



Waterford Institute of Technology

Docker & RabbitMQ ECS Container & Scaling Container Labs

Name: Eoin Dalton

Student Number: 20070289

Module: Cloud Computing

Contents

Introduction	2
1.0 Docker & RabbitMQ Exercise	2
2.0 ECS Container Lab	6
3.0 Scaling Container Exercise	8
Conclusion.....	9
Figure 1.1: Running RabbitMQ Management Container.	2
Figure 2.1: some-rabbit IP Address	2
Figure 3.1: RabbitMQ Management Plugin.	3
Figure 4.1: RabbitMQ Management Plugin Empty Queue	3
Figure 5.1: Script for send.py on GitHub.....	3
Figure 6.1: Editing host value in send.py script.	3
Figure 7.1: Running Containers.....	4
Figure 8.1: Running python scripts.	4
Figure 9.1: Dockerfile Script.	5
Figure 10.1: Building send and receive images.....	5
Figure 11.1: Docker Images.....	5
Figure 12.1: Pushing images to Docker Hub.	5
Figure 13.1: Showing Images on Docker Hub.	6
Figure 14.1: Installing AWS CLI.	6
Figure 15.2: Configuring AWS CLI.	6
Figure 16.2: Creating Cluster in AWS through CLI.	6
Figure 17.2: Cluster Successfully Created.	7
Figure 18.2: Script for Linking Instance to my Cluster.	7
Figure 19.2: Clone App from GIT.....	7
Figure 20.2: Registering Task.	7
Figure 21.2: Running Task.	7
Figure 22.3: Creating Cluster for Fargate.....	8
Figure 23.3: Cluster Created.	8
Figure 24.3: Load Balancer Created.....	8
Figure 25.3: Errors trying to create service.....	9

Introduction

The first part of practical 1 we were asked to interact with “Docker” by pulling different containers down from docker repository running the containers and learning about the different options available with docker such as networking, creating docker images, docker volumes and basic commands like running, inspecting, stopping and restarting the containers. Once we had a handle on the basic commands we then asked to do an exercise. The exercise involved creating two python scripts receive and send that would interact with “RabbitMQ” which a queueing service is built for these types of apps so that there is some control on the way the messages are sent. Once this was completed and the scripts were working properly, we were then asked to create image with the use of a docker file and push them up to our docker hub accounts.

In lab two we were asked to install “AWS CLI” on our local machine. Then use the “AWS CLI” to create a “AWS ECS Cluster” and launch an ECS instance into the cluster that was created. When all of this was done, we needed to create a Docker image for an PHP application and upload this to docker. We then create a task on the ECS instance that would link and run the PHP application and finally to run the ECS task with AWS.

1.0 Docker & RabbitMQ Exercise

To begin we were asked to run the RabbitMQ container. This was done with this command:

```
docker run -d -e RABBITMQ_NODENAME=my-rabbit --name some-rabbit -p 8080:15672 rabbitmq:3-management
```

This command says run the RabbitMQ container. Give it a name of some-rabbit and the node name of my-rabbit. The -p looks at ports and this says go from port 8080 and map to destination port 15672. Note that after running this command if docker can't find the image locally it will pull it from the docker repository. Please refer to figure 1.1 to see the RabbitMQ management system running on my local machine.

After this we could view a management plugin. The management plugin is a graphical interface to view and monitor the queueing system that is in place. To view this, we needed to locate the IP address of the RabbitMQ container as shown in figure 2.1. You could view the IP of a container with the docker inspect followed by the container name or container ID. Once we had the IP address I could then use <http://172.17.0.1:15672> this would bring up the management plugin as seen in figure 3.1 below. The login was guest and the password is guest. If you refer to figure 4.1 below you can see an empty

queue as nothing has been configured at this this.

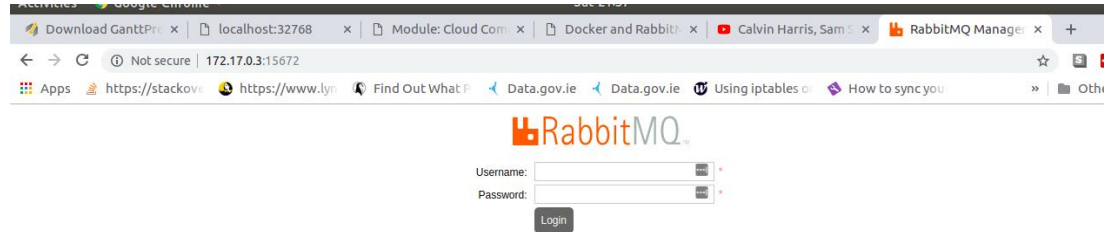


Figure 3.1: RabbitMQ Management Plugin.

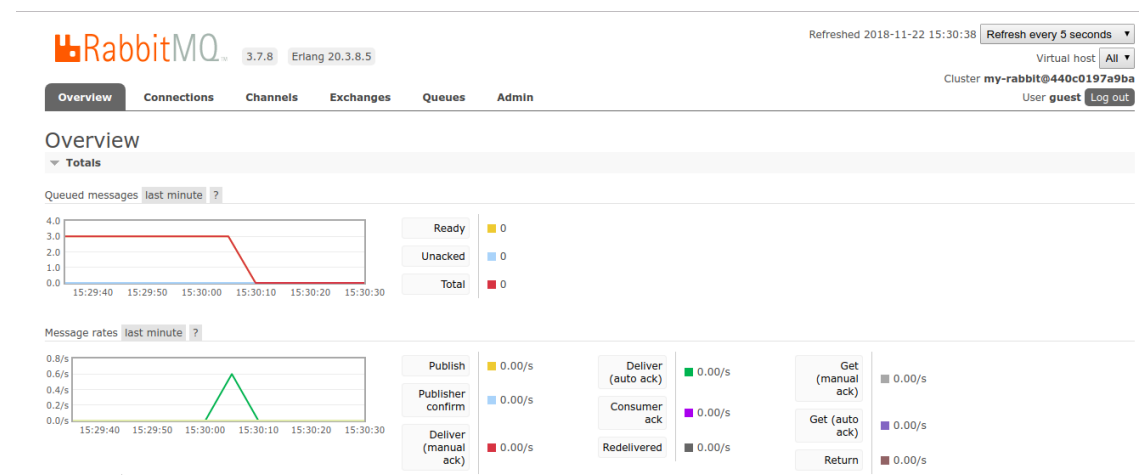


Figure 4.1: RabbitMQ Management Plugin Empty Queue

The next task was to create two python scripts that would send and receive a hello world message. The two scripts were located on GitHub where you can copy the code or clone the git repository. If you refer to figure 5.1 you will see the GitHub repository for send.py script. Once we copy the script, we needed to create an empty directory to put these in mine was mkdir CloudExercise. I then cd to this directory where I create two scripts. Send and receive:

- Sudo nano send.py
- Sudo nano recieve.py

It was just a matter of pasting the corresponding code into each script and that was the two scripts created. The next task was to configure the two files so that they are pointing to the RabbitMQ management system. This was done by replacing the **host** value as seen in figure 6.1 with the IP address of some-rabbit container. After the two files were configured the next step was to start the send and receive containers. Command see below

```
sudo docker run -v $PWD:/code --name send --link=some-rabbit -it python:2 bash
```

The command above runs some-rabbit and links it to the volume containing the two python scripts on my local machine.

If you refer to figure 7.1 you can see the two containers running.

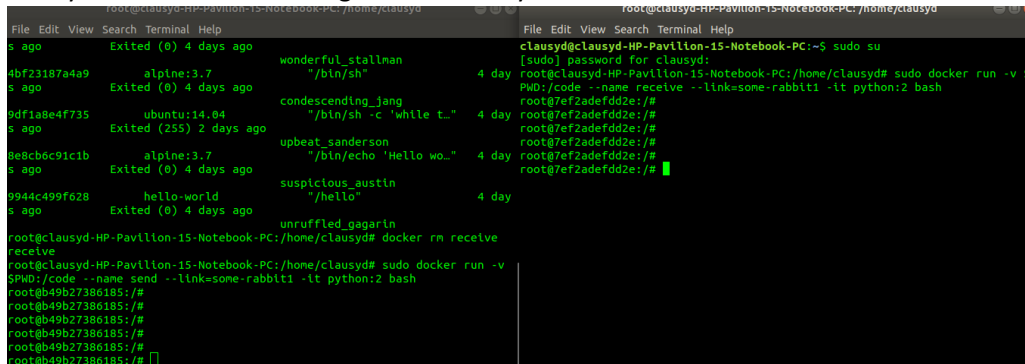


Figure 7.1: Running Containers.

Once the two containers were running, I needed to install pika with a pip install as seen in figure 8.1. Pika is used to run the python scripts. The command to run the two python scripts is:

- Python send.py
- Python review.py

This is show in figure 8.1 below. Each time this command is run form the send container it will send a hello world to the RabbitMQ management service and the will then forward to the receive container again we can see this in figure 8.1.

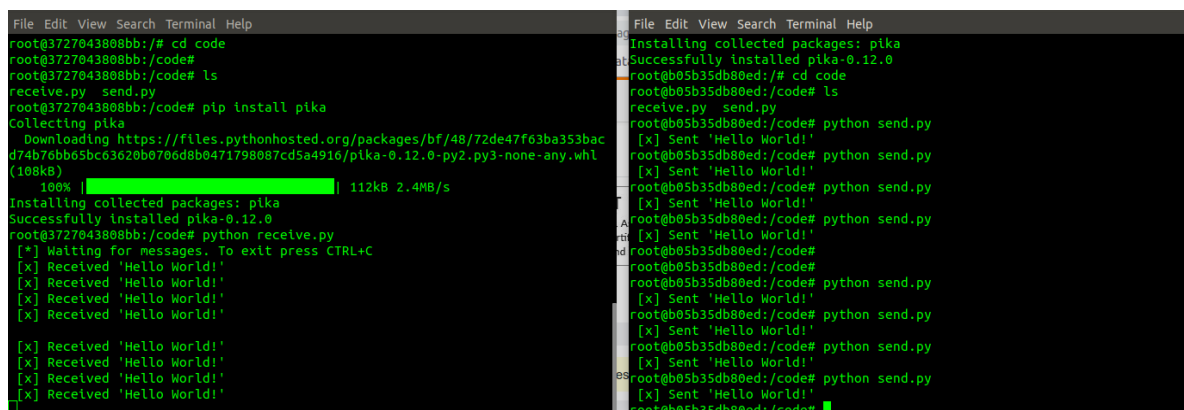
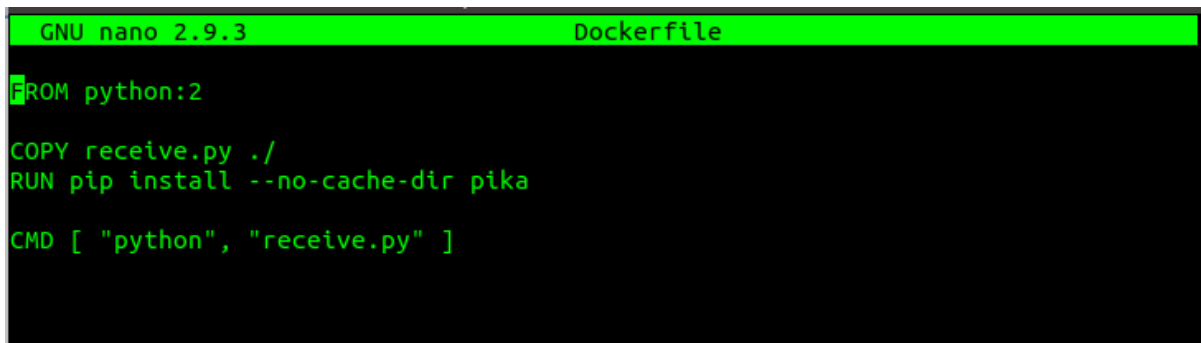


Figure 8.1: Running python scripts.

The next task we were asked to do was to create a docker file for the send and receive scripts. In a “Dockerfile” we create a bash script with commands that will build an image of the send and receive scripts. What I done was to create two empty directories. In one directory I placed the send python script and in the other directory I placed the receive python script. Now in each directory I needed to create two “Dockerfiles” that would contain the commands to build each image as seen in figure 9.1.



```

GNU nano 2.9.3 Dockerfile
FROM python:2

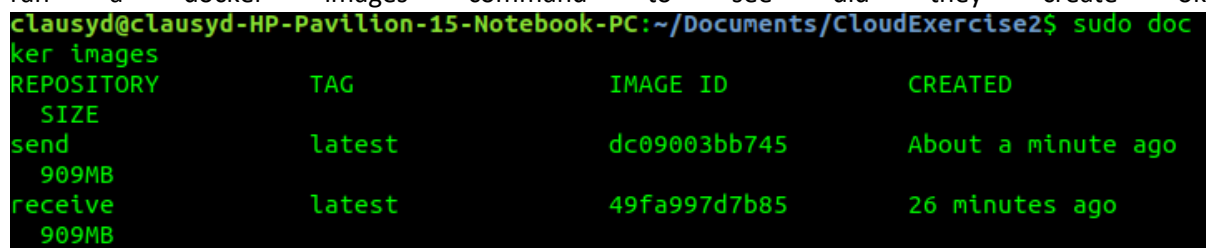
COPY receive.py ./
RUN pip install --no-cache-dir pika

CMD [ "python", "receive.py" ]

```

Figure 9.1: Dockerfile Script.

Once the two Docker files were created, I then run the build commands as seen in figure 10.1. The -t in the command is putting a tag on the file with the tag being receive. This will automatically pick up the Dockerfile and run each command contained in the script. If you refer to figure 11.1 you will see I ran a docker images command to see did they create ok.



```

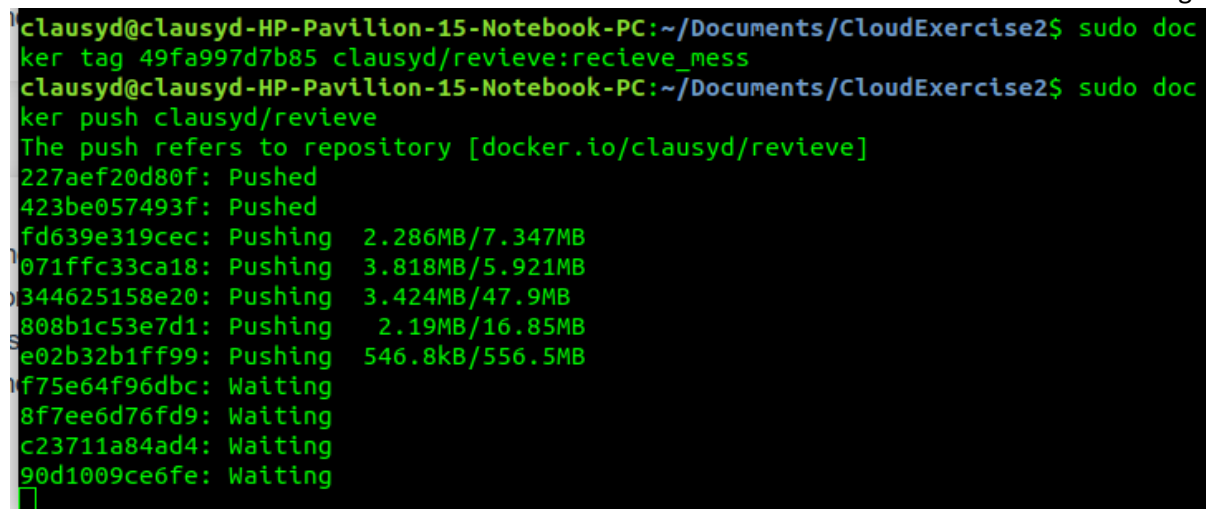
clausyd@clausyd-HP-Pavilion-15-Notebook-PC:~/Documents/CloudExercise2$ sudo docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED
send	latest	dc09003bb745	About a minute ago
receive	latest	49fa997d7b85	26 minutes ago

Figure 11.1: Docker Images.

For last part of the exercise asked us to push are images to docker hub. I created a Docker Hub account to perform this action. If you refer figure 12.1 you will see the push command the was issued for the receive image.



```

clausyd@clausyd-HP-Pavilion-15-Notebook-PC:~/Documents/CloudExercise2$ sudo docker tag 49fa997d7b85 clausyd/revieve:recieve_mess
clausyd@clausyd-HP-Pavilion-15-Notebook-PC:~/Documents/CloudExercise2$ sudo docker push clausyd/revieve
The push refers to repository [docker.io/clausyd/revieve]
227aef20d80f: Pushed
423be057493f: Pushed
fd639e319cec: Pushing 2.286MB/7.347MB
071ffc33ca18: Pushing 3.818MB/5.921MB
344625158e20: Pushing 3.424MB/47.9MB
808b1c53e7d1: Pushing 2.19MB/16.85MB
e02b32b1ff99: Pushing 546.8kB/556.5MB
f75e64f96dbc: Waiting
8f7ee6d76fd9: Waiting
c23711a84ad4: Waiting
90d1009ce6fe: Waiting

```

Figure 12.1: Pushing images to Docker Hub.

Finally, I am showing in figure 13.1 that the push of each image was a success.



 clausyd/receive public	0 STARS	1 PULLS	➤ DETAILS
 clausyd/send public	0 STARS	1 PULLS	➤ DETAILS

Figure 13.1: Showing Images on Docker Hub.

2.0 ECS Container Lab

The first step in this practical was to install “AWS CLI on the local machine. If you refer to figure 14.2 you will see me installing the awscli. This was straightforward as all the command we are giving in the practical. The next part was to configure awscli. We were each giving a login for an AWS account and this need to be configure from the cli on the local desktop. If you refer to figure 15.2 you will see the configurations.

```
clausyd@clausyd-HP-Pavilion-15-Notebook-PC:~$ aws configure
AWS Access Key ID [None]: AKIAIKTHOCXRH7PK2EJA
AWS Secret Access Key [None]: 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
Default region name [None]: eu-west-1
Default output format [None]: Enter
```

Figure 15.2: Configuring AWS CLI.

We were then asked to create an AWS ECS Cluster. If you see the command below in figure 16.2 it creates a cluster with a name of edClusterDemo.

```
clausyd@clausyd-HP-Pavilion-15-Notebook-PC:~$ aws ecs create-cluster --cluster-name edClusterdemo
{
  "cluster": {
    "status": "ACTIVE",
    "statistics": [],
    "tags": [],
    "clusterName": "edClusterdemo",
    "registeredContainerInstancesCount": 0,
    "pendingTasksCount": 0,
    "runningTasksCount": 0,
    "activeServicesCount": 0,
    "clusterArn": "arn:aws:ecs:eu-west-1:828000029458:cluster/edClusterdemo"
  }
}
```

Figure 16.2: Creating Cluster in AWS through CLI.

If you refer to figure 17.2 you will see the cluster successfully created on AWS console.

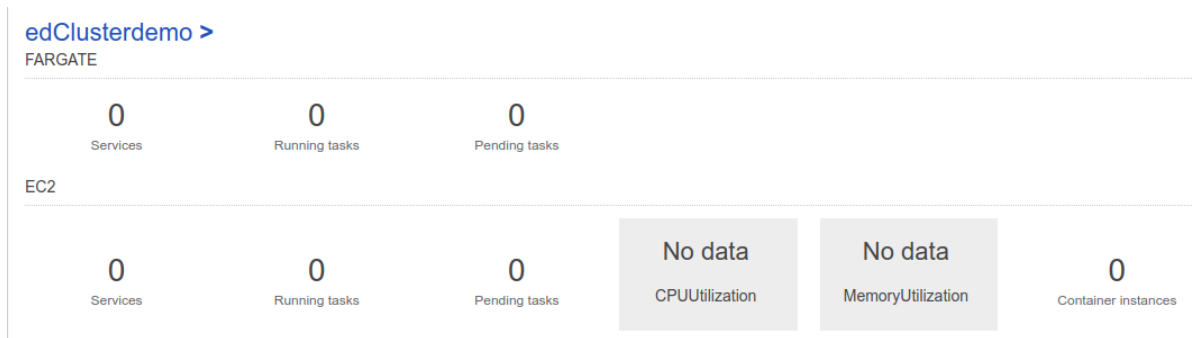


Figure 17.2: Cluster Successfully Created.

For the next step we were asked to launch an ECS instance into my cluster. To do this I found a create tutorial https://docs.aws.amazon.com/AmazonECS/latest/developerguide/launch_container_instance.html it stepped me through each phase. By default, if I launch an ECS instance it will go to the default VPC but in this case I wanted it to be launched into my cluster. This was a matter adding a script that points my instance to my cluster this is done in advanced details on the first config page as seen in figure 18.2.

Once this was finished, I could see the instance running on my cluster. I then needed to create the Docker image for the sample PHP application. If you refer to figure 19.2 you will see the I have the app cloned from Git and I am in the cloned file and am using ls to list files in the directory.

```
clausyd@clausyd-HP-Pavilion-15-Notebook-PC:~/ecs-demo-php-simple-app$ ls
Dockerfile LICENSE NOTICE.md README.md simple-app-task-def.json src
clausyd@clausyd-HP-Pavilion-15-Notebook-PC:~/ecs-demo-php-simple-app$
```

Figure 19.2: Clone App from GIT.

The next task was to run a docker build on the directory. As seen in figure 19.2 you will see the Dockerfile that is needed to build these containers. After that the next step was to create and register a task. The task is on the cluster is what will run the container on the ECS instance. This is a JSON file with a list of configurations. The first step is to register the task with AWS and then you must run the task for it to execute in the instance. If you see figure 20.2 you will see me registering the task definition.

Figure 20.2: Registering Task.

```
root@clausyd-HP-Pavilion-15-Notebook-PC:/home/clausyd/ecs-demo-php-simple-app# aws ecs register-task-definition --cli-input-j igabed
son file://simple-app-task-def.json
{
  "taskDefinition": {
    "status": "ACTIVE",
    "family": "console-sample-app",
    "placementConstraints": [],
    "compatibilities": [
      "EC2"
    ],
    "volumes": [
      {
        "host": {},
        "name": "my-vol"
      }
    ]
  }
}
ip
ubcommand> help
nt --task-definition is required
-Pavilion-15-Notebook-PC:~/task$ aws ecs create-service --cluster fargate-cluster --service-name fargate-service --taskdefinition[]
```

Figure 21.2: Running Task.

If you refer to figure 21.2, I am executing the run task command. This should in theory run the task on the instance and in figure 21.2 below you should be able to see the sample php app in the browser with the public IP of the ECS instance.

3.0 Scaling Container Exercise

The first step in this exercise was the create a cluster on AWS the cli. If you see figure 23.3 you will see that this was successful.

```
clausyd@clausyd-HP-Pavilion-15-Notebook-PC:~$ aws ecs create-cluster --cluster-name ED-fargate-cluster
{
  "cluster": {
    "status": "ACTIVE",
    "statistics": [],
    "tags": [],
    "clusterName": "ED-fargate-cluster",
    "registeredContainerInstancesCount": 0,
    "pendingTasksCount": 0,
    "runningTasksCount": 0,
    "activeServicesCount": 0,
    "clusterArn": "arn:aws:ecs:eu-west-1:828000029458:cluster/ED-fargate-cluster"
  }
}
```

Figure 22.3: Creating Cluster for Fargate.

The next step was to create a service. The command that was giving in the practical wouldn't work. I altered the code to suit what I thought was the correct subnet and security group but still nothing. At this point I went to AWS to configure the service from there. If you refer to figure 2.3 you will see that I managed to create service through AWS.

Service : fargate-service Update Delete

Cluster	ED-fargate-cluster	Desired count	2
Status	ACTIVE	Pending count	0
Task definition	sample-fargate:16	Running count	0
Service type	REPLICA		
Launch type	FARGATE		
Platform version	LATEST(1.2.0)		
Service role	AWSServiceRoleForECS		

Details Tasks Events Auto Scaling Deployments Metrics Tags

Figure 23.3: Cluster Created.

The next step was the define the load balancer. Again, this was a simple as following the steps provided. Refer to figure 24.3 and you can see the load balancer that was created.

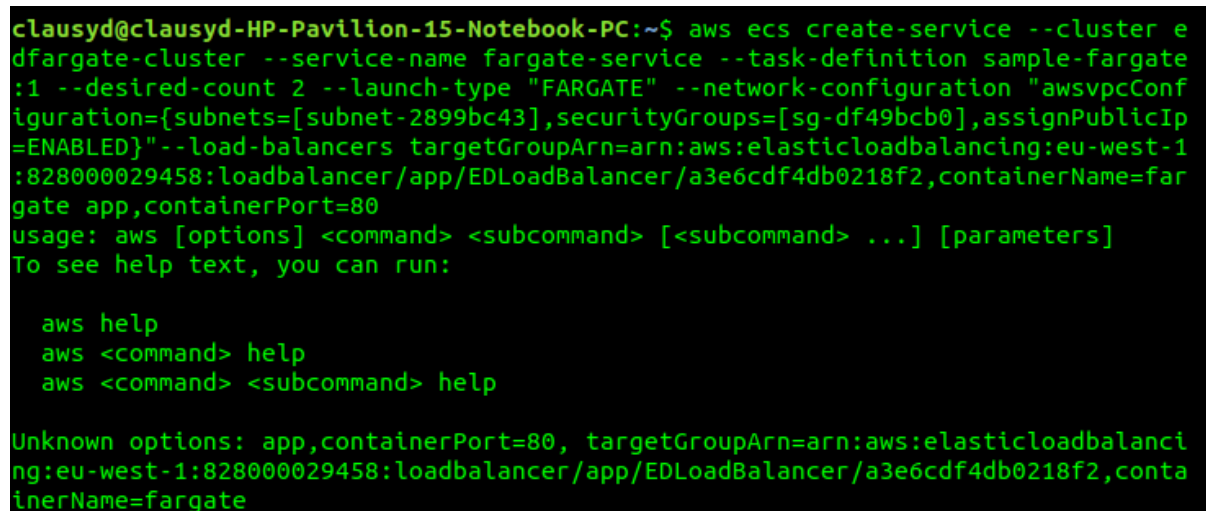
<input type="checkbox"/>	EDLoadBalancer	EDLoadBalancer-20412488...	provisioning	vpc-2999bc42	eu-west-1b, eu-west-1a	application
--------------------------	----------------	----------------------------	--------------	--------------	------------------------	-------------

Figure 24.3: Load Balancer Created.

After this step I got lost. I will show you the command that is was running but I kept getting errors as seen in figure 25.3. I understand what the command is doing and can't see why they are unknow option. I want to create a service on my cluster. I then set up the network configurations by setting

the security group and the subnet I choose which is the default subnet on AWS. Then I try to link my load balancer that I created on AWS to the service I am creating.

```
aws ecs create-service --cluster edfargate-cluster --service-name fargate-service --task-definition
sample-fargate:1 --desired-count 2 --launch-type "FARGATE" --network-configuration
"awsvpcConfiguration={subnets=[subnet-2899bc43],securityGroups=[sg-
df49bcb0],assignPublicIp=ENABLED}" --load-balancers
targetGroupArn=arn:aws:elasticloadbalancing:eu-west-
1:828000029458:loadbalancer/app/EDLoadBalancer/a3e6cdf4db0218f2,containerName=fargate
app,containerPort=80
```



```
clausyd@clausyd-HP-Pavilion-15-Notebook-PC:~$ aws ecs create-service --cluster e
dfargate-cluster --service-name fargate-service --task-definition sample-fargate
:1 --desired-count 2 --launch-type "FARGATE" --network-configuration "awsvpcConf
iguration={subnets=[subnet-2899bc43],securityGroups=[sg-df49bcb0],assignPublicIp
=ENABLED}" --load-balancers targetGroupArn=arn:aws:elasticloadbalancing:eu-west-1
:828000029458:loadbalancer/app/EDLoadBalancer/a3e6cdf4db0218f2,containerName=far
gate app,containerPort=80
usage: aws [options] <command> <subcommand> [<subcommand> ...] [parameters]
To see help text, you can run:

    aws help
    aws <command> help
    aws <command> <subcommand> help

Unknown options: app,containerPort=80, targetGroupArn=arn:aws:elasticloadbalanci
ng:eu-west-1:828000029458:loadbalancer/app/EDLoadBalancer/a3e6cdf4db0218f2,conta
inerName=fargate
```

Figure 25.3: Errors trying to create service.

Because I couldn't get this working I couldn't continue to the end of the exercise. So, in this exercise I got the cluster built. I was able to create a service on AWS but not through the CLI. I build the load balancer in AWS but wasn't able to link it to my cluster.

Conclusion

This was a really interesting practical. I feel it will be really useful to know when I go out in industry. Unfortunately, I could not give this enough time due to circumstances that were out of my control. I was able to get all of the first and second practical working. But the last practical I could not finish I would need more time which I don't have due to other hand-ups that need attention. I feel if I hadn't missed the two practical's then I would have had a better chance at completing all of this report.