CI/CD with Jenkins and AWS EC2 Container Service

Cloud Technology
BSc in Applied Computing Year 4

Rob Shelly
20068406

November 28, 2017

Contents

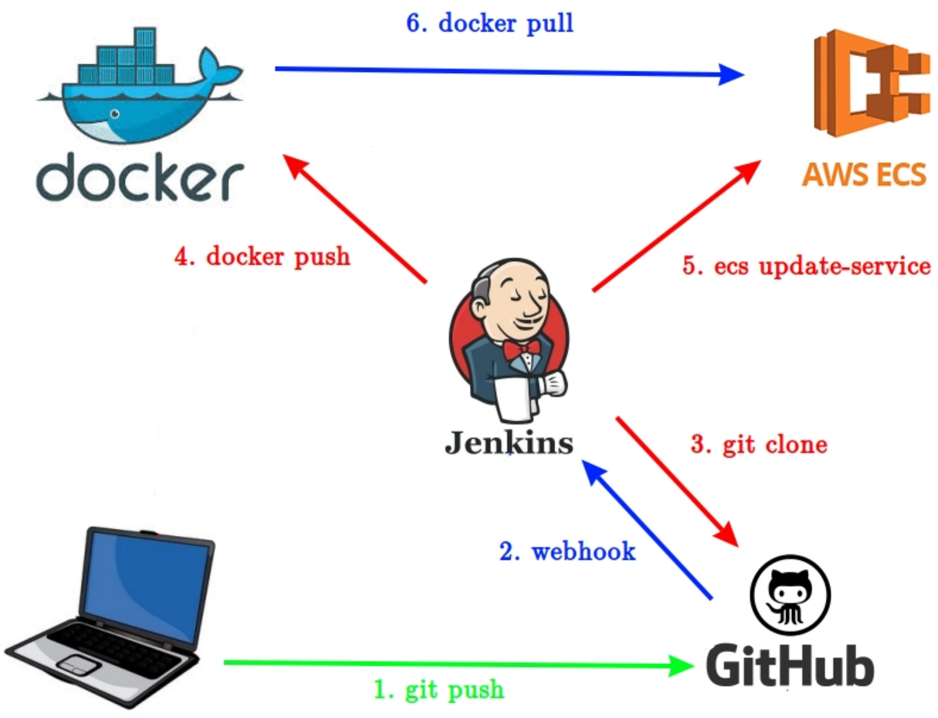
1 Introduction

This paper will explore continuous integration and continuous deployment (CI/CD) using a very basic sample NodeJS app designed 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 by a Jenkins build server 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. A Jenkins job starts and completes the following build steps:
   3.1. Pulls the source code of the app, the including a **Dockerfile**, and builds an image;
   3.2. Pushes the image to a **Docker Hub registry**;
   3.3. Creates a new **task definition** on ECS specifying the new image as the source Docker image;
   3.4. Updates 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.

Figure 1: Workflow Overview



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.
- Explore 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.

1.2 Background

CI/CD is interchangeably referred to as Continuous Integration/Continuous Deployment and Continuous Integration/Continuous Delivery. In order to implement a CI/CD workflow as part of this paper it is worth first understanding what each of these terms means. This will provide a better understanding of the objective of this paper and indication as to whether continuous deployment or continuous delivery will be implemented.

1.2.1 Continuous Integration

CI is the easiest of the terms to define. It is the practice of continually preparing a product for release. This takes the form of developers continually committing their work to the source code as soon as any body of work has been completed. Commits to source code usually undergo some form of testing and building each time they are made. This means that code is constantly committed, tested and built at each iteration of development instead of when it comes to release time (Pittet, 2017).

The CI aspect of this project refers to the image build. Every time a commit is made to GitHub, the source code will be cloned by Jenkins and a Docker image built. As this is just a very basic sample web app there is no testing in place. However, testing before building is generally a main aspect on continuous integration.

1.2.2 Continuous Delivery

Continuous delivery is a step further than continuous integration. It refers to the act of continuously pushing the built code to a server. Thus, each time a commit is made and the code is tested and built (CI) it is then deployed to development server. Deploying the code to a production server must then be manually triggered if it is decided it is ready for release (Ellingwood, 2017).

The continuous delivery element of this project refers to deploying the app to ECS where it can then be viewed by a web browser.

1.2.3 Continuous Deployment

Just as continuous delivery is a step beyond to continuous integration, continuous deployment is the next step. It removes the manual trigger to deploy to production servers. Therefore, code is constantly deployed to production at each commit, possibly many times a day (Ramos, 2016).

Although it may seem at first that this project is deploying straight to production, it will be observed later that this is not the case. The Dockerfile which defines how the image is built runs the Node app in development mode. This is not suitable for an app in production and therefore it cannot be said that this project implements continuous deployment.

2 Tools and Services

This paper will utilize a number of different tools and services to implement CI/CD for the sample web app. Part of the objective of this paper is to gain an understanding of each of these tools and explore how they can be integrated to implement the desired workflow in a secure manner.

2.1 Jenkins

Jenkins is an automated build server used for CI/CD. It features a large library of plugins for integrating the server with various services, for example GitHub and Docker Hub which will be explored in this paper. It simple to set up and features a web app interface for easy configuration. It also features a REST API and a Java based CLI for interfacing with the server (Jenkins, 2017). Only the web console will be used in this paper however. The range of plugins available makes Jenkins an ideal build server for this project.

2.2 ECS

ECS is AWS' container management service. It allows running Docker images on EC2 instances. Application images can be scaled and load balanced across multiple containers and multiple instances. Through the use of task definitions and services ECS abstracts much the of the management that would be required if deploying a scalable application using self managed docker containers (AWS, 2017c). ECS' ability to deploy Docker images makes it an ideal candidate for this workflows as Jenkins will be building the web app as Docker images.

2.3 IAM

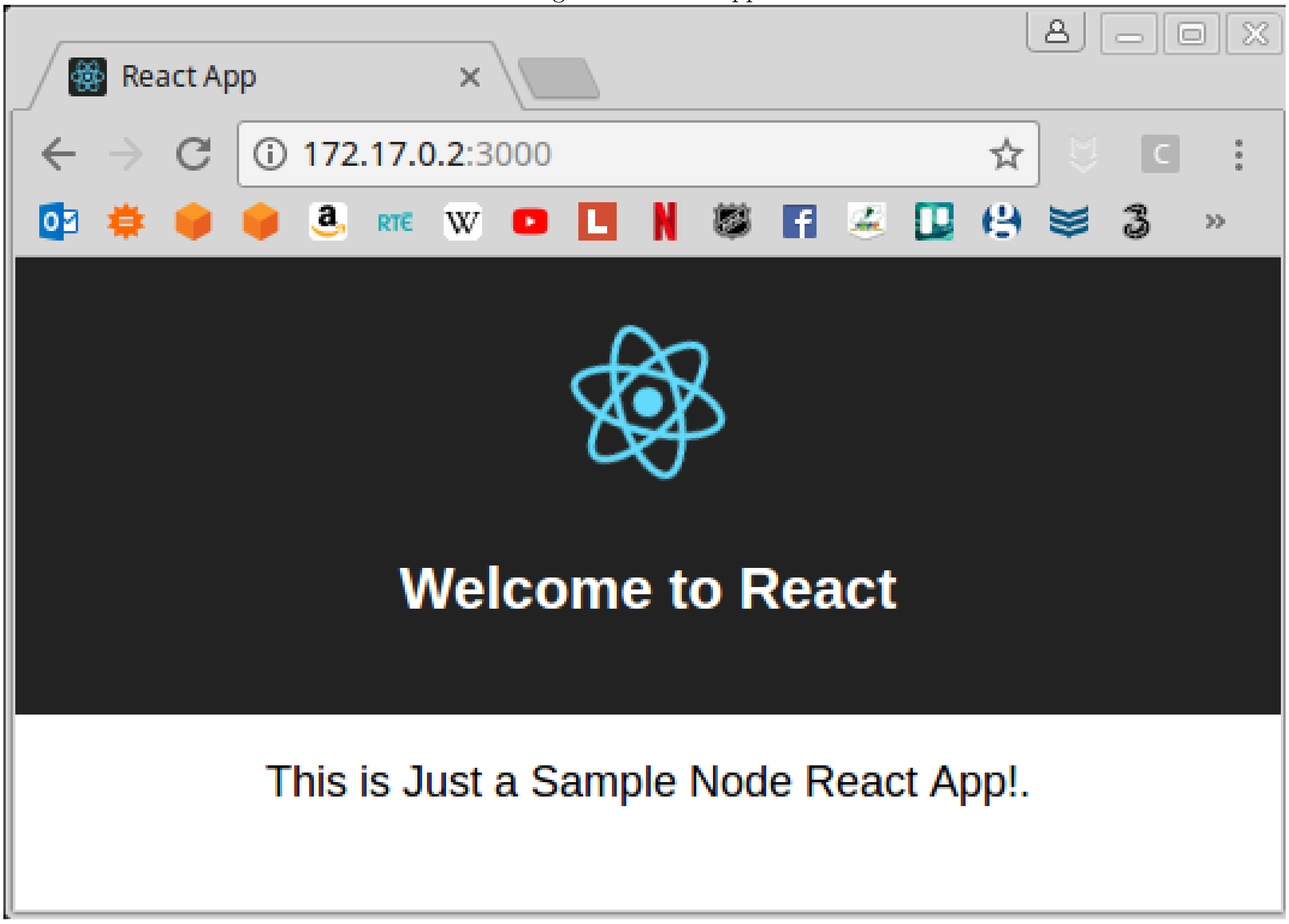
Identity Access Management is AWS' service for controlling access to infrastructure. It provides secure methods for granting users and infrastructure permissions to communicate and integrate with other resources (AWS, 2017d). As the Jenkins server will be creating/modifying resources within AWS, it will need a set a credentials to do so. This may pose a security threat. However, with the use of IAM, a user can be created for Jenkins with it's own set of access credentials. The user can be given bespoke policies which allow it to perform only the actions it is required to perform.

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 2.

Figure 2: 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 continuous integration would often also implement testing before building the image(Docker, 2017). However, as this is just a basic sample app no 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 2. Later, when the image is run on an ECS instance, this container port will need to be 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 AWS setup

Some initial setup of AWS infrastructure was needed before the Jenkins job could run.

3.2.1 IAM Resources

The following IAM resources were needed for the project:

Jenkins User: As part of the build process the Jenkins job will execute a number of AWS CLI commands:

- Create an ECS Task Definition;
- Describe an ECS Task Definition;
- Update and ECS Service;
- Describe an ECS Service.

The most secure method of allowing this was to create an IAM user for Jenkins. Also created were two policies which together allow only the above actions. he policies were attached to the Jenkins user. An Access Key was generated for the Jenkins user to perform AWS CLI commands.

Code Sample 2: Task Definition Policy

```
{
        "Version": "2012-10-17",
        "Statement": [
                {
                        "Sid": "VisualEditor0",
                        "Effect": "Allow",
                        "Action": [
                                "ecs:UpdateService",
                                "ecs:DescribeServices"
                        ],
                        "Resource": "*"
                }
        ]
}
```

Code Sample 3: Service Policy

```
{
        "Version": "2012-10-17",
        "Statement": [
                {
                        "Sid": "VisualEditor0",
                        "Effect": "Allow",
                        "Action": [
                                "ecs:RegisterTaskDefinition",
                                "ecs:DescribeTaskDefinition"
                        ],
                        "Resource": "*"
                }
        ]
}
```

ECS Instance Role: This role was created and the predefined policy *AmazonEC2ContainerServiceforEC2Role* attached to it. The EC2 instance on which the app runs is registered to an ECS cluster. It uses an ECS agent to register to the cluster upon creation. In order for the agent to do so, it must communicate with ECS via an API call (AWS, 2017a). This policy is used to grant and instance permission to communicate and interact with resources within ECS.

3.2.2 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 an Application Load Balancer and configuring the ECS Service to autoscale on some configured Cloudwatch alarms. This is not necessary however for the demonstration purpose of this paper. However, an ECS cluster must still be created, with the instance running the app registered to it.

Code Sample 4: Create ECS Cluster

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

3.2.3 EC2 Instance

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

Code Sample 5: 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 6: User Data

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

Notice in Code Sample 5 that the instance is configured with the ECS Instance Role. By assigning this IAM role to the instance, it is able to assume the role, thus granting it permission to issue commands to ECS such as the necessary *Register Container Instance*.

3.2.4 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 7: 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 7 the build number parameter is 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 mapping. 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 8: 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
```

3.2.5 Service

An ECS service runs the task definitions. It also takes care of scaling and load balancing. If these were implemented it would have been a simple addition to the service of adding the Application 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 built 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 9: 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. Due to size this script is not included as an appendix but can be viewed at the Github (see appendix A).

During development the security groups for the instance were refined to the following:

- **HTTP(S) and SSH from working locations:** A security group was added that allowed the aforementioned protocols for each of IP address from which development was completed e.g. WIT campus IP address.
- **HTTP(S) from GitHub's address range:** Builds will be triggered by a webhook on GitHub which will send push notifications to the Jenkins server. Therefore traffic from GitHub needed to be permitted. The security group was configured to allow traffic from GitHub's publish address range. It is recommended that authentication with HTTPS is used rather than whitelisting IP address ranges (GitHub, 2017). However, for the purposes of this paper TLS was not implemented on the Jenkins server so the later method was implemented.

The second script is shown in appendix B. This script installs Jenkins and Nginx to serve the Jenkins web console. It also installs Docker and as it will be needed to build the image of the web app. As indicated above, the task definition template 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.

3.3.1 Jenkins Plugins

A number of Jenkins plugins were installed to provide some of the necessary functionality.

GitHub Plugin: This plugin allows Jenkins to trigger jobs when a push to GitHub is performed.

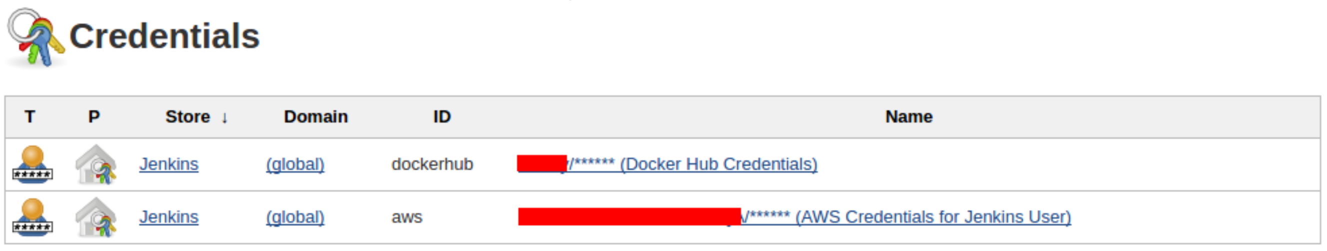
CloudBees Docker Build and Publish Plugin: This plugin allows the Jenkins job to build Docker images using a Dockerfile and push them to Docker Hub registries using a Dockerfile.

Credentials Plugin This plugin allows Jenkins to securely store and use credentials in various forms (e.g. private keys, username:password pairs). Credentials can be used during build stages but will be obscured if used printed to the console.

This plugin was used to overcome a key issue encountered during the initial setup of the Jenkins server. The tutorial used as a guide for this paper instructs to configure both Docker Hub and AWS credentials directly on the server (using *docker login* and *aws configure*). However, when doing so, issues were encountered regards the Jenkins user a permission for running these commands and creating credentials files in the home folder. This plugin provided an easy and safe solution as the credentials can be safely stored and used, while also easily managed via the Jenkins console.

Credentials for both Docker Hub and AWS were configured on the Jenkins console once the plugin was installed.

Figure 3: Credentials



Credentials

T	P	Store ↓	Domain	ID	Name
		Jenkins	(global)	dockerhub	[redacted]/****** (Docker Hub Credentials)
		Jenkins	(global)	aws	[redacted]/****** (AWS Credentials for Jenkins User)

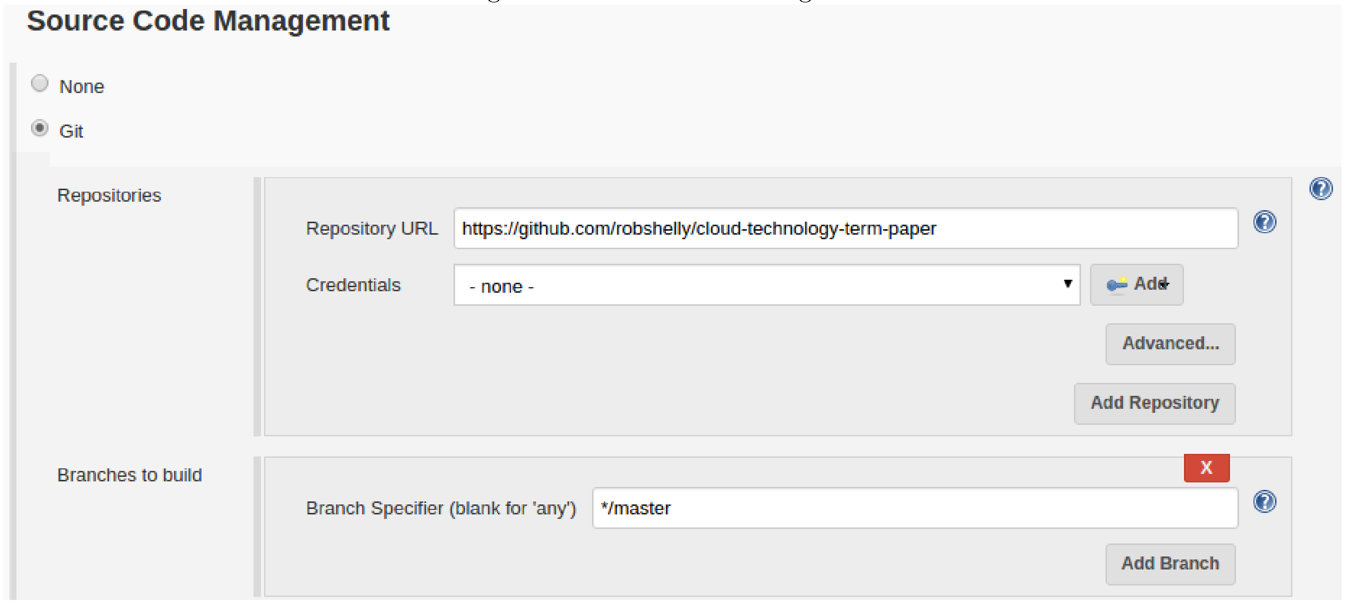
3.4 Jenkins Job

The last step was to create the Jenkins job. This was created and configured on the Jenkins console. Configuration of the main build steps is outlined below.

3.4.1 Cloning the Source Code

The URL for the repo containing the source code was configured. Credentials can be configured for pulling from a private repo. However, the source code for the app is contained in a public repo so credentials are not needed.

Figure 4: Source Code Management



Source Code Management

None
Git [✓]

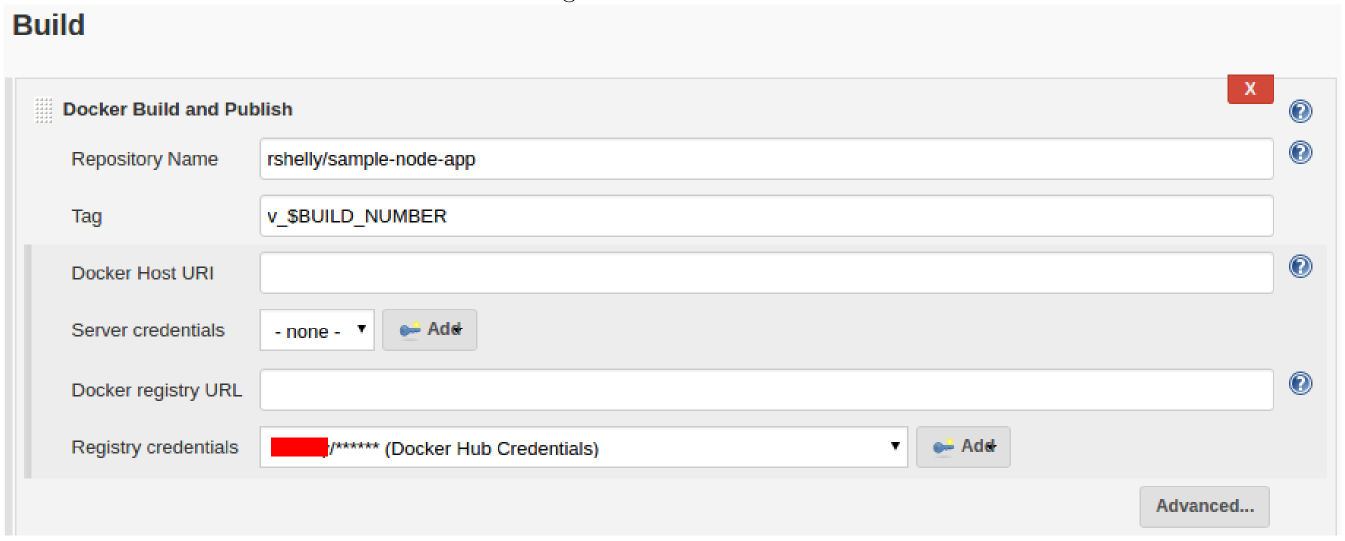
Repositories
Repository URL: https://github.com/robshelly/cloud-technology-term-paper
Credentials: - none -

Branches to build
Branch Specifier (blank for 'any'): */master

3.4.2 Build and Push Image

This step was possible using the Docker Build and Publish Plugin. To configure it, the repository name for the image was entered. The Tag for the image is configured as a version number created using the BUILD_NUMBER argument. This argument is made available to the Jenkins job and corresponds to the build number of the jenkins job. The credentials for the Docker Registry are also provided, allowing the image to be pushed to the registry.

Figure 5: Docker Build



Build

Docker Build and Publish
Repository Name: rshelly/sample-node-app
Tag: v_$BUILD_NUMBER
Docker Host URI:
Server credentials: - none -
Docker registry URL:
Registry credentials: [redacted]/****** (Docker Hub Credentials)

3.4.3 Shell Script

The final step of the build is a shell script. This takes care of deploying the image to AWS.

Code Sample 10: Shell Script

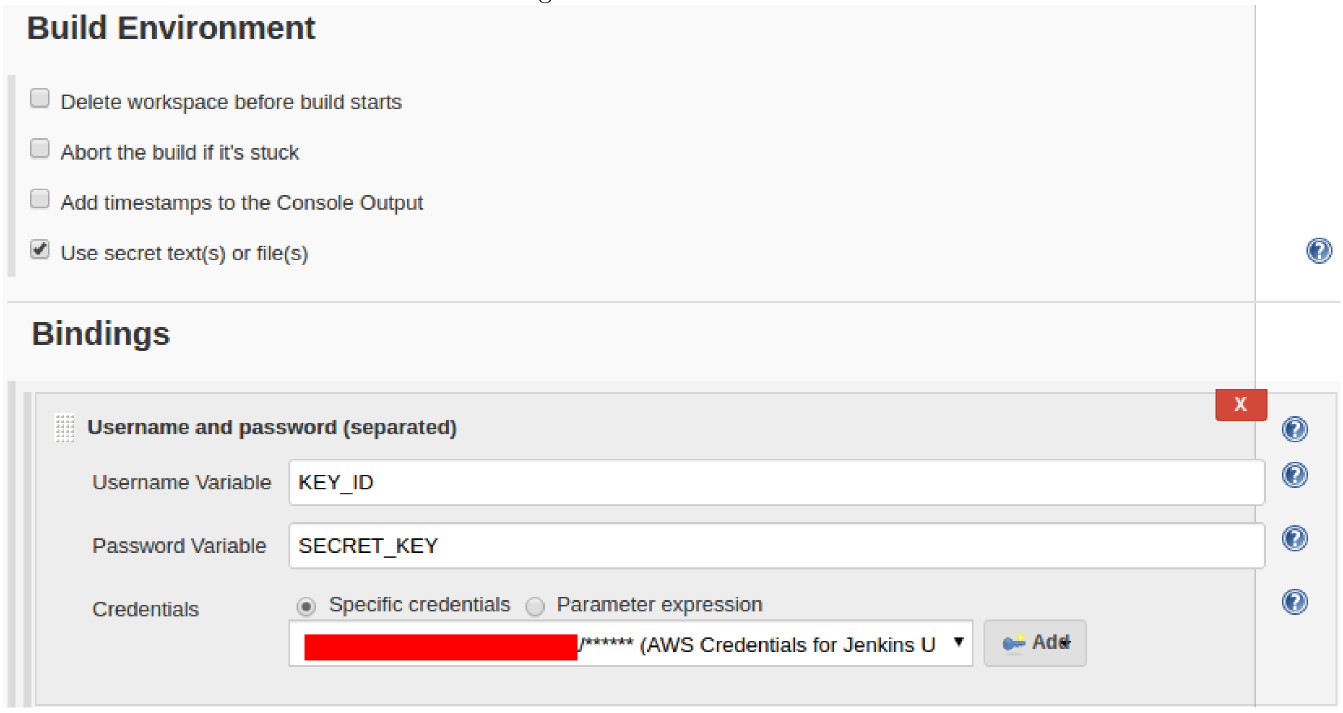
```
SERVICE_NAME="node-app-service"
IMAGE_VERSION="v_"${BUILD_NUMBER}
TASK_FAMILY="node-app"
CLUSTER_NAME="node-app-cluster"

# Create a new task definition for this build
sed -e "s;%BUILD_NUMBER%;${BUILD_NUMBER};g" /home/ubuntu/app/node-app.json > node
     -app-v_${BUILD_NUMBER}.json
AWS_ACCESS_KEY_ID=$KEY_ID AWS_SECRET_ACCESS_KEY=$SECRET_KEY aws ecs --region eu-
     west-1 register-task-definition --family $TASK_FAMILY --cli-input-json file
     ://node-app-v_${BUILD_NUMBER}.json

# Update the service with the new task definition and desired count
TASK_REVISION=`AWS_ACCESS_KEY_ID=$KEY_ID AWS_SECRET_ACCESS_KEY=$SECRET_KEY aws
     ecs --region eu-west-1 describe-task-definition --task-definition
     $TASK_FAMILY | egrep "revision" | tr "/" " " | awk '{print $2}' | sed 's/"$
     //'`
DESIRED_COUNT=`AWS_ACCESS_KEY_ID=$KEY_ID AWS_SECRET_ACCESS_KEY=$SECRET_KEY aws
     ecs --region eu-west-1 describe-services --services ${SERVICE_NAME} --
     cluster $CLUSTER_NAME | egrep -m 1 "desiredCount" | tr "/" " " | awk '{print
      $2}' | sed 's/,$//'`
if [ ${DESIRED_COUNT} = "0" ]; then
DESIRED_COUNT="1"
fi

AWS_ACCESS_KEY_ID=$KEY_ID AWS_SECRET_ACCESS_KEY=$SECRET_KEY aws ecs --region eu-
     west-1 update-service --cluster $CLUSTER_NAME --service ${SERVICE_NAME} --
     task-definition ${TASK_FAMILY}:${TASK_REVISION} --desired-count ${
     DESIRED_COUNT}
```

The actions performed by the shell script can be summarised as follow:

1. From the task definition template, create a new task definition specifying the latest image version;
2. Register the new task definition;
3. Query the task definition for the Revision Number of the latest registered definition.
4. Query the service to find the number of desired tasks (this step would accommodate cases where autoscaling/load balancing are implemented, finding out the current number of tasks and ensuring the same number of updated tasks are started);
5. Update the service to start the correct number of the tasks defined by the latest revision.

Notice the use of the KEY_ID and SECRET_KEY variables in the shell script. These are the AWS credentials that were added to Jenkins. They are made available to the job via these variable names in the job configuration.

Figure 6: AWS Credentials



Build Environment

- [ ] Delete workspace before build starts
- [ ] Abort the build if it's stuck
- [ ] Add timestamps to the Console Output
- [✓] Use secret text(s) or file(s)

Bindings

Username and password (separated)

Username Variable: KEY_ID

Password Variable: SECRET_KEY

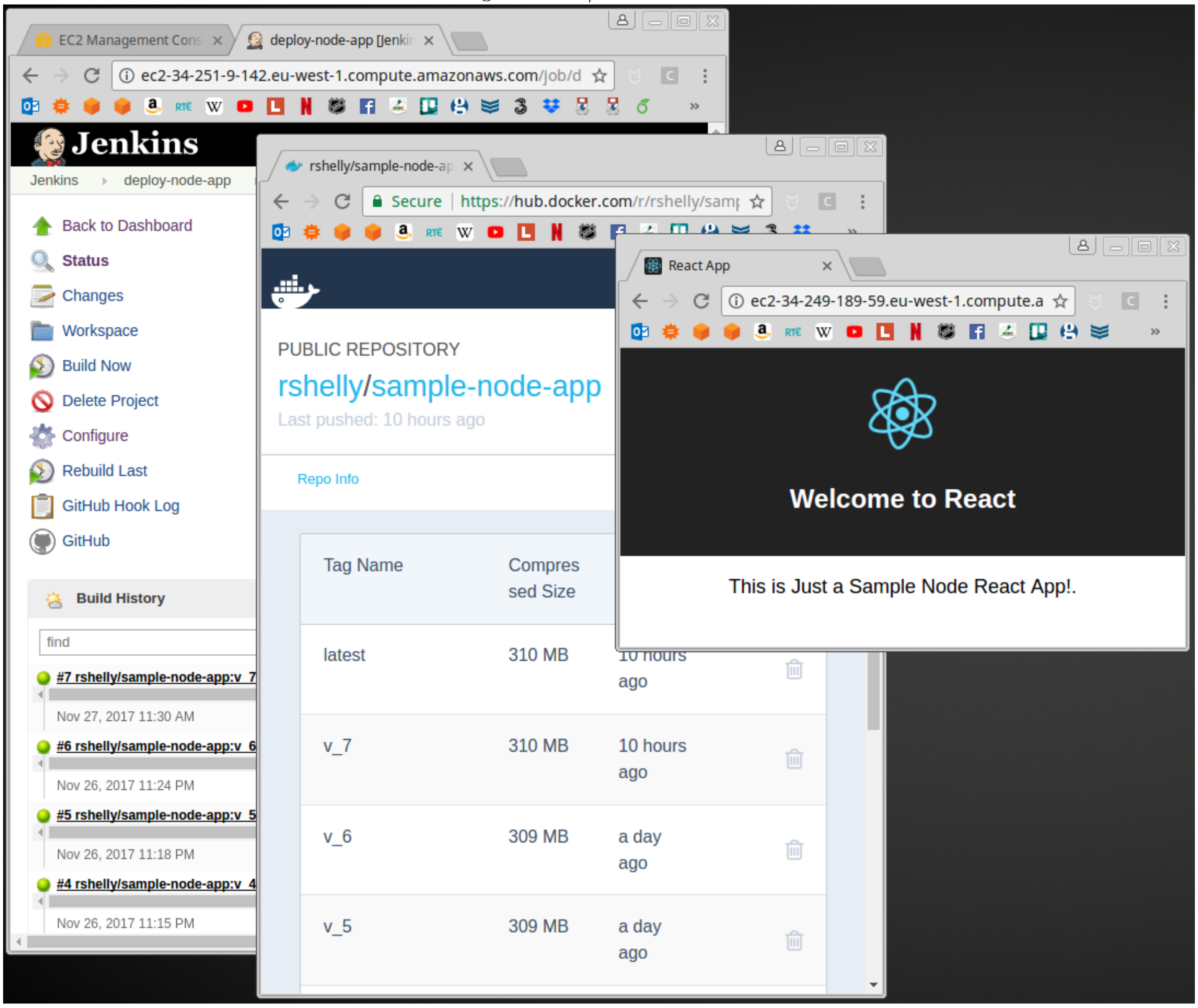
Credentials: [✓ Specific credentials] Parameter expression

[redacted]/****** (AWS Credentials for Jenkins U

4 Running the Job

Once all the infrastructure was created and the Jenkins job configured the CI/CD of the web app could be demonstrated. To do this a change was made to the app's source code. The changes where then committed and push to GitHub using Git. A job was immediately observed starting on the Jenkins server. Once the job was finished the app could be viewed by visiting the DNS of the EC2 instance. Figure 7 shows the successful results of a number a jobs, i.e. the various versions of the image on Docker Hub and the app running on the

Figure 7: CI/CD in Action



ec2-34-251-9-142.eu-west-1.compute.amazonaws.com/job/d

#7 rshelly/sample-node-app:v_7
Nov 27, 2017 11:30 AM
#6 rshelly/sample-node-app:v_6
Nov 26, 2017 11:24 PM
#5 rshelly/sample-node-app:v_5
Nov 26, 2017 11:18 PM
#4 rshelly/sample-node-app:v_4
Nov 26, 2017 11:15 PM

https://hub.docker.com/r/rshelly/samp

rshelly/sample-node-app

Last pushed: 10 hours ago

Tag Name	Compressed Size	
latest	310 MB	10 hours ago
v_7	310 MB	10 hours ago
v_6	309 MB	a day ago
v_5	309 MB	a day ago

ec2-34-249-189-59.eu-west-1.compute.a

Welcome to React

This is Just a Sample Node React App!.

5 Conclusion

On successfully completing the practical element of this paper the sample node app could be easily modified and updated on the server using the automated Jenkins pipeline. This successfully demonstrated the continuous integration and continuous deployment workflow that was desired.

The workflow demonstrated how Jenkins suite of plugins can be used to create jobs or indeed pipelines that integrate the various service needed to build, test and deploy a web app. This consisted of integrating the source code on GitHub, Docker images and registries and various AWS services for deployment and management. The practical work also successfully implemented a secure methods od providing credentials to the Jenkins server, including both Docker Hub credentials and AWS credentials created using IAM.

6 Bibliography

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

AWS. Amazon eCS container instance iam role, 2017b. URL http://docs.aws.amazon.com/AmazonECS/latest/developerguide/instance_IAM_role.html.

AWS. Amazon Elastic Container Service, 2017c. URL https://aws.amazon.com/ecs/.

AWS. What is iam, 2017d. URL http://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html.

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

Justin Ellingwood. An introduction to continuous integration, delivery, and deployment, May 2017. URL https://www.digitalocean.com/community/tutorials/an-introduction-to-continuous-integration-delivery-and-deployment.

GitHub. GitHub's IP addresses, 2017. URL https://help.github.com/articles/github-s-ip-addresses/.

Jenkins. Jenkins, 2017. URL https://jenkins.io/.

Sten Pittet. Continuous integration vs. continuous delivery vs. continuous deployment, 2017. URL https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd.

Marcia Ramos. Continuous integration, delivery, and deployment with gitlab, 08 2016. URL https://about.gitlab.com/2016/08/05/continuous-integration-delivery-and-deployment-with-gitlab/.

Appendix A Repos

All code used for this paper can be view at the following repos:

1. Cloud Technology Term Paper
2. Deploying a Jenkins Server on AWS

Appendix B install-jenkins.py

```
#!/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
```