Cloud Computing: Containerisation

Dimitra Zuccarelli, 20072495

November 2018

Contents

1 Introduction

Containerisation and microservices have completely revolutionised the face of application architecture in the space of the last few years [1]. Containers can be thought of small virtual machines that don't have the overhead of the operating system. They are single-purpose, lightweight and platform independent which make them a suitable starting point for microservice architectures [2]. Below is a good definition of what containers are, taken from the Docker documentation.

```
A container is a standard unit of software that packages up code and all its
    dependencies so the application runs quickly and reliably from one computing
    environment to another.
(Docker)
```

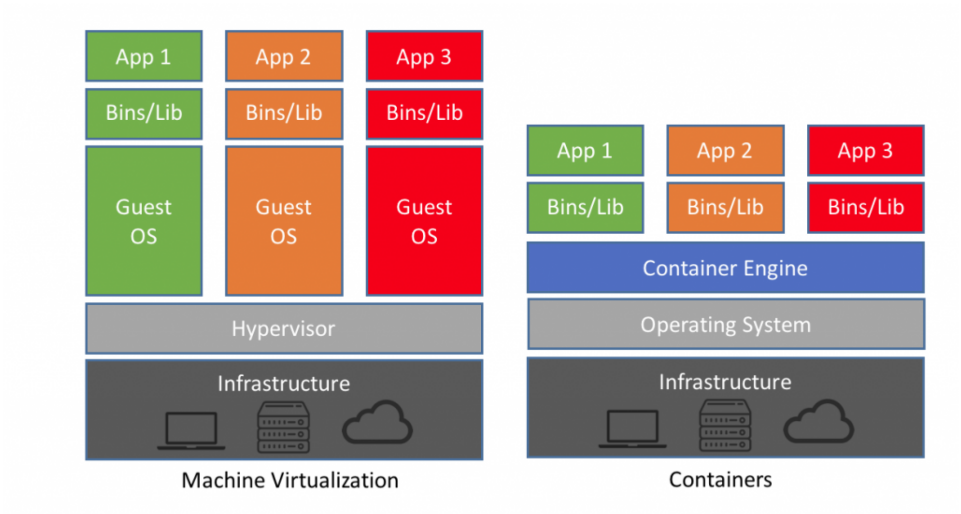


Figure 1: Virtual Machines vs Containers

This report will attempt to outline firstly the process of containerising applications, by building Docker images for RabbitMQ simple producers and consumers and deploying them to Dockerhub. Secondly, it will attempt to show how containers and containerised applications can be managed in the cloud, using a container orchestration platform by AWS called Elastic Container Service (ECS).

2 Containerising RabbitMQ

The aim of this exercise is to Dockerise a simple Go RabbitMQ producer and consumer.

2.1 Containerising the consumer

The first Docker image that will be created is for the RabbitMQ consumer. The code for the consumer is the following:

Listing 1: RabbitMQ consumer code to be dockerised

```go
func failOnError(err error, msg string) {
   if err != nil {
      log.Fatalf("%s: %s", msg, err)
   }
}

func getEnv(key, fallback string) string {
   if value, ok := os.LookupEnv(key); ok {
      return value
   }
   return fallback
}

func main() {
   broker := getEnv("BROKER", "localhost")
   queue := getEnv("QUEUE", "hello")

   conn, err := amqp.Dial(fmt.Sprintf("amqp://guest:guest@%s:5672/", broker))
   failOnError(err, "Failed to connect to RabbitMQ")
   defer conn.Close()

   ch, err := conn.Channel()
   failOnError(err, "Failed to open a channel")
   defer ch.Close()

   q, err := ch.QueueDeclare(
      queue, // name
      false, // durable
      false, // delete when usused
      false, // exclusive
      false, // no-wait
      nil,   // arguments
   )
   failOnError(err, "Failed to create queue")

   msgs, err := ch.Consume(
      q.Name, // queue
      "",     // consumer
      true,   // auto-ack
      false,  // exclusive
      false,  // no-local
      false,  // no-wait
      nil,    // args
   )
   failOnError(err, "Failed to register consumer")

   forever := make(chan bool)
```

```
    go func() {
        for msg := range msgs {
            log.Printf("Received message: %s", msg.Body)
        }
    }()

    log.Printf(" [*] Waiting for messages. To exit press CTRL+C")
    <-forever
}
```

The program contains a helper function `getEnv(key, fallback string)`, which takes in an environment variable name and returns its value if it exists, or a default string if no value has been defined. This is necessary to be able to pass in environment variables to the container that the program can use. There are two environment variables in the consumer:

- `BROKER`: This is the address of the broker, which defaults to "localhost" if the environment variable is not passed in
- `QUEUE`: This is the name of the queue that the consumer will read from, which defaults to "hello" if the environment variable is not passed in

The first iteration of the Dockerfile looks like this:

Listing 2: RabbitMQ Consumer Dockerfile

```
FROM golang:1.10.2

ENV PROJECT_PATH $GOPATH/src/github.com/dimitraz/go-rabbitmq

# Copy the consumer file
COPY . $PROJECT_PATH

# Change directory
WORKDIR $PROJECT_PATH

# Pull down amqp and build the executable
RUN go get github.com/streadway/amqp && go install && go build -o consumer .

# Run the app
CMD ["./consumer"]
```

The steps for the Dockerfile are the following:

1. Firstly, the base image that will be used to build on top of is specified. This is a publicly available official Golang Docker image for Go 1.10.2.
2. Next, the `PROJECT_PATH` environment variable is set. This is simply to define where the consumer code will be copied to in the container. This needs to be set because all Go projects can only be compiled from within the `$GOPATH`. The convention for the project path is `$GOPATH/src/github.com/<user>/<repo-nam` if the project is being hosted on Github.
3. The consumer code (`consumer.go`) is copied from the local machine to the project path within the container.
4. The project dependencies (`amqp`) are installed using `go get`.
5. Finally, the code is compiled into an executable file called `consumer`, and the `CMD` instruction is used to run it.

The commands used to build the image and run the consumer container are:

```
# Create a new docker network
docker network create test

# Start the RabbitMQ broker, with host name "my-rabbit"
docker run -d -P --network test --hostname my-rabbit --name some-rabbit rabbitmq:3

# Build the image
docker build -t dimitraz/rabbitmq-consumer:1.0 .

# Run the consumer, passing in the name of the broker as an environment variable
docker run -e BROKER=some-rabbit --network test dimitraz/rabbitmq-consumer:1.0
```

2.2 Containerising the producer

The process for containerising the producer is identical to the consumer. The code for the producer looks like this:

```
func main() {
   broker := getEnv("BROKER", "localhost")
   queue := getEnv("QUEUE", "hello")

   conn, err := amqp.Dial(fmt.Sprintf("amqp://guest:guest@%s:5672/", broker))
   failOnError(err, "Failed to connect to RabbitMQ")
   defer conn.Close()

   ch, err := conn.Channel()
   failOnError(err, "Failed to open a channel")
   defer ch.Close()

   q, err := ch.QueueDeclare(
      queue, // name
      false, // durable
      false, // delete when unused
      false, // exclusive
      false, // no-wait
      nil,   // arguments
   )
   failOnError(err, "Failed to declare a queue")

   body := "Hello World!"
   err = ch.Publish(
      "",     // exchange
      q.Name, // routing key
      false, // mandatory
      false, // immediate
      amqp.Publishing{
         ContentType: "text/plain",
         Body:        []byte(body),
      })
   log.Printf("Message: '%s' sent to Queue: %s", body, q.Name)
   failOnError(err, "Failed to deliver message")
```

The Dockerfile is similar, with the main difference being that the executable is now called `producer`:

```
FROM golang:1.10.2

ENV PROJECT_PATH $GOPATH/src/github.com/dimitraz/go-rabbitmq

# Copy the consumer file
COPY . $PROJECT_PATH

# Change directory
WORKDIR $PROJECT_PATH

# Pull down amqp and build the executable
RUN go get github.com/streadway/amqp && go install && go build -o producer .

# Run the app
CMD ["./producer"]
```

And the commands for running the container in the same network as before:

```
# Build the image
docker build -t dimitraz/rabbitmq-producer:1.0 .

# Run the consumer, passing in the name of the broker as an environment variable
docker run -e BROKER=some-rabbit --network test dimitraz/rabbitmq-producer:1.0
```

2.3 A second iteration

A more widely accepted way of containerising Go programs is to build the Go binaries beforehand, copying them into the container afterwards. This reduces the size of the container as well as any potential security risks. Typically the base image used for this is the `scratch` image, which is an empty image used when creating base images or very minimal Docker images, like the one below.

A `Makefile` was created to automate tasks such as building the binary, building the image and pushing the image to Dockerhub:

```
DOCKER_ORG=dimitraz
IMAGE_NAME=rabbitmq-producer
IMAGE_TAG=1.1
BINARY_NAME=producer
GOOS=linux

.phony: docker_release
docker_release: docker_build docker_push

.phony: docker_push
docker_push:
   docker push $(DOCKER_ORG)/$(IMAGE_NAME):$(IMAGE_TAG)

.phony: docker_build
docker_build: build_binary
   docker build -t $(DOCKER_ORG)/$(IMAGE_NAME):$(IMAGE_TAG) .

.phony: build_binary
build_binary:
   GOOS=$(GOOS) go build -o $(BINARY_NAME) main.go
```

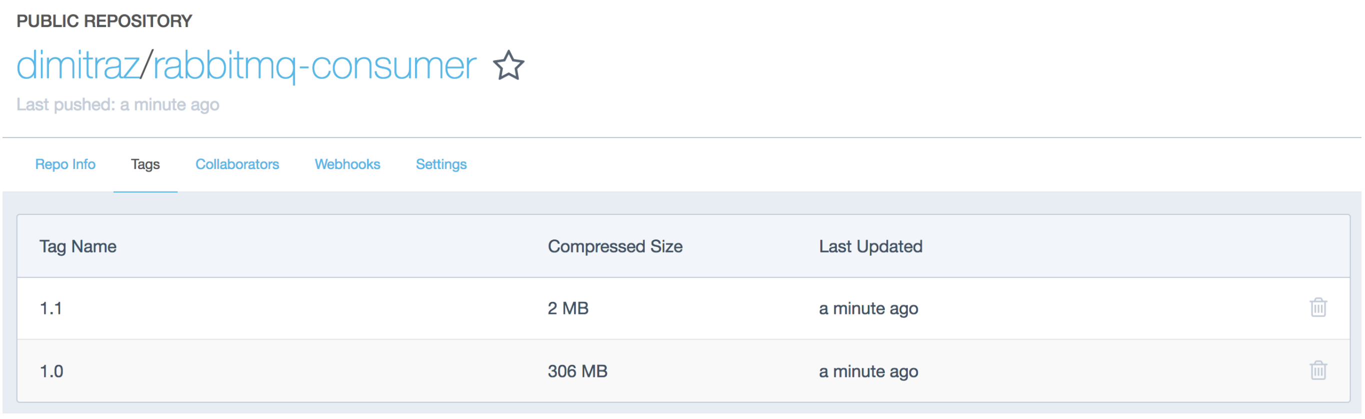
The new Dockerfile is extremely simple:

```
FROM scratch

ADD producer /producer

CMD ["/producer"]
```

The steps for the new Dockerfile are:

1. Specify the base image, which is the empty `scratch` image
2. Assuming that the Go binary has been built in the same directory, copy this over to the container
3. Run the binary.

The `make docker_build` command from the Makefile was used to build the binary and the Docker image. To do all of this and then push the image to Dockerhub, the `make docker_release` can be used.

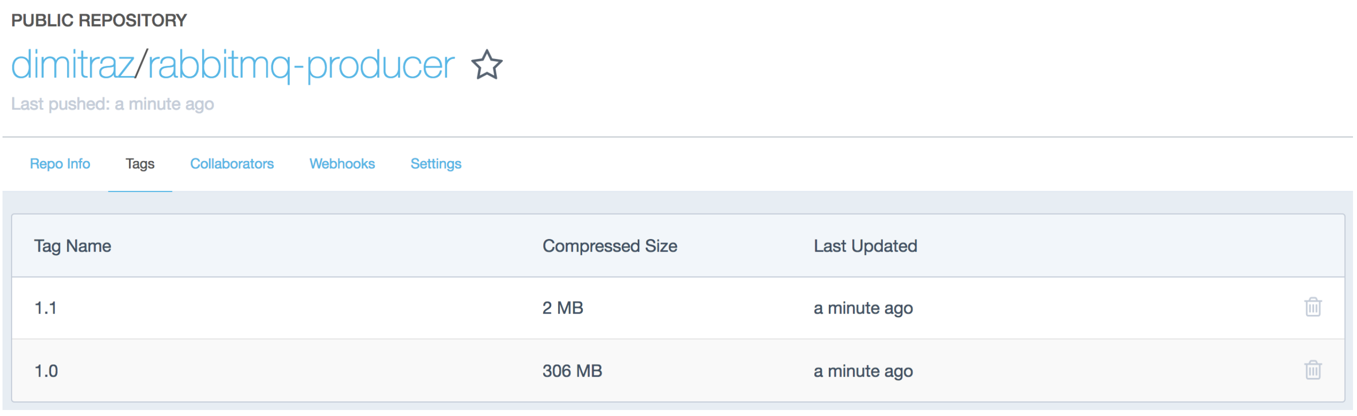

PUBLIC REPOSITORY

dimitraz/rabbitmq-consumer

Last pushed: a minute ago

Tag Name	Compressed Size	Last Updated
1.1	2 MB	a minute ago
1.0	306 MB	a minute ago

Figure 2: Consumer Image on Dockerhub


PUBLIC REPOSITORY

dimitraz/rabbitmq-producer

Last pushed: a minute ago

Tag Name	Compressed Size	Last Updated
1.1	2 MB	a minute ago
1.0	306 MB	a minute ago

Figure 3: Producer Image on Dockerhub

The Dockerfile, Makefile and code can be found here for the consumer and here for the producer.

3 Serverless Container Management with AWS ECS and AWS Fargate

Despite the benefits that containers may offer, containerised applications may become difficult to manage and maintain as they grow. Amazon Elastic Container Service (Amazon ECS) is a container orchestration service developed by AWS to manage the complexities of these applications. In combination with AWS Fargate, applications can be deployed to ECS without the need of any server management at all. A few advantages of using the Fargate launch type over the EC2 launch type (seen in the first lab) are:

- When using the EC2 launch type, your cluster contains a group of container instances that you have to manage. When using the Fargate launch type with tasks within your cluster, Amazon ECS manages your cluster resources for you. With Fargate you do not have to worry about underlying structure or maintenance (hence the term 'Serverless').
- With EC2 launch types, port mappings are more difficult. Because multiple containers could be listening on the same port externally (for example, two web servers could both be listening on port 80), there would be a collision that would need to be addressed if both were to be scheduled on the same instance. With Fargate dynamic port mappings this is managed for you.

The next section of the report will aim to illustrate the steps in deploying and scaling a containerised application to AWS using Fargate.

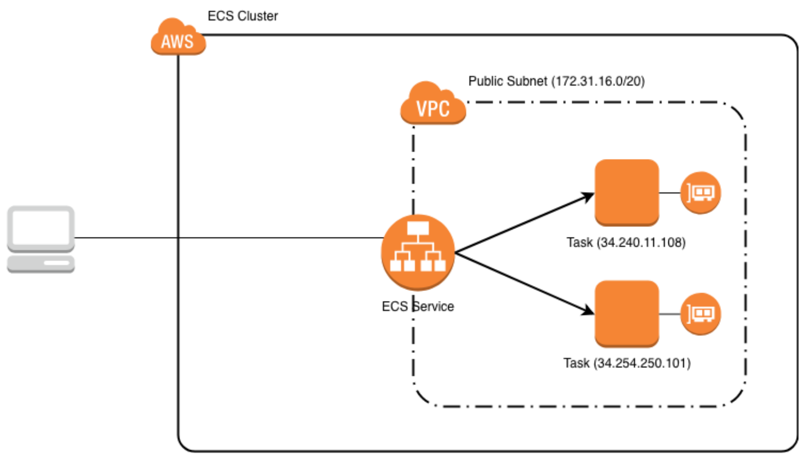


Figure 4: ECS Fargate Topology

3.1 Task Definition

A `Task Definition` in ECS is a JSON file containing the configuration necessary (essentially a "blueprint") for running the application. This file will usually include the name of the container image, the ports that need to be exposed, the memory requirements, any environment variables etc. For this assignment a task definition was provided for running an Apache web server:

```
{
    "family": "dz-sample-fargate",
    "networkMode": "awsvpc",
    "containerDefinitions": [
        {
            "name": "dz-fargate-app",
            "image": "httpd:2.4",
            "portMappings": [
                {
                    "containerPort": 80,
                    "hostPort": 80,
                    "protocol": "tcp"
                }
            ],
            "essential": true,
            "entryPoint": [
                "sh",
      "-c"
            ],
            "command": [
                "/bin/sh -c \"echo '<html> <head> <title>Amazon ECS Sample App</title>
                    <style>body {margin-top: 40px; background-color: #333;} </style>
                    </head><body> <div style=color:white;text-align:center> <h1>Amazon
                    ECS Sample App</h1> <h2>Congratulations!</h2> <p>Your application
                    is now running on a container in Amazon ECS.</p><p>hostname is:
                    $(hostname)</p> </div></body></html>' >
                    /usr/local/apache2/htdocs/index.html && httpd-foreground\""
            ]
        }
    ],
    "requiresCompatibilities": [
        "FARGATE"
    ],
    "cpu": "256",
    "memory": "512"
}
```

This task definition:

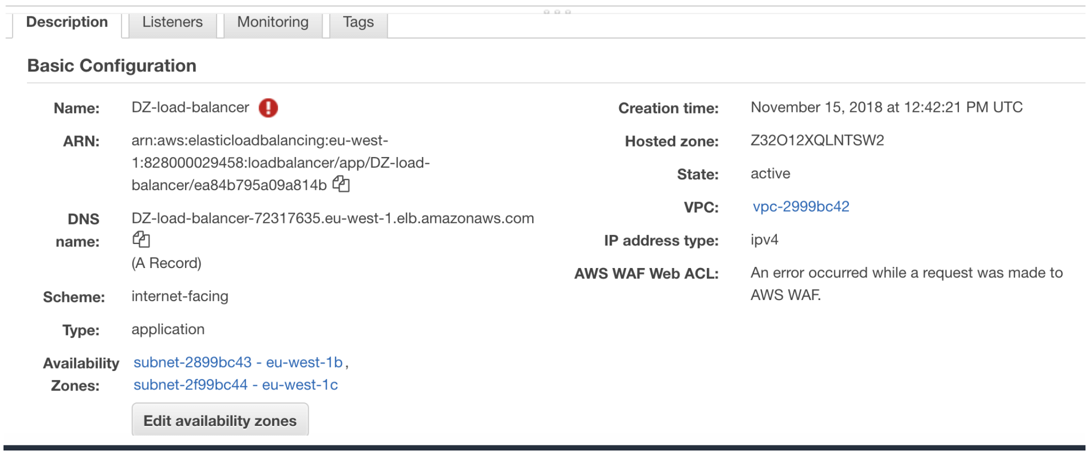
- Uses the Fargate launch type
- Is based on the `httpd:2.4` Docker image
- Needs 512 MB Task memory and 256 (.25 vCPU) task CPU
- Listens on port 80 both internally and externally
- Uses the `awsvpc` network mode, which provides each task with an elastic network interface
- Additionally, the `hostname` is printed in the HTML, which will be used to show the load balancing across tasks later on.

3.2 Creating a load balancer

Before this application can be deployed to ECS, a few other components must first be created:

- A **Service**, which will be able to run a given amount of tasks, and make sure these are up and running at all times. Using a service means fault tolerance and availability is handled for us.
- A **Load Balancer**, which will allow the service to load balance across the tasks in the service's target group. This once again allows for fault tolerance and reliability, because any tasks that go down (therefore failing the health checks) will not be load balanced across. It also means that the application can be scaled very easily, by simply adding new tasks to the target group. The load balancer will be created first, and the service will be created in the next section.

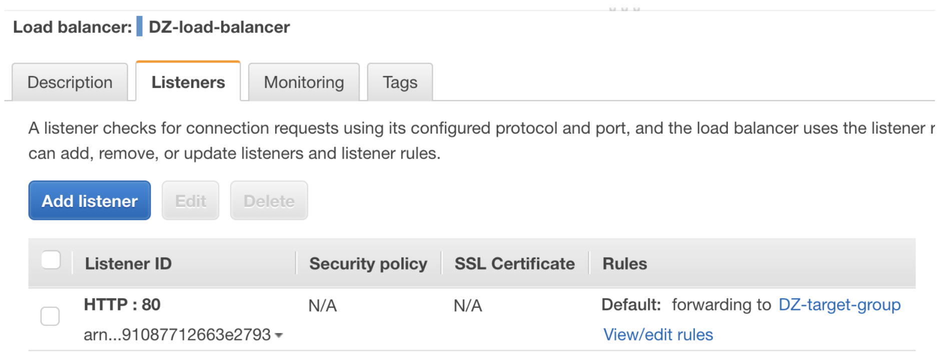
A load balancer, listener and target group were created following the steps from the lab with the following configuration:



Basic Configuration

Name:	DZ-load-balancer	Creation time:	November 15, 2018 at 12:42:21 PM UTC
ARN:	arn:aws:elasticloadbalancing:eu-west-1:828000029458:loadbalancer/app/DZ-load-balancer/ea84b795a09a814b	Hosted zone:	Z32O12XQLNTSW2
DNS name:	DZ-load-balancer-72317635.eu-west-1.elb.amazonaws.com (A Record)	State:	active
Scheme:	internet-facing	VPC:	vpc-2999bc42
Type:	application	IP address type:	ipv4
Availability Zones:	subnet-2899bc43 - eu-west-1b, subnet-2f99bc44 - eu-west-1c	AWS WAF Web ACL:	An error occurred while a request was made to AWS WAF.

Figure 5: Load Balancer Configuration

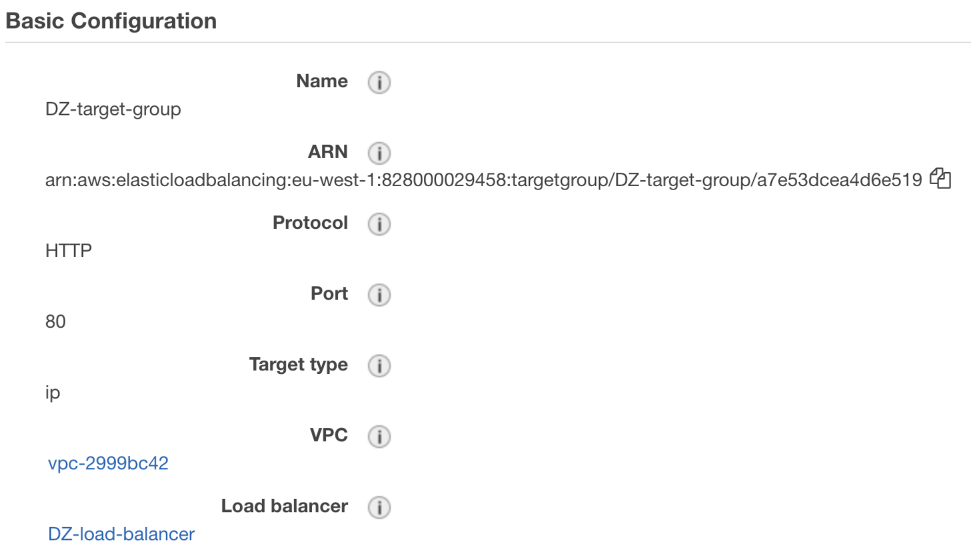


Load balancer: DZ-load-balancer

A listener checks for connection requests using its configured protocol and port, and the load balancer uses the listener r
can add, remove, or update listeners and listener rules.

Listener ID	Security policy	SSL Certificate	Rules
HTTP : 80 arn...91087712663e2793	N/A	N/A	Default: forwarding to DZ-target-group View/edit rules

Figure 6: Load Balancer Listener Configuration



Basic Configuration

Name	DZ-target-group
ARN	arn:aws:elasticloadbalancing:eu-west-1:828000029458:targetgroup/DZ-target-group/a7e53dcea4d6e519
Protocol	HTTP
Port	80
Target type	ip
VPC	vpc-2999bc42
Load balancer	DZ-load-balancer

Figure 7: Target Group Configuration

No targets need to be specified for the moment. This is because ECS handles this for us, automatically registering and deregistering containers within the target group as needed.

3.3 Creating a service

Once the load balancer and target groups are created, they can be passed to an ECS service, which will be able to load balance across the containers in the target group if everything is correctly set up.

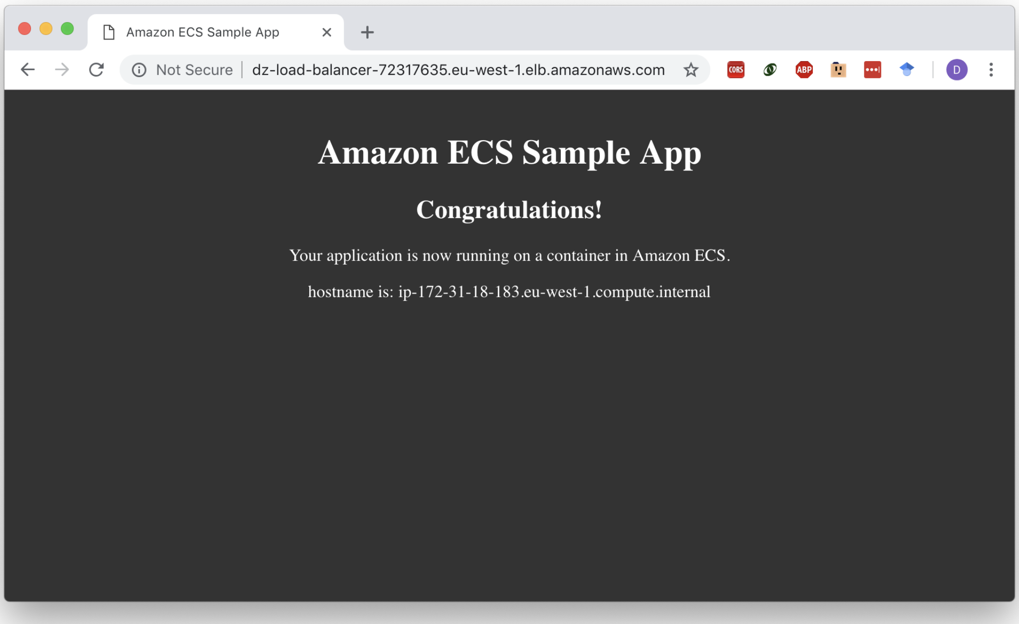
The following command was used to create a new service (`DZ-fargate-service`) associated with the ECS cluster (`DZ-fargate-cluster`):

```
aws ecs create-service
--cluster DZ-fargate-cluster
--service-name DZ-fargate-service
--task-definition dz-sample-fargate:1
--desired-count 2
--launch-type "FARGATE"
--network-configuration "awsvpcConfiguration={subnets=[subnet-2899bc43],
    securityGroups=[sg-0cb3c160887d875d0],
    assignPublicIp=ENABLED
}"
--load-balancers targetGroupArn=arn:aws:elasticloadbalancing:eu-west-1:828000029458:
targetgroup/DZ-target-group/a7e53dcea4d6e519,containerName=fargate-app,containerPort=80
```

This command

- Specifies a desired count of 2 containers running at all times
- Specifies the VPC subnet to use (the default *public* subnet is used in this case), the security group, and ensures a public IP address is assigned to the tasks' elastic network interface. This is needed to give the task a route to the internet to pull down container images.
- Tells the service to use the load balancer and target group created earlier. Specifying the target group here will allow ECS to deploy new tasks to this group automatically.

Following the link to the load balancer's DNS name shows the service load balancing across the tasks in action:

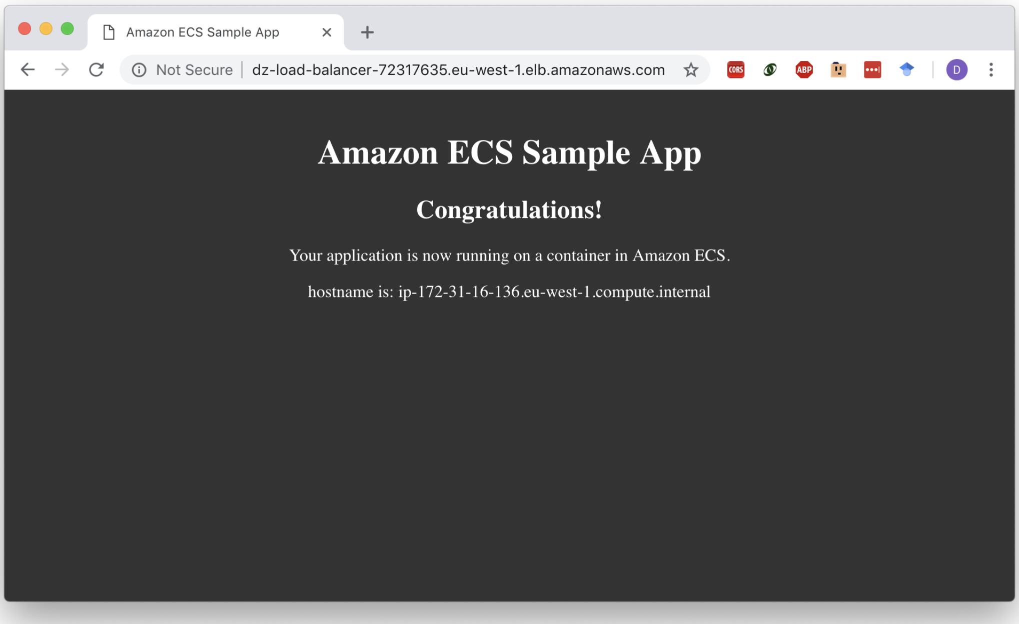


Amazon ECS Sample App

Congratulations!

Your application is now running on a container in Amazon ECS.

hostname is: ip-172-31-18-183.eu-west-1.compute.internal



Amazon ECS Sample App

Congratulations!

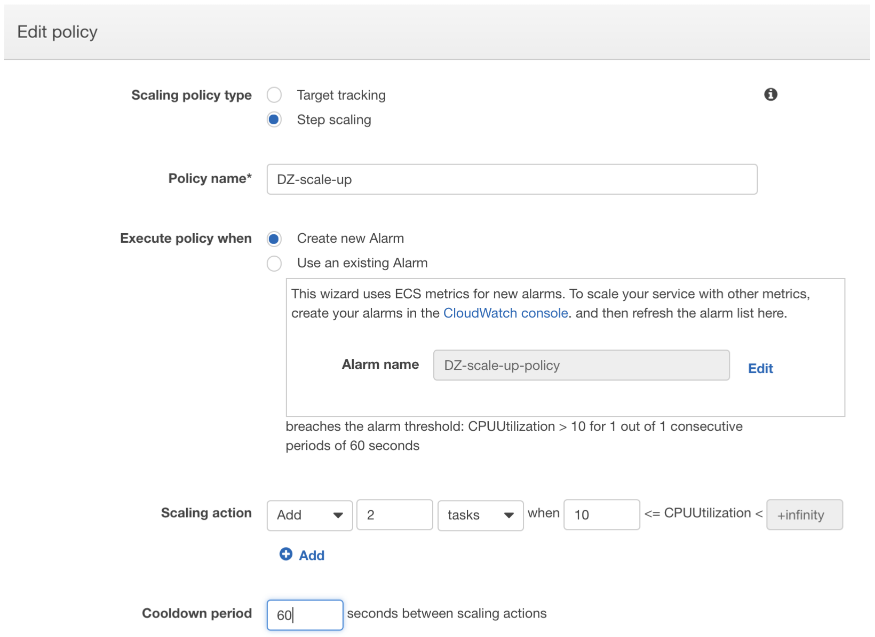
Your application is now running on a container in Amazon ECS.

hostname is: ip-172-31-16-136.eu-west-1.compute.internal

Figure 8: ECS Service and Fargate tasks

3.4 Autoscaling Policies

Autoscaling policies can be created to manage how tasks get scaled up and down in response to peaks and troughs in traffic and CPU/memory usage. In this case `Step Scaling` was used, based on CPU usage:


Edit policy

Scaling policy type: Target tracking [✓ Step scaling]

Policy name*: DZ-scale-up

Execute policy when: [✓ Create new Alarm] Use an existing Alarm

This wizard uses ECS metrics for new alarms. To scale your service with other metrics, create your alarms in the CloudWatch console. and then refresh the alarm list here.

Alarm name: DZ-scale-up-policy Edit

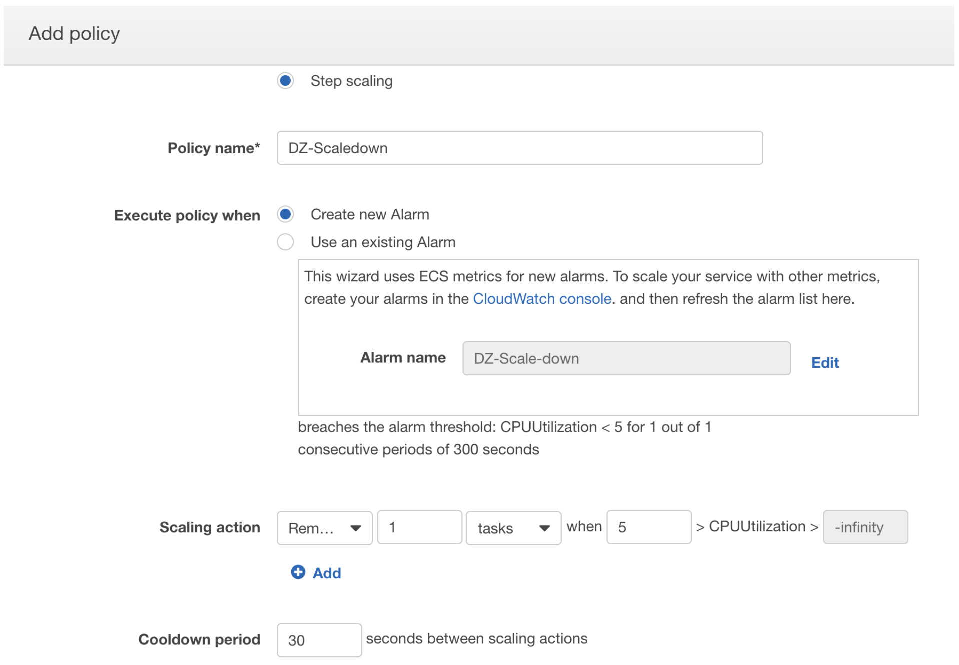
breaches the alarm threshold: CPUUtilization > 10 for 1 out of 1 consecutive periods of 60 seconds

Scaling action: Add 2 tasks when 10 <= CPUUtilization < +infinity

Add

Cooldown period: 60 seconds between scaling actions

Figure 9: Scale up by 2 tasks when CPU utilisation exceeds 10


Add policy

[✓ Step scaling]

Policy name*: DZ-Scaledown

Execute policy when: [✓ Create new Alarm] Use an existing Alarm

This wizard uses ECS metrics for new alarms. To scale your service with other metrics, create your alarms in the CloudWatch console. and then refresh the alarm list here.

Alarm name: DZ-Scale-down Edit

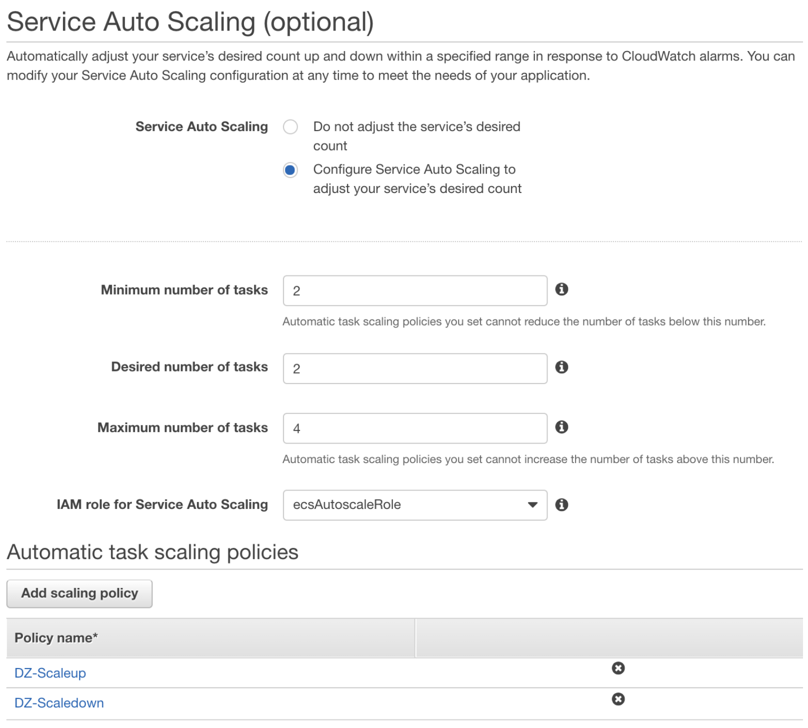
breaches the alarm threshold: CPUUtilization < 5 for 1 out of 1 consecutive periods of 300 seconds

Scaling action: Rem… 1 tasks when 5 > CPUUtilization > -infinity

Add

Cooldown period: 30 seconds between scaling actions

Figure 10: Scale down 1 task when CPU utilisation is less than 5



Service Auto Scaling (optional)

Automatically adjust your service's desired count up and down within a specified range in response to CloudWatch alarms. You can modify your Service Auto Scaling configuration at any time to meet the needs of your application.

Service Auto Scaling: ○ Do not adjust the service's desired count [✓ Configure Service Auto Scaling to adjust your service's desired count]

Minimum number of tasks	2
	Automatic task scaling policies you set cannot reduce the number of tasks below this number.
Desired number of tasks	2
Maximum number of tasks	4
	Automatic task scaling policies you set cannot increase the number of tasks above this number.
IAM role for Service Auto Scaling	ecsAutoscaleRole

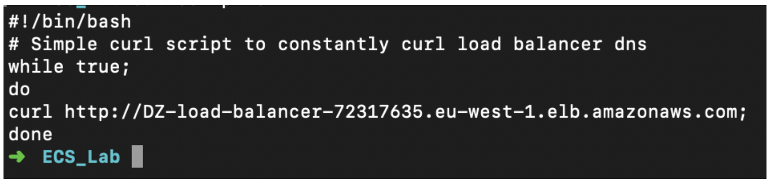
Automatic task scaling policies

Add scaling policy

Policy name*	
DZ-Scaleup	
DZ-Scaledown	

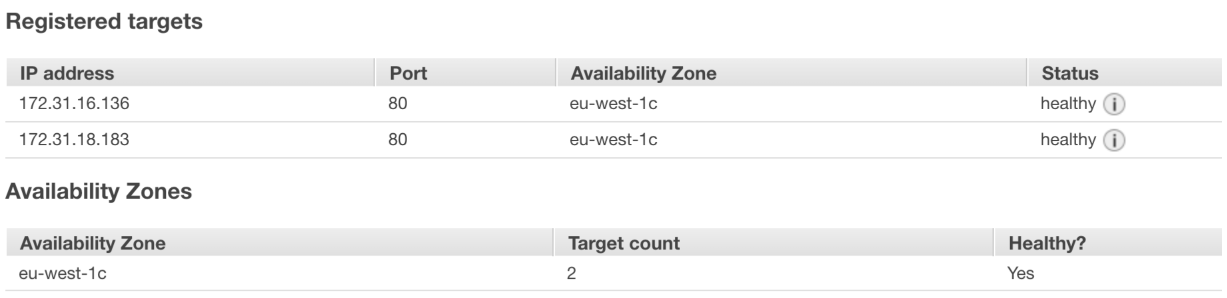
Figure 11: Autoscaling Configuration

These are of course unrealistic policies, but make testing the scaling a much easier task. To trigger a scale up the following script was executed, which constantly sends CURL requests to the load balancer DNS:



```
#!/bin/bash
# Simple curl script to constantly curl load balancer dns
while true;
do
curl http://DZ-load-balancer-72317635.eu-west-1.elb.amazonaws.com;
done
➜  ECS_Lab
```

Figure 12: Load Balancer Listener Configuration



Registered targets

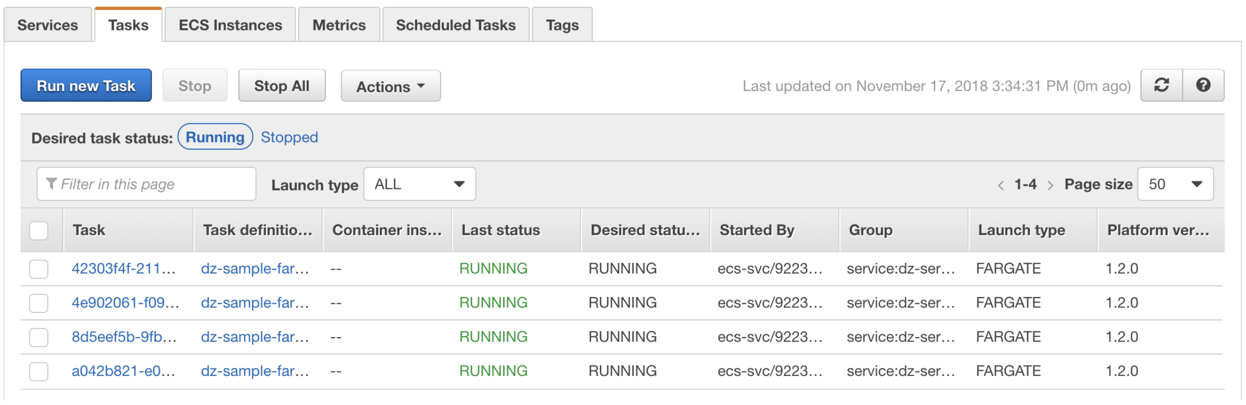
IP address	Port	Availability Zone	Status
172.31.16.136	80	eu-west-1c	healthy
172.31.18.183	80	eu-west-1c	healthy

Availability Zones

Availability Zone	Target count	Healthy?
eu-west-1c	2	Yes

Figure 13: Target group targets before scaling up

The autoscaling policy is successfully triggered, and 2 new tasks are spun up:



Last updated on November 17, 2018 3:34:31 PM (0m ago)

Desired task status: Running Stopped

Filter in this page | Launch type ALL | 1-4 | Page size 50

Task	Task definitio…	Container ins…	Last status	Desired statu…	Started By	Group	Launch type	Platform ver…
42303f4f-211…	dz-sample-far…	--	RUNNING	RUNNING	ecs-svc/9223…	service:dz-ser…	FARGATE	1.2.0
4e902061-f09…	dz-sample-far…	--	RUNNING	RUNNING	ecs-svc/9223…	service:dz-ser…	FARGATE	1.2.0
8d5eef5b-9fb…	dz-sample-far…	--	RUNNING	RUNNING	ecs-svc/9223…	service:dz-ser…	FARGATE	1.2.0
a042b821-e0…	dz-sample-far…	--	RUNNING	RUNNING	ecs-svc/9223…	service:dz-ser…	FARGATE	1.2.0

Figure 14: Two new tasks are triggered

4 Conclusion

While this practical nicely illustrates shows how to deploy containerised applications to ECS, there are several improvements/considerations that should be made before it is ready for production. A few of these considerations are listed below.

- Tasks should not be publicly accessible. A more appropriate solution would be to deploy them into private subnets, letting them get out onto the internet (for example, to pull the Docker images) via a NAT gateway. This topology might look something like the one below.

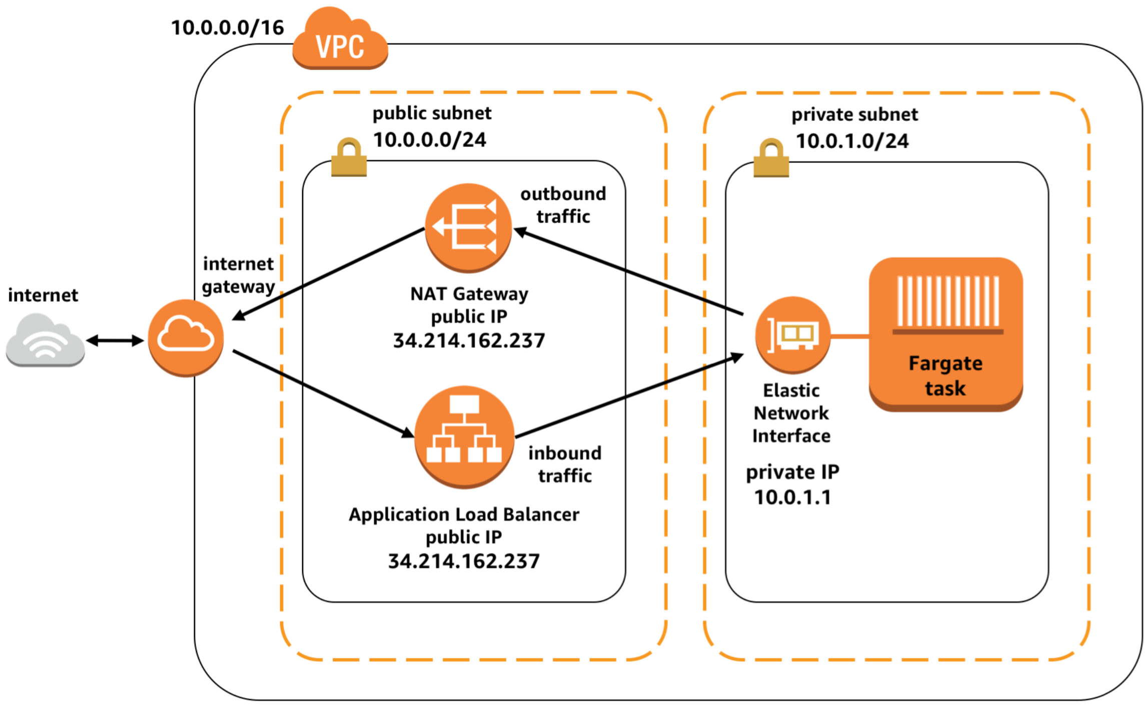
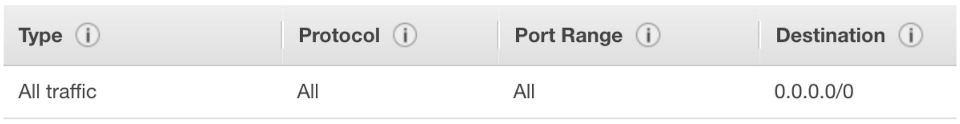


Figure 15: Private subnet with NAT gateway (AWS, 2018)

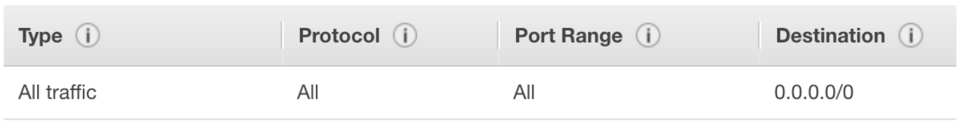
- Scale up/down policies are not optimal. These should be optimised based on peaks and troughs in traffic, and the metric should be chosen appropriately.
- Communications could be secured by attaching a secure listener to the load balancer using TLS certificates.

- Security groups should be more carefully considered. The security group used for this practical allows all inbound HTTP traffic and any type of outbound traffic:



Type	Protocol	Port Range	Destination
All traffic	All	All	0.0.0.0/0

Figure 16: Inbound Security Group rules



Type	Protocol	Port Range	Destination
All traffic	All	All	0.0.0.0/0

Figure 17: Outbound Security Group rules

References

[1] G2 Crowd. 2018 digital trend: Containers, Oct 2018.

[2] Docker. What is a container, Oct 2018.