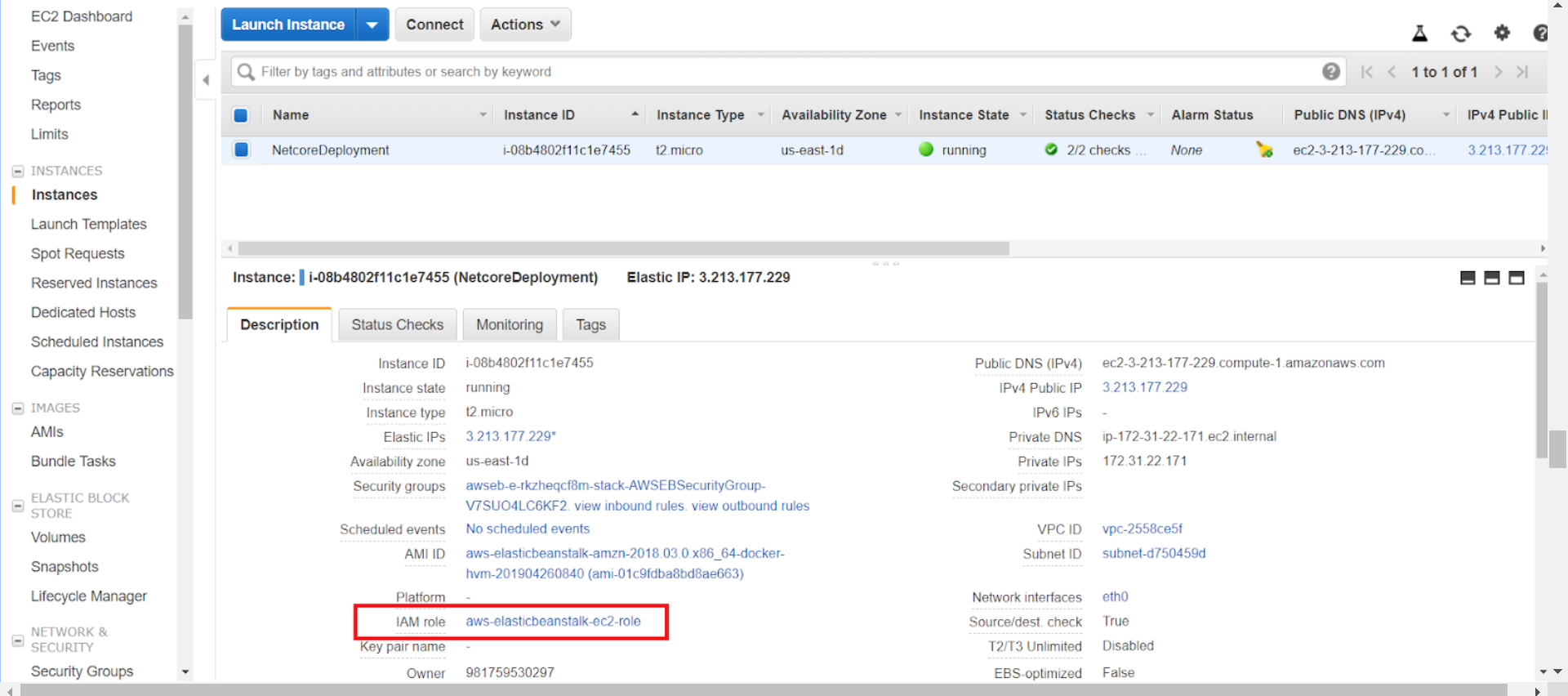
"HttpHeaders": {

"User-Agent": "Docker-Client/18.09.7 (linux)"

}

}

6. Go to EC2 and search for the recently created instance. Once there, update its IAM credentials and attach theAmazonS3ReadOnlyAccess policy. It should be set like as shown below



7. Initialize the AWS EB CLI using the `eb init` command, follow the steps and once it’s done, go to the `.elasticbeanstalk` folder and [configure](https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3-configuration.html#_blank) the `config.yml` file to only upload the Dockerrun.aws.json file. File should look like below

branch-defaults:

default:

environment: ENVIRONMENT\_NAME

deploy:

artifact: Dockerrun.aws.json

environment-defaults:

ENVIRONMENT\_NAME:

branch: null

repository: null

global:

application\_name: APPLICATION\_NAME

default\_ec2\_keyname: EC2\_KEYNAME

default\_platform: arn:aws:elasticbeanstalk:us-east-1::platform/Docker running on

64bit Amazon Linux/2.12.6

default\_region: us-east-1

include\_git\_submodules: true

instance\_profile: null

platform\_name: null

platform\_version: null

profile: eb-cli

sc: null

workspace\_type: Application

8. Push all the code to a branch that meets the conditions set in Docker Hub ‘autobuild’ (master branch)

9. Hold on until Docker Hub finishes building the image, once the status is “Success” we ready to deploy it on Elastic Beanstalk

10. Execute `eb deploy` command in terminal to automatically upload the `Dockerrun.aws.json` file. Now Elastic beanstalk will start to update our application using our Docker Hub container. This process should take only a few minutes.

Henceforth every time we want to update our application, we only need to push our code to a branch that matches our regex, wait for Docker Hub to build your image and deploy it to AWS

Similarly node.js and react.js app can also be deployed based on their checkin to master branch. We need to change the port in Dockerrun.aws.json and make sure they all have it different and also modify the Dockerfile like below

* **sample node.js application Dockerfile below**

FROM node:10

# Create app directory

WORKDIR /usr/src/app

# Install app dependencies

# A wildcard is used to ensure both package.json AND package-lock.json are copied

# where available (npm@5+)

COPY package\*.json ./

RUN npm install

# If you are building your code for production

# RUN npm ci --only=production

# Bundle app source

COPY . .

EXPOSE 8080

CMD [ "node", "server.js" ]

* **Sample react.js application Dockerfile**

# base image

FROM node:12.2.0-alpine

# set working directory

WORKDIR /app

# add `/app/node\_modules/.bin` to $PATH

ENV PATH /app/node\_modules/.bin:$PATH

# install and cache app dependencies

COPY package.json /app/package.json

RUN npm install --silent

RUN npm install react-scripts@3.0.1 -g --silent

# start app

CMD ["npm", "start"]