

NWEN 243 – 2023 T2

Project 3 – Part B:

Making it Scale - Containers and Serverless Compute

Due: See the **submission system** for authoritative dates and times, however **both** parts **a** and **b** will be submitted together.

FOR SUBMISSION:

- In terms of supporting evidence, it is up to you to take appropriate **screenshots** and make commentary.
 - You must put together a folio of evidence supporting your work in this lab.
 - In your screenshots ensure any IP addresses and/or instance names are visible.
 - Collect your screen shots and organise into a PDF with a narrative explaining what is going on, what each screen shot reveals and how they line up to the sections in this Project.
 - Label this narrative, **Project3partB.pdf**
 - You've seen what we're after, by example from Project 2, you need to provide a similar level of evidence here. This is certainly in the marking criteria. Handing in some working code and the pdf equivalent of crumpled paper won't get you the marks you'd like - there needs to be evidence of introspection and understanding.
 - There are also a couple of questions to answer at the end - you could make some tests or experiments and in general provide an insightful commentary of what is going on.
-

TODO

Up until now we've only looked at a deploying a single instance of your MusicGuru service - remember the *raison d'être* for cloud - is elasticity and scalability, so that is what we will explore in this lab - in particular, horizontal scaling using Containers.

- As we saw in part A, if we want to scale up using VMs to potentially millions of users we'd need many EC2 instances, and we'd need to create, launch, coordinate and load balance them. That's a lot of work. Alternatively,
- we want to keep our service small and boutique? In this case keeping even a small micro instances running is a waste of resources, perhaps even getting rid of the whole server idea and going serverless?

What we need to do is "right-size" our resources and the scale of our program to match our service - what we're doing in this project is to build a container for your MusicGuru server, so that we can create any number of essentially identical duplicate containers and then look at scaling this out.

GETTING STARTED WITH CONTAINERS

There are lots of different ways to start - we can use a standard test Docker image, we could make our own image using docker, or other containerising service. Once we have created the image we need to load it into one of many possible choices of Docker registry, such as Amazon ECR, Artifactory, or Docker's own Registry). There are many paths to success here:

- My plan for this project is to do things as manually as possible, not because this is how its usually done, but because this is a learning exercise and much of the work on the cloud may as well be magic if you don't see what is going on 'under the hood'.
- Where we can - I will keep choices within the AWS ecosystem (plus Docker). I'm not suggesting that this is the best way to do things every time, indeed it is not - this is Docker unplugged! - but for learning the basics, sure why not!

Our first goal will be to Containerise your MusicGuru server.

PLAN

Normally you would install Docker on your development machine. If I got you to do this, it would be a can-o-worms, everyone running every conceivable system would ask me questions I can't answer. So, a better idea! We're going to use the micro-instance from project 1 as our (Docker not Java) development machine!

Raw, command line, unplugged. Hold my Beer!

INSTALLING DOCKER ON YOUR INSTANCE

1. Locate your existing instance or just make another - from your Project 3A AMI. You are an expert now it'll only take a few moments)...

You need to remove the crontab and remove any existing server - or Docker will fail in step 7 because the port will be busy. **reboot** the instance after:

% **sudo crontab -r**

2. Now we'll install Docker on our instance and make this our Docker "**dev**" machine.

(a) Run a general update on your instance: **sudo yum update**

(b) Install docker: **sudo amazon-linux-extras install docker**

(c) Start Docker: **sudo service docker start**

(d) Add the default user (so we don't need to be root and sudo everything): **sudo usermod -a -G docker ec2-user**

(e) Log out of ssh using **exit** (to refresh the permissions we just set)

(f) SSH back in.

(g) Try the following command: **docker info**. You'll see a status dump, and you'll see no containers are hosted. We'll add some soon.

- If that didn't work - then you did something above incorrectly. Go back and check.
- If you get the error: **Cannot connect to the Docker daemon. Is the docker daemon running on this host?**, try rebooting your instance in the EC2 dashboard.

If you can't find the key-pair that you saved for this instance, then I am afraid you can't recover the .pem file so you'll need to make a new instance and install...

MAKING A DOCKER IMAGE

Amazon ECS is able to use Docker images to launch containers on the instances in your clusters. So what we need to do next is use our Docker installation on the AWS instance to make a Docker image from your server.

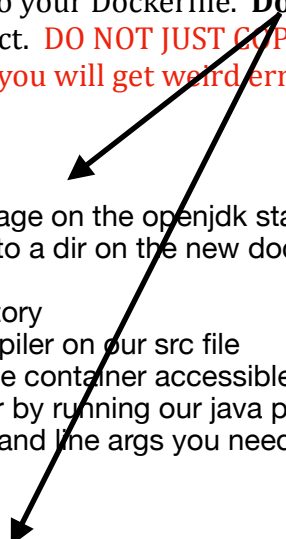
3. If they're not already on your instance, use SCP to copy your source files for your server to your AWS instance. The reason we need to do this is that **we're going to compile the java inside the container**. Believe me it is much less likely to go wrong this way! If you're using Python, then they should already be there.
4. Now, what we need to do now is create a **Dockerfile** - this will be used by docker to construct an image. It contains environmental and build instructions. You need to either:
 - directly create a file containing the following codes, using say cat, or
 - upload it from your machine using scp.
5. You will need to substitute your program's name for mine - see the red parts. The file name is "Dockerfile" with no extension. You will also need to put the port in that your server uses. As usual mine is port 5000, so make that change to your Dockerfile. **Don't copy** the comments over - they're an explanation for this project. **DO NOT JUST COPY AND PASTE - the encoding and hidden characters will break things, you will get weird errors!**

(a) JAVA DOCKERFILE:

```
FROM openjdk:8
COPY *.java /usr/src/MGS/
COPY musicfile /usr/src/MGS/
WORKDIR /usr/src/MGS
RUN javac MusicGuruServer.java
EXPOSE 5000
CMD ["java", "MusicGuruServer", "5000"]
```

// We are basing our image on the openjdk standard image
// copy our local files into a dir on the new docker image

// set the working directory
// execute the java compiler on our src file
// Make port 5000 on the container accessible
// Execute the container by running our java program
// passing in any command line args you need (port)



(b) PYTHON DOCKERFILE:

```
FROM python:latest
COPY *.py /usr/src/MGS/
COPY musicfile /usr/src/MGS/
WORKDIR /usr/src/MGS
EXPOSE 5000
CMD ["python", "MusicGuruServer.py", "5000"]
```

// We are basing our image on the python standard image
// copy our local files into a dir on the new docker image

// set the working directory
// Make port 5000 on the container accessible
// Execute the container by running our java program
// passing in any command line args you need (port)

6. Now we have a Dockerfile, we can get Docker to build the new image - the name of the image needs to be lower case, incidentally mgs = *MusicGuruServer*:

docker build -t mgs .

7. Assuming all is well Docker will now download the base image (remember it is immutable), update (add layers) and environment, and build your image. You might get errors, for example if the name of your source file is wrong. They're mostly self explanatory.

8. Now we can check if Docker built the image properly, with:

docker images --filter reference=mgs

Mine looked like this:

```
[ec2-user@ip-172-31-46-245 ~]$ docker images --filter reference=mgs
REPOSITORY TAG          IMAGE ID            CREATED             SIZE
mgs         latest               1eeacb846db6       3 minutes ago      1.01GB
```

9. Now, we can finally run the newly built image. Recall from the Dockerfile how we exposed port 5000 (or whichever one your application used). What we need to do now, is to map this exposed port on the container to a port on the host machine. In this instance we want to map 5000 to 5000, but we could map 5000 to a different port if we desired. We need to do this mapping so our application can interact with the outside:

docker run -t -i -p 5000:5000 mgs &

[1] 3482 <- you will get this response (different number) which says you have job 1 running in the background and its PID is 3482

10. Use **docker ps** to see if everything is running. I can't emphasise **enough**, any errors in your docker file, including funny characters - **will** cause the execution to fail.

```
[ec2-user@ip-172-31-46-245 ~]$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED    STATUS    PORTS
770b786ce7    mgs      "python MusicGuruS..." 18s ago   Up 16s   0.0.0.0:5000->5000/tcp, :::5000->5000/tcp
```

You can see the port mapping, the command, image and container ID.

11. You should now be able to run your original client and connect to the AWS instance, which will then tunnel the TCP connection to the container and your application. Your client should operate exactly the same as it did in Project 1. Mine looked like this:

```
python3 MusicGuruClient.py 3.89.7.201 5000 0
Specified year out of range (1950-2009), using random date instead: 2004
In 2004 the number 6. song was Neighborhood #1 (Tunnels) - Arcade Fire (172.17.0.2)
```

12. Congrats! You just containerised your first "app"

13. If you want to see a log of prior docker executions, you can use:

docker ps -a

14. **[Don't do this unless you need to]** Incidentally you can also terminate a running docker image with:

docker stop <CONTAINER ID>

CHECKPOINT:

What you have just done is the original way of hosting containers in AWS - were you explicitly used an EC2 instance to host - any number of containers. Each of the containers ports were mapped appropriately to ports on the host instance and the containers were supported and executed using the resources of the EC2 virtual machine.

However, this requires the developer to instantiate and manage EC2 instances or clusters of instances to host their containers, this also means small, ad-hoc containers that implement tiny-services like our server likely underutilise the EC2 instance, which means you're paying for resources you're not using.

PUSHING THE DOCKER IMAGE TO A CONTAINER REGISTRY

So far, we've made a docker image, run it and communicated with it, this is still a bit limited - because we're not taking advantage of the power of the container service. Indeed, we haven't yet told AWS anything about our image! It just looks like any other EC2 instance. What we want to do is put our Docker image into a repository, so that we can point AWS at it and then get AWS to manage, scale and deploy our containers for us. **Cool.**

CREATING YOUR DOCKER ACCOUNT

15. First things, sign up for a Docker account - *use lower case in your name (see below).*
16. Go do this at: hub.docker.com
17. Validate your email

PUSH YOUR IMAGE TO THE DOCKER REGISTRY

To deploy a Docker image on AWS, it needs to be visible in a Docker registry. We can choose to publish our Docker image in any Docker registry we want, as long as AWS can reach it from the internet.

18. Log in to docker from your AWS development instance with:

docker login

WARNING, DOCKER DOES WEIRD THINGS WITH CAPITALS IN YOUR USER NAME. IT LOWER CASES THEM! IF YOU CAN'T LOG IN, IT'S PROBABLY THAT!

19. You will need to enter your docker login credentials (the first time).
20. Use `docker ps` - to double check your joke server is running. We push running container images. (if not start one)
21. When you pull or run an image the name you give it is actually a location (URI) that also refers to the repository host. So you need to create an image tag that references your repo in Docker Hub before you push it to the repository. When we initially ran the image, we actually 'tagged it already' as 'mgs'. This tag isn't enough - we need to tag it with a tag that includes the repository. You can either tag it manually using `docker tag`, or run (in the background) it with the right tag. I'm choosing the first option (check your repository name on docker and use below):

```
docker tag mgs XXXX/mgs
```

22.push your image to the repository, obviously using your account and whatever name you built it with:



```
docker push XXXX/mgs:latest
```

If you don't have an exiting repository with this name "mgs" on Docker hub, the push command will automatically create that repository the first time you push.

23.It should now show up in your DockerHub. Well, after a little while anyway.

Tags

This repository contains 1 tag(s).

| Tag | OS | Type | Pulled | Pushed |
|--|---|-------|--------|---------------|
|  latest |  | Image | --- | 2 minutes ago |

[See all](#)[Go to Advanced Image Management](#)

MAKE A TEST EC2 INSTANCE AND PULL THE CONTAINER.

24. Test your docker image - make a new vanilla EC2 instance and give it a suitable name, like "MGS Test" - so you can distinguish it from your dev instance. We're doing this so it is a clean install and we will know for certain our pushed image is correct. Just make a plain basic standard EC2 instance, remember to use the right Security Group and instance OS type.
25. Use the same keys.
26. You don't need to install java, but you will need the usual prompted updates and install docker (way back in step 2). You also need to start the docker service on the new test instance. It goes without saying - there are no files in the directory...
27. Now pull the docker image you made - you can get this from your Docker repository screen:

```
docker pull docker.io/XXXX/mgs:latest
```
28. Now we run it a little differently, we still need the port map mine is:

```
docker run -t -i -p 5000:5000 XXXX/mgs:latest &
```
29. Now using your client (from before) you will be able to connect to the pulled docker image from docker hub on the new EC2 instance.

DISTRIBUTING YOUR CONTAINER - AWS ECS AND FARGATE.

Now we have a Docker container version of our MusicGuruServer - this is a standardised thing, so we can use other services to control and distribute our service. What we're going to do in the rest of this lab is a Serverless version of the MusicGuruServer - using the Elastic Cluster Service and Fargate.

"AWS Fargate is a technology that you can use with Amazon ECS to run containers without having to manage servers or clusters of Amazon EC2 instances. With Fargate, you no longer have to provision, configure, or scale clusters of virtual machines to run containers. This removes the need to choose server types, decide when to scale your clusters, or optimise cluster packing. "

30. You'll probably have to type "ECS" into the search bar on the AWS console to find the right menus.
31. Make sure you use the "classic Amazon ECS console", i.e. don't turn on the new one - because it is not configured yet for use in AWS Academy.
32. Select Clusters, and then "Get Started"

Getting Started with Amazon Elastic Container Service (Amazon ECS) using Fargate

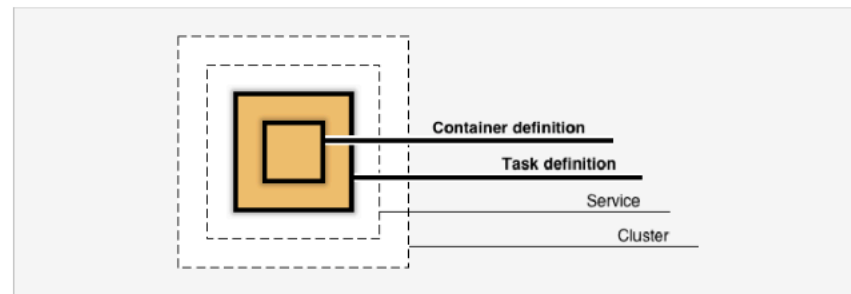
Step 1: Container and Task

Step 2: Service

Step 3: Cluster

Step 4: Review

Diagram of ECS objects and how they relate



Container definition

Edit

Choose an image for your container below to get started quickly or define the container image to use.

sample-app

image : httpd:2.4
memory : 0.5GB (512)
cpu : 0.25 vCPU (256)

nginx

image : nginx:latest
memory : 0.5GB (512)
cpu : 0.25 vCPU (256)

tomcat-webserver

image : tomcat
memory : 2GB (2048)
cpu : 1 vCPU (1024)

custom

image : --
memory : --
cpu : --

Configure



33. Click on configure.

34. fill in the 4 places on the diagram on the next page:

- name your container
- In the "image" box put the location of your Docker image from the Docker repository, something like [docker.io/XXXX/mgs:latest](https://hub.docker.com/r/mgs/mgs-latest/)
- Change the port to your port
- ensure the protocol is TCP
- Click save.

Edit container
✕

▼
Standard

Container name*
MGSECS
i

Image*
i

Private repository authentication*
☐
i

Memory Limits (MiB)

Soft limit
▼
128

+ Add Hard limit

Define hard and/or soft memory limits in MiB for your container. Hard and soft limits correspond to the `memory` and `memoryReservation` parameters, respectively, in task definitions.
ECS recommends 300-500 MiB as a starting point for web applications.

Port mappings

Container port
5000

Protocol
tcp

+ Add port mapping

* Required

Cancel
Update

35. Now, before you click next, you need to edit the task definition.

- You can name the task (optional)
- The only important thing on the task definition is the role. You need to use the drop down to select “Lab Role”. Normally you’d set this up under an IAM rule, but because you have educate/Academy accounts this is not possible.

Task definition details

Task definition name*
first-run-task-definition
i

Network mode*
awsipc
i

Task execution role

Create new role
▼

Create new role

Compatibilities*

LabRole
None

LabRole

DEFINE YOUR SERVICE

36. You can leave the defaults on the Service Definition. You'll notice you could choose a load balancer, the number of tasks and security group, however, the load balancer is an application level balancer and would need HTTP for the health check and the security group that is created is exactly what we need.

CONFIGURING YOUR CLUSTER

37. Change the name, and leave the defaults.

REVIEW THE CLUSTER

38. Check the ports and Role are correct.
39. Select create. This will take a while....
40. ...but will eventually look like this:

Getting Started with Amazon Elastic Container Service (Amazon ECS) using Fargate

Launch Status

We are creating resources for your service. This may take up to 10 minutes. When we're complete, you can view your service.

[Back](#)[View service](#)

Additional features that you can add to your service after creation

Scale based on metrics

You can configure scaling rules based on CloudWatch metrics

Preparing service : 9 of 9 complete

| | | |
|---|----------|---|
| ECS resource creation | complete | ✓ |
| Cluster MGSCluster | complete | ✓ |
| Task definition first-run-task-definition:1 | complete | ✓ |
| Service MGSECS-service | complete | ✓ |
| Additional AWS service integrations | complete | ✓ |
| Log group /ecs/first-run-task-definition | complete | ✓ |
| CloudFormation stack EC2ContainerService-MGSCluster | complete | ✓ |
| VPC vpc-052b513aba30134c3 | complete | ✓ |
| Subnet 1 subnet-085a6394fd8372c8 | complete | ✓ |
| Subnet 2 subnet-03b02fa5389b0eb34 | complete | ✓ |
| Security group sg-02fa2d1bc90a995cb | complete | ✓ |

41. You can now click on view service.
42. And eventually your task will be provisioned, this can take a bit of extra time.
43. It is worth noting, if you have your repository wrong, you'll just notice the tasks tab, there may be nothing with a running status. You should then click on the stopped button, and if you see a lot of containers start and terminate with a "pull error" - thats your problem.
44. Assuming you have 1 running service, then you can test your service using your original client from anywhere - university, home, the bus.

Cluster : MGS

Update Cluster

Delete Cluster

Get a detailed view of the resources on your cluster.

Cluster ARN arn:aws:ecs:us-east-1:556874762784:cluster/MGS

Status **ACTIVE**

Registered container instances 0

Pending tasks count 0 Fargate, 0 EC2, 0 External

Running tasks count 1 Fargate, 0 EC2, 0 External

Active service count 1 Fargate, 0 EC2, 0 External

Draining service count 0 Fargate, 0 EC2, 0 External

Services Tasks ECS Instances Metrics Scheduled Tasks Tags Capacity Providers

Create Update Delete Actions Last updated on September 24, 2023 2:59:50 PM (0m ago)

Filter in this page Launch type ALL Service type ALL < 1-1 >

| | Service Name | Status ... | Service... | Task D... | Desire... | Runni... | Launc... | Platfor... |
|--------------------------|--------------|------------|------------|-----------|-----------|----------|----------|------------|
| <input type="checkbox"/> | MGS-service | ACTIVE | REPLICA | MGS:1 | 1 | 1 | FARGATE | LATES... |

45. What you need to do now is to find the IP address for this 'serverless' service. We're going to have to do some digging.

46. Click the tasks tab and then the service.

Services **Tasks** ECS Instances Metrics Scheduled Tasks Tags Capacity Providers

Run new Task Stop Stop All Actions Last updated on September 24, 2023 3:07:06 PM (0m ago)

Desired task status: **Running** Stopped

Filter in this page Launch type ALL < 1-1 > Page size 50

| | Task | Task de... | Contain... | Last stat... | Desired ... | Started ... | Started ... | Group | Launch ... | Platform... |
|--------------------------|---------------------------|------------|------------|----------------|-------------|-------------|-------------|-------------|------------|-------------|
| <input type="checkbox"/> | aca454... | MGS:1 | -- | RUNNING | RUNNING | 2023-09-... | ecs-svc/... | service:... | FARGATE | 1.4.0 |

aca454a50eb64c9dbbb231375e5a3321

47. Then look for the line under network and copy the public IP - use it on your client.

Network

Network mode awsvpc

ENI Id [eni-0b8bb12e9d4271d1f](#)

Subnet Id subnet-0d916a6eb1906240f

Private IP 10.0.1.39

Public IP [3.91.181.140](#)

Mac address 12:ee:63:de:82:e1

48. On your local machine run your client with this IP:

```
~$python3 MusicGuruClient.py 3.91.181.140 5000 0
```

Specified year out of range (1950-2009), using random date instead: 1973

In 1973 the number 6. song was Money - Pink Floyd (10.0.1.39)

```
~$python3 MusicGuruClient.py 3.91.181.140 5000 0
```

Specified year out of range (1950-2009), using random date instead: 1981

In 1981 the number 2. song was Don't You Want Me? - Human League (10.0.1.39)

49. Assuming everything has been configured correctly, Congratulations! You've created a docker container image, put it in a repository and then had AWS provision a server-less FARGATE service for our MusicGuru server! All in the bare metal, largely with a CLI!

REPLICAS AND QUESTIONS

50. Your final task in this lab is to go back and create another ECS fargate cluster - but this time choose 2 or 3 tasks. Everything else should stay the same - except choose different names ok.

51. **Question 1:** What is unique about each replica - no **not** the ID. Although that **is** unique.

52. **Question 2:** How might we manage clients talking to replicas better?