

Workshop - Docker & ECS

Objetivo

El presente workshop tiene como objetivo mostrar un escenario real en donde se apliquen los conceptos vistos hasta la fecha (5 de octubre 2020) en la cursada M8B de Arquitectura de Software en la práctica.

Se espera que el alumno consolide estos conceptos acompañando el desarrollo del workshop. De esta manera generaremos un espacio de intercambio de dudas e inquietudes.

Requisitos previos

Para la correcta ejecución de este workshop, es necesario que el alumno:

- Haya repasado conceptos de la asignatura dictados hasta el momento: cloud computing, docker, docker-compose. AWS: modelo y servicios. 12 factor, business SaaS, performance testing, caching.
- Tenga su cuenta AWS Educate [dada de alta](#) y correctamente configurada

Aplicación

Para fines del workshop, desarrollamos un [servicio backend](#) que expone el siguiente recurso para ser operado bajo las recomendaciones REST

- posts: Un posteo representa una entrada en nuestro blog

Podremos realizar las siguientes operaciones

- POST /posts
Crea un nuevo posteo
- DELETE /posts/{id}
Elimina el posteo bajo el identificador "id"
- PUT /posts/{id}
Actualiza el posteo bajo el identificador "id"
- GET /posts
Retorna el listado de posts disponibles
- GET /posts/{id}
Retorna el posteo identificado por "id"

La aplicación se encuentra desarrollada en NodeJS, y hace uso de las siguientes dependencias

- koa
- @koa/router
- koa-bodyparser
- dotenv
- sequelize
- mysql2

Como podremos ver, es necesario contar con un MySQL para ejecutar y realizar pruebas. Además, estamos haciendo uso de 'dotenv' para manejar variables de entorno en nuestro desarrollo local. Es necesario que creamos un archivo llamado '.env' en la raíz de nuestro proyecto, y que cuente con las siguientes entradas:

- PORT: Puerto en donde levanta nuestra app
- DB_NAME: Nombre de la base de datos
- DB_USER: Usuario con permisos para operar sobre la base de datos
- DB_PASSWORD: Contraseña para acceder a la base de datos
- DB_HOST: Host donde se aloja el servidor mysql
- DB_LOG: Atributo que indica si se debe loguear la actividad de mysql

Desarrollo del workshop

Dockerizando la aplicación

El primer punto es tener un ambiente Dockerizado. Buscaremos dockerizar nuestra API, al mismo tiempo que buscaremos contar con un ambiente multi container para facilitar nuestro entorno de desarrollo. Particularmente no queremos lidiar con la instalación y configuración del MySQL.

Dockerfile para aplicación Node

Hemos visto en clases anteriores cómo Dockerizar una API desarrollada en NodeJS.

```
FROM node:12.18-alpine
COPY . ./app
WORKDIR /app
RUN npm install
CMD ["node", "index.js"]
```

Utilizando docker-compose

A partir de docker-compose seremos capaces de ejecutar un entorno mutli contenedor. Nuestro objetivo es ejecutar nuestra API en Node + nuestro servidor MySQL como backing service. Para esto configuramos el archivo YML de docker-compose como sigue:

```
version: "3.7"
services:
  node:
    build: .
    container_name: posts-api
    ports:
      - "3000:3000"
    depends_on:
      - mysql
    networks:
      - asp-network
  mysql:
    container_name: mysql-posts-api
    image: mysql:5.6
    env_file:
      - ./docker.env
    ports:
      - "3306:3306"
    networks:
      - asp-network
networks:
  asp-network:
    name: asp-network
```

Como podemos ver, se hace uso de muchas de las directivas que hemos trabajado en clase. Para que funcione correctamente, debemos crear el archivo "docker.env" en la raíz de nuestro proyecto, seteando algunas variables de entorno que son requeridas por la imagen de mysql:

- MYSQL_USER: Nombre de usuario para acceder a nuestra tabla
- MYSQL_PASSWORD: Contraseña para acceder a nuestra tabla
- MYSQL_ROOT_PASSWORD: Contraseña del usuario root
- MYSQL_DATABASE: Nombre de la base de datos con la que queremos trabajar (será creada al momento de levantar el contenedor)

Una vez que tengamos nuestro entorno configurado, podremos ejecutar los contenedores:

```
$ docker-compose up --build
```

A partir de este momento, podríamos interactuar con nuestra API a través de la colección de potsman que [hemos adjuntado](#)

Luego de algunas interacciones (crear, borrar, listar, actualizar), podemos validar cómo queda impactado directamente en el mysql, ingresando a su bash y ejecutando las consultas que quisiéramos:

```
$ docker exec -it mysql-posts-sql bash
$ # mysql -uposts_user -p
$ # use posts;
$ # select * from posts;
```

A este punto, contamos con un entorno de desarrollo en nuestras máquinas, listos para introducir cambios, o eventualmente pasar a desplegar nuestro servicio en la nube.

AWS Educate

Realizaremos las acciones necesarias dentro de nuestra cuenta de AWS Educate, para que podamos desplegar y manejar nuestra API. Particularmente buscamos:

- Crear un registro en ECR para nuestra imagen de docker
- Crear un backing service a partir de RDS para la gestión de la base de datos
- Crear un cluster en ECS que nos permita desplegar nuestra imagen en múltiples instancias
- Trabajar con un ELB para ir a un esquema de alta disponibilidad

Creando nuestro registry

Para desplegar nuestra aplicación es necesario contar con un registry para nuestra imagen de Docker.

Para esto, utilizaremos [Elastic Container Registry](#)

Crearemos un repositorio llamado “posts-api” que almacenará nuestra imagen

aws Servicios

ECR > Repositories > Create repository

Create repository

Access and tags

Repository name
[redacted].dkr.ecr.us-east-2.amazonaws.com/ posts-api
A namespace can be included with your repository name (e.g. namespace/repo-name).

Tag immutability
Enable tag immutability to prevent image tags from being overwritten by subsequent image pushes using the same tag. Disable tag immutability to allow image tags to be overwritten.
☒ Disabled

Una vez creado, lo veremos listado

ECR > Repositories

Repositories (1)

Find repositories

Repository name	URI	Created at
posts-api	[redacted].dkr.ecr.us-east-2.amazonaws.com/posts-api	10/04/20, 03:52:30 PM

En nuestra máquina deberemos ejecutar las instrucciones para autenticarnos en dicho ECR, haciendo uso de dos atributos

- Región en la que estamos operando
- URI del ECR

```
$ aws ecr get-login-password --region us-east-2 | docker login  
--username AWS --password-stdin  
*****.dkr.ecr.us-east-2.amazonaws.com/posts-api
```

Si todo funcionó correctamente, debería ver el mensaje “Login Succeeded”

```
[palbarrazin@uy-palabarrazin-n ~]$ aws ecr get-login  
WARNING! Your password will be stored unencrypted in  
Configure a credential helper to remove this warning  
https://docs.docker.com/engine/reference/commandline  
Login Succeeded
```

Los siguientes pasos involucran preparar nuestra imagen

- Taggear nuestra imagen de posts-api
- Pushear dicha imagen al repositorio

Para taggear la imagen, ejecutamos el comando “docker tag”, indicando cuál es la imagen seleccionada, y cómo se compone el tag. El mismo debe indicar la URI del ECR, y opcionalmente podría agregarse una versión.

```
docker tag posts-api_node:latest  
*****.dkr.ecr.us-east-2.amazonaws.com/posts-api
```

```
docker push *****.dkr.ecr.us-east-2.amazonaws.com/posts-api
```

Una vez ejecutados estos comandos, podremos acceder al detalle del registry a través de la consola de AWS, donde nos listará la imagen recién subida

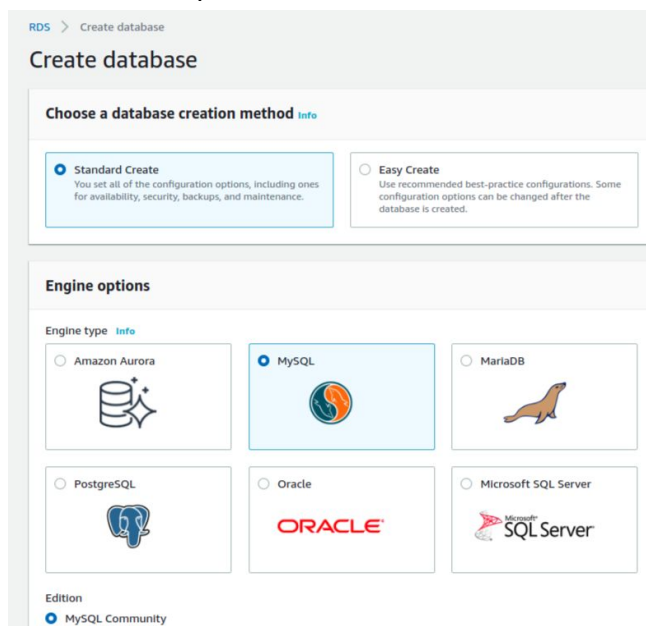


Creando nuestra base de datos

Nuestro servicio requiere de un backing service para su base de datos.

Para este fin, utilizaremos [Relational Database Service](#)

Para nuestra aplicación, se creará una instancia de MySQL



Luego de unos minutos, tendremos nuestra base de datos lista para ser usada.

IMPORTANTE - Verificar que nuestro security group permita la conexión remota a nuestra instancia de base datos tanto para entrada, como para salida de datos

Reglas de entrada			
Tipo	Protocolo	Intervalo de puertos	Origen
Todos los TCP	TCP	0 - 65535	0.0.0.0/0
Todos los TCP	TCP	0 - 65535	:::/0

Reglas de salida			
Tipo	Protocolo	Intervalo de puertos	Destino
Todo el tráfico	Todo	Todo	0.0.0.0/0
Todo el tráfico	Todo	Todo	:/0

De esta manera podemos crear nuestra base de datos en la instancia recién levantada

Creando clúster de ECS

Nuestro próximo paso será desplegar la imagen en un cluster de [Elastic Container Service](#)

En esta sección, buscaremos:

- Crear un cluster ECS de 2 instancias
- Crear un load balancer a partir de [ELB](#)
- Crear un servicio junto con una task definition que haga uso de nuestra imagen

Creación del cluster

Al momento de elegir un template para crear nuestro cluster, seleccionamos EC2 Linux + Networking, ya que es el que nos ofrece las features que buscamos

Select cluster template

The following cluster templates are available to simplify cluster creation. Additional configuration and integrations can be added later.

Networking only
Resources to be created:
Cluster
VPC (optional)
Subnets (optional)

Powered by AWS Fargate

EC2 Linux + Networking
Resources to be created:
Cluster
VPC
Subnets
Auto Scaling group with Linux AMI

EC2 Windows + Networking

Luego configuraremos nuestro cluster con las especificaciones que consideremos.
En este ejemplo lo configuramos para usar 2 instancias, y para que sea parte de la VPC con la que ya cuento

Configure cluster

Cluster name* ⓘ

☐ Create an empty cluster

Instance configuration

Configuración de red: Importante que este atributo se encuentre habilitado

Auto assign public IP Enabled ⓘ

Una vez configurado, procedemos a “Crearlo”. Esto lanzará el proceso, que puede demorar unos segundos, y al finalizar podremos ver el cluster.

Clusters

An Amazon ECS cluster is a regional grouping of one or more container instances on EC2 instance type.

For more information, see the [ECS documentation](#).

[Create Cluster](#) [Get Started](#)

View list card

[posts-cluster >](#)

CloudWatch monitoring
● Default Monitoring

FARGATE

0

Services

0

Running tasks

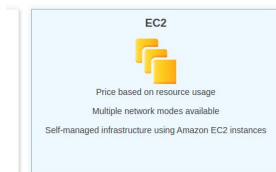
EC2

Creando nuestra definición de tarea

El próximo paso será crear una [task definition](#), que luego será parte de un [servicio](#) que será ejecutado en este cluster.

Entonces, creamos una task definition de tipo EC2

compatible with based on where you want to launch your task.



Lo importante en la task definition es que agreguemos el container con el que trabajará. La URI de nuestra imagen la obtenemos de nuestro ECR.

▼ Standard

Container name*

Image*

Además, debemos prestar especial atención a las siguientes configuraciones:

- Variables de entorno: Son las propias a ejecutar en este contenedor. Son análogas a las que configuramos localmente en el archivo `‘.env’`, pero para nuestro despliegue

en ECS.

Environment Variables		
Key		Value/ValueFrom
DB_HOST	posts-	us-east-2.rds.amazonaws.com
DB_LOG		true
DB_NAME		posts
DB_PASSWORD		
DB_USER		posts
PORT		3000

- **Mapeo de puertos:** A fines de la configuración que luego haremos con nuestro ELB, debemos mapear el puerto 80 del host, hacia aquel puerto que hayamos configurado como variable de entorno en el contendor (3000 en el ejemplo)

Port Mappings		
Host Port	Container Port	Protocol
80	3000	tcp

Una vez creada, verificamos que esté OK en el panel de listado de Task Definitions

Task Definitions

Task definitions specify the container information for your application, such as how many containers are part of your task, what resources they will use, how the

Create new Task Definition Create new revision Actions

Status: (ACTIVE) INACTIVE

Filter in this page

Task Definition	Latest revision status
posts-api-task	ACTIVE

Creando nuestro balanceador de carga

Antes de continuar con la creación de nuestro servicio para ser ejecutado en el cluster, vayamos a EC2 para crear nuestro balanceador de carga.

Para esto usaremos [Elastic Load Balancing](#), haciendo uso del tipo “Application Load Balancer”

Application Load Balancer

HTTP HTTPS

Create

Choose an Application Load Balancer when you need a flexible feature set for your web applications with HTTP and HTTPS traffic. Operating at the request level, Application Load Balancers provide advanced routing and visibility features targeted at application architectures, including microservices and containers.

[Learn more >](#)

Allí, ingresamos un nombre para identificar el balanceador de carga, y seleccionamos la VPC sobre la que funcionará. También indicamos el puerto donde se ejecuta, el cual dejamos en su default (80)

Listeners

A listener is a process that checks for connection requests, using the protocol and port that you configured.

Load Balancer Protocol	Load Balancer Port
HTTP	80

Add listener

Al final de la configuración del load balancer, nos pedirá crear un target group. Este target group será el que contendrá las instancias de nuestro cluster. Con esto sabrá hacia dónde realizar el balanceo de carga.

Es importante también configurar el endpoint de healthcheck, con el que sabrá si una instancia es candidata o no para realizar el balanceo de la carga

Step 4: Configure Routing
Your load balancer routes requests to the targets in this target group using the protocol a

Target group

Target group ⓘ

Name ⓘ

Target type
☒ Instance
☐ IP
☐ Lambda function

Protocol ⓘ

Port ⓘ

Health checks

Protocol ⓘ

Path ⓘ

▶ Advanced health check settings

Luego pedirá registrar los targets. Por el momento obviemos este paso. Con esto finalizamos el proceso y tenemos nuestro load balancer listo.

Create Load Balancer Actions

search : arn:aws:elasticloadbalancing:us-east-2:... Add filter

Name	DNS name
posts-api-load-balancer	posts-api-load-balancer-200...

Creando nuestro servicio para ECS

El próximo paso se basa en crear un servicio: un servicio ejecuta una task definition dentro de un conjunto de recursos.

Primero configuramos atributos básicos de un servicio, en donde se incluye cuál es la task que lo compone

Configure service

A service lets you specify how many copies of your task definition to run and maintain in a cluster. Amazon ECS uses an Elastic Load Balancing load balancer to distribute incoming traffic to containers in your service. Amazon ECS also coordinates task scheduling with the load balancer. You can also optionally use Service Auto Scaling to scale the number of tasks in your service.

Launch type ☐ FARGATE ☒ EC2 ⓘ

[Switch to capacity provider strategy](#) ⓘ

Task Definition Family
 ▼

Revision
 ▼

Cluster ▼ ⓘ

Service name ⓘ

Service type* ☒ REPLICHA ☐ DAEMON ⓘ

Number of tasks ⓘ

Minimum healthy percent ⓘ

Maximum percent ⓘ

Luego podemos configurar un balanceador de carga. Aquí indicamos “Application Load Balancer” (ya que fue el tipo de balanceador que hemos creado), y lo seleccionamos de la lista de disponibles.

Load balancing

An Elastic Load Balancing load balancer distributes incoming traffic across the tasks running in your service. You can use an existing load balancer, or create a new one in the [Amazon EC2 console](#).

Load balancer type*

☐ None
Your service will not use a load balancer.

☒ **Application Load Balancer**
Allows containers to use dynamic host port mapping (multiple tasks allowed per instance). Multiple services can use the same listener port on a single load balancer. Supports advanced routing and paths.

☐ Network Load Balancer
A Network Load Balancer functions at the fourth layer of the Open Systems Interconnection model. After the load balancer receives a request, it selects a target from the default rule using a flow hash routing algorithm.

☐ Classic Load Balancer
Requires static host port mappings (only one task allowed per container instance). Advanced routing and paths are not supported.

Service IAM role ⓘ

Load balancer name

También indicamos cuál es el container sobre el cual se hará load balancing. En este ejemplo tenemos uno solo, pero podríamos contar con varios que trabajen en diferentes endpoints (ej. aplicación backend + frontend)

Container to load balance

posts-api-container : 3000

[Remove](#)

Production listener port* 80:HTTP ⓘ

Production listener protocol* HTTP

Target group name posts-api-target-group ⓘ

Target group protocol HTTP ⓘ

Target type instance ⓘ

Path pattern / Evaluation order default

Health check path /health ⓘ

Additional health check options can be configured in the ELB console after you create your service.

Opcionalmente puedo sumar alguna política de auto-scaling. Esto permitirá hacer uso de la elasticidad de nuestras instancias, logrando así un uso eficiente de los recursos de acuerdo a la política que configuremos (CPU, Memoria, Requests en el ELB, etc)

Luego de estos pasos, podemos ver el servicio creado con los parámetros configurados

[Clusters](#) > [posts-cluster](#) > Service: posts-api-service-1

Service : posts-api-service-1

Cluster	posts-cluster	Desired count	2
Status	ACTIVE	Pending count	0
Task definition	posts-api-task-definition:3	Running count	2
Service type	REPLICA		
Launch type	EC2		
Service role	AWSRoleForECS		
Created By	arn:aws:iam::670780503335:root		

Details	Tasks	Events	Auto Scaling	Deployments	Metrics	Tags
---------	-------	--------	--------------	-------------	---------	------

Target Group Name	Container Name	Container Port
posts-api-target-group	posts-api-container	3000

Health check grace period	0
---------------------------	---

Podremos verificar que se encuentra corriendo accediendo al dominio expuesto por nuestro load balancer

← → ↻ ⚠ No seguro | posts-api-load-balancer-2000941073.us-east-2.elb.amazonaws.com/health

```
{
  "status": "ok"
}
```

Probemos además otras operaciones, como crear o listar recursos:

▶ Create Blog post

POST http://posts-api-load-balancer-2000941073.us-east-2.elb.amazonaws.com/posts

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings

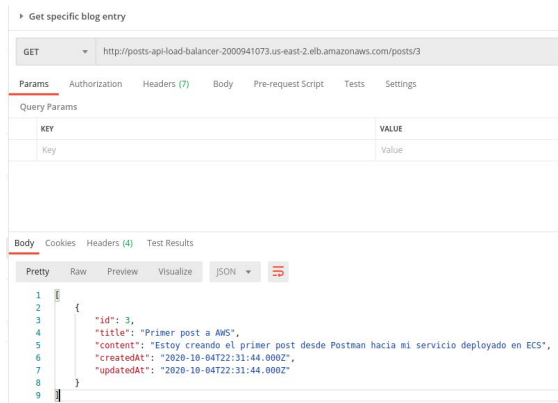
none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "title": "Primer post a AWS",
3   "content": "Estoy creando el primer post desde Postman hacia mi servicio deployado en ECS"
4 }
```

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 3,
3   "title": "Primer post a AWS",
4   "content": "Estoy creando el primer post desde Postman hacia mi servicio deployado en ECS",
5   "updatedAt": "2020-10-04T22:31:44.979Z",
6   "createdAt": "2020-10-04T22:31:44.979Z"
7 }
```



¿Qué aprendimos?

- **Ejemplo real world!**
- Dockerizar nuestra aplicación
- Hacer uso de las capacidades de AWS para crear recursos
 - Para nuestra imagen de docker
 - Para nuestra aplicación
 - Para nuestro backing service
- Desplegar una aplicación con atributos de
 - Alta disponibilidad, a través de múltiples copias y el uso de un balanceador de carga
 - Escalabilidad, a través de políticas de auto-scaling