CI/CD with Jenkins and AWS EC2 Container Service

Cloud Technology
BSc in Applied Computing Year 4

Rob Shelly
20068406

November 27, 2017

Contents

1 Introduction

This paper will explore continuous integration and continuous deployment (CI/CD) using a very basic sample NodeJS app design with ReactJS. The practical element of this paper will implement CI/CD using a Jenkins server and AWS EC2 Container Service. The app will be deployed though the use of a Jenkins Job through the following workflow:

1. Developer performs a *git push* of the apps source code to GitHub;
2. GitHub, detecting the *push*, uses a **webhook** to a Jenkins Server to trigger a **Jenkins job**;
3. The Jenkins job starts and completes the following build steps:
   - 3.1. Pull the source code of the app, the including a **Dockerfile**, and build an image;
   - 3.2. Push the image to a **Docker Hub registry**;
   - 3.3. Create a new **task definition** on ECS specifying the new image as the source Docker image;
   - 3.4. Update an ECS **Service** to launch the new task definition on an **ECS instance**.
4. The ECS service starts the desired number of task definitions in place of older version(s);
5. The task definition(s) pull the image from Docker Hub and run it in a container on the ECS instance.

Terms highlighted above will be explored in the practical report.

1.1 Objectives

Based on the above work to be completed and the services which will be utilised, the paper has four main objectives:

- Implement a CI/CD pipeline for a simple web app.
- Examine the use of a number of Jenkins plugins to allow the execution of various build steps within a Jenkins job.
- Explored AWS' EC2 Container Server
- Examine the use of AWS' IAM service as a method of securely allowing users and services to perform task on AWS.

2 Tools and Services

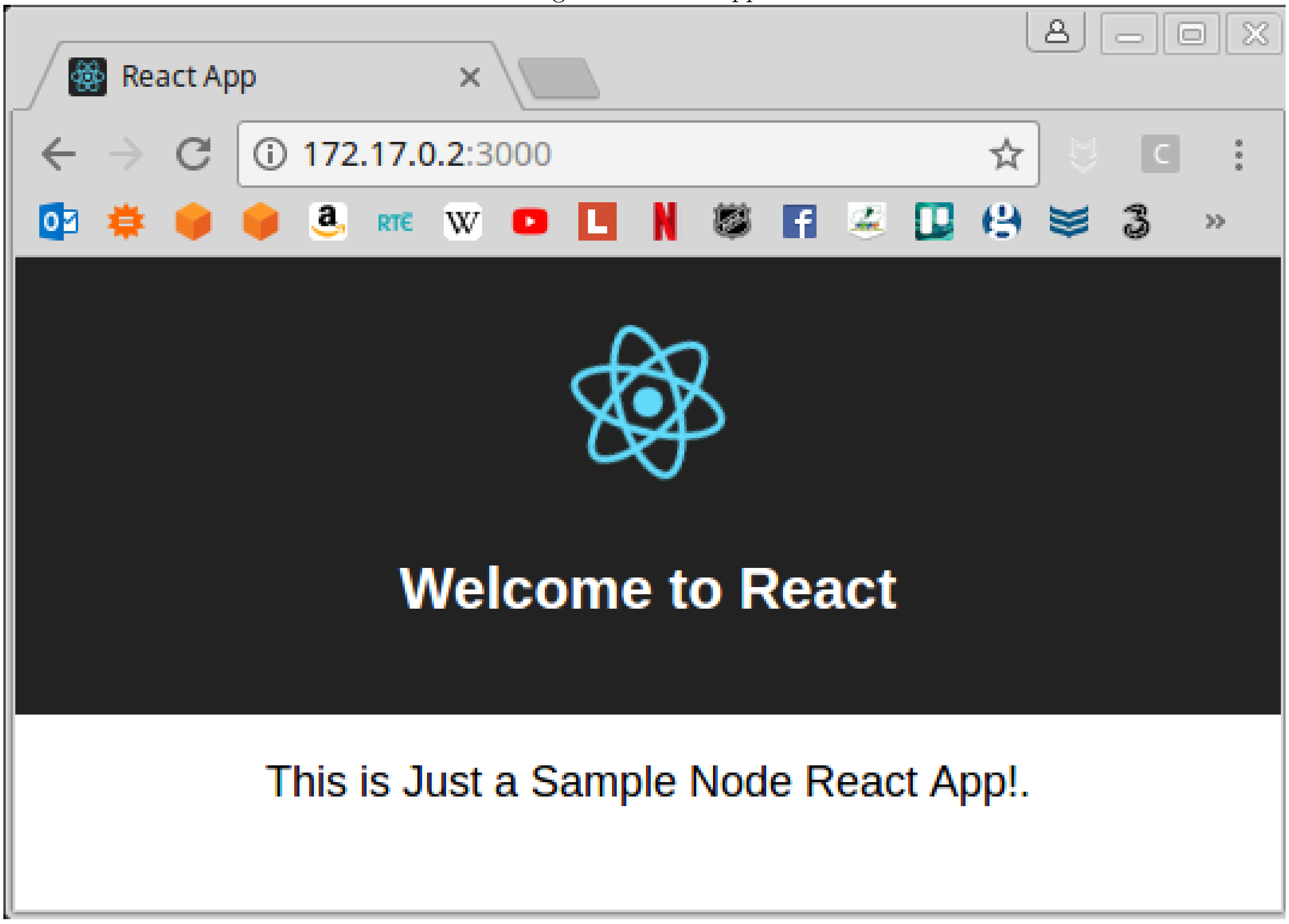
some background on each Jenkins, ECS, IAM, Dockerhub,

3 Practical

3.1 Creating the The App

CI/CD was demonstrated using a simple web app. The objective was to create a web app which can be continuously deployed to AWS. For demonstration purposes a very basic app was created using NodeJS library ReactJS. It is simply a frontend web app that displays a basic ReactJS homepage as a visual indication of a successful deploy. The app running locally can be seen in Figure 1.

Figure 1: React App



Welcome to React

This is Just a Sample Node React App!.

In order to implement continuous integration, the app will be built by means of a Docker image which can be pushed to a Docker Hub registry. True continous integration would often also implements testing before building the image(Docker, 2017). However, as this is just a basic sample app not tests are included.

To *dockerise* the image a *Dockerfile* was added to the source code. This *Dockerfile* specifies a base image for the file (a standard node image for Docker Hub) and copies the source code into the image. It installs the necessary software (npm) and starts the app. It is worth noting that this is starting the app in development mode as opposed to building the app with *npm build* as this is just for demonstration purposes. The *Dockerfile* also exposes port

3000 of the container. This is the port on which the app run in development mode and can be observed in Figure 1. Later, when the image is run on an ECS instance, this containers port will need to me mapped to a host port on the instance.

Code Sample 1: Dockerfile

```
FROM node:carbon
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

3.2 Initial ECS setup

Some initial setup of ECS infrastructure was needed before the Jenkins job could run. This comprised of the following AWS services:

ECS Cluster: An ECS cluster is a grouping of ECS instances. It allows scaling and load balancing of service such as a web app running in containers on multiple instances. As part this paper, autoscaling and load balancing are not implemented, however, they could be easily added by configuring a an Application Load Balancer and configuring the ECS Service to autoscale on some configured Cloudwatch alarms. This is not necessary however for the demonstration purposes of this paper. However, an ECS cluster must still be created, with the instance running the app registered to it.

Code Sample 2: Create ECS Cluster

```
aws ecs create-cluster --cluster-name $CLUSTER_NAME
```

EC2 Instance: This is where the app runs, albeit within a container. The instance must be an ECS optimised instance. This means is includes the ECS agent, a tool which runs on the instance allowing it to register with the cluster (AWS, 2017). It is registers with the cluster by way of user data.

Code Sample 3: Create EC2 Instance

```
aws ec2 run-instances --image-id $AMI_ID --count 1 --instance-type t2.micro --key
    -name $KEY_NAME --security-groups $HTTPSSH_SEC_GROUP $HTTP_GITHUB --iam-
    instance-profile Name=$ECS_INSTANCE_ROLE --user-data file://user-data --tag-
    specifications 'ResourceType=instance,Tags=[{Key=Name,Value='$INSTANCE_NAME
    '}]'
```

Code Sample 4: User Data

```
#!/bin/bash
echo ECS_CLUSTER=node-app-cluster >> /etc/ecs/ecs.config
```

Notice in Code Sample 3 that the instance is configured with the EC2 Instance IAM role. In order for the ECS agent running on the instance to communicate with ECS via an API and register with the cluster, the instance needs permissions. By assigning this IAM role to the instance, it is able to assume this role. The role is granted permissions that allow it to issues commands to ECS such the necessary *Register Container Instance*.

Task Definition: The task definition contains instructions on running the app. For example, the image to use and from where to pull it. In this case, the image is pulled from a registry on Docker Hub.

Code Sample 5: Task Definition Template

```
{
        "family": "node-app",
        "volumes": [
                {
                "name": "my-vol",
                "host": {}
                }
        ],
        "containerDefinitions": [
                {
                        "environment": [],
                        "name": "node-app",
                        "image": "rshelly/sample-node-app:v_%BUILD_NUMBER%",
                        "cpu": 10,
                        "memory": 500,
                        "portMappings": [
                                {
                                        "containerPort": 3000,
                                        "hostPort": 80
                                }
                        ],
                        "mountPoints": [
                                {
                                        "sourceVolume": "my-vol",
                                        "containerPath": "/var/www/my-vol"
                                }
                        ],
                        "essential": true
                }
        ]
}
```

Notice in Code Sample 5 the build number parameter specified as the tag for the image. This is used because every time the Jenkins job runs, building the latest image, it will tag it with a version number (i.e. the build number). The task definition must specify that this latest image is to be used. Therefore, this task definition serves as a template for the Jenkins job to create up-to-date definitions every time a new image it built. For this reason, this task definition template was also placed on the Jenkins server where the job will have access to it.

Also specified in the task definition is the port mappings. The container port is configured as port 3000 as this is the port on which the *Dockerfile*, created above, exposes the app. This is mapped directly to host port 80, as not scaling is being implemented.

As the Jenkins job has not run yet, there is no image in the registry yet. A dummy task definition with a build number of zero was registered with the cluster during the initial build.

Code Sample 6: Register Task Definition

```
sed -e "s;%BUILD_NUMBER%;0;g" ./node-app.json > ./node-app-v_0.json
aws ecs register-task-definition --cli-input-json file://node-app-v_0.json
```

Service: An ECS service runs the task definitions. It also takes care of scaling and load balancing. If these were implement it would have been a simple addtion to the service of adding the Applcation Autoscaling Policy and registering a Target Group for the instance(s).

Every time a new image is built, Jenkins creates a new task definition to run the newly image from the Docker Hub registry. It then updates the service to run the new task definition in place of the old one. Therefore, the service must be created on ECS before the first run of the job. However, because no images have been built yet, it is configured with a desired count of zero. I.e. it will not run any task definitions.

Code Sample 7: Create Service

```
aws ecs create-service --cluster $CLUSTER_NAME --service-name node-app-service --
    task-definition node-app --desired-count 0
```

3.3 Jenkins Setup

The Jenkins server was deployed to an AWS instance. The process of creating the Jenkins server was automated through two scripts. The first script is a python script with uses the Boto3 library to create and configure the necessary AWS infrastructure i.e. security groups and the instance. Note that during later development, more security groups were added to the instance. Due to size this script is not included as an appendix but can be viewed at the Github (see appendix A).

The second script is shown in appendix B. This script installs Jenkins and Nginx to serve the Jenkins web console. It also install Docker and as it will be needed to build the image of the web app. As indicated above, the task definition templates was also added to the *app* folder in the home directory where the Jenkins job can access it. Note that in practice this script was not run, but rather the commands were run on the instance via SSH.

3.4 Jenkins Job

Setting up the Jenkins job,

3.4.1 Credentials

maybe talk about the credentials plugin here and how credentials for AWS, dockerhub are managed

4 Running the Job

Just show that it works on git push

5 Conclusion

TODO

6 Bibliography

AWS. Amazon eCS container agent, 2017. URL
http://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS_agent.html.

Docker. CI/CD, 2017. URL https://www.docker.com/use-cases/cicd.

Appendix A Repos

All scripts and source code for this assignment can be found at the following repositories: //TODO

Appendix B install-jenkins

```
#!/bin/bash

# Install docker, nginx, git
apt-get install -y docker.io nginx git

# Install jenkins
wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key | sudo apt-key add -
sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.
    list'
apt-get update -y
apt-get install -y jenkins

# Add jenkins user to docker group
gpasswd -a jenkins docker

# Configure nginx as proxy for jenkins
rm /etc/nginx/sites-available/default
cat > /etc/nginx/sites-available/jenkins <<EOL
upstream app_server {
server 127.0.0.1:8080 fail_timeout=0;
}

server {
listen 80;
listen [::]:80 default ipv6only=on;
server_name ci.yourcompany.com;

location / {
proxy_set_header X-Forwarded-For \$proxy_add_x_forwarded_for;
proxy_set_header Host \$http_host;
proxy_redirect off;

if (!-f \$request_filename) {
proxy_pass http://app_server;
break;
}
}
}
EOL
ln -s /etc/nginx/sites-available/jenkins /etc/nginx/sites-enabled/

# (re)start the services
service docker restart
service jenkins restart
service nginx restart
```

```
# Install pip
apt-get install -y python-pip python-dev build-essential
# Fix for pip
export LC_ALL=C

# Install aws cli
pip install awscli --upgrade

mkdir app
chown jenkins: app
```