UNIVERSIDAD ORT
Uruguay

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 creemos 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:

```yaml
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:

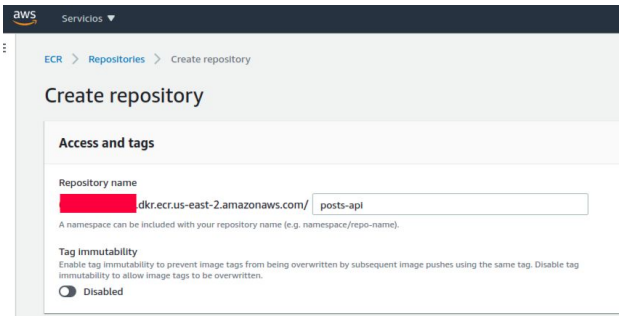
- Crear un registro en ECR para nuestra imagen de docker
- Crear un backing service a partir de RDS para la gestión de la base de datos
- Crear un cluster en ECS que nos permita desplegar nuestra imagen en múltiples instancias
- Trabajar con un ELB para ir a un esquema de alta disponibilidad

Creando nuestro registry

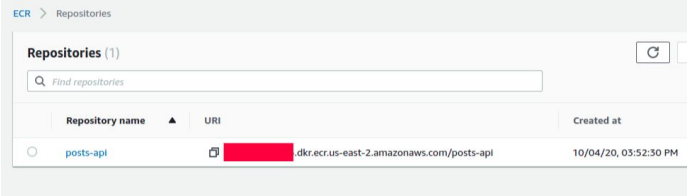
Para desplegar nuestra aplicación es necesario contar con un registry para nuestra imagen de Docker.

Para esto, utilizaremos Elastic Container Registry

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



Una vez creado, lo veremos listado

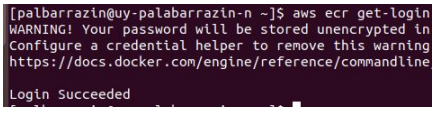


En nuestra máquina deberemos ejecutar las instrucciones para autenticarnos en dicho ECR, haciendo uso de dos atributos

- 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”



Los siguientes pasos involucran preparar nuestra imagen

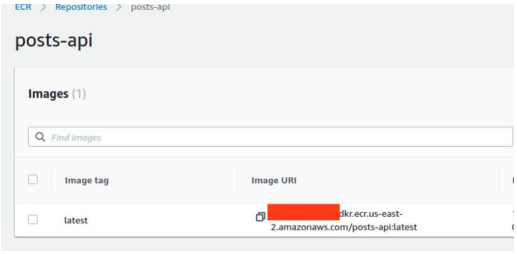
- Taguear nuestra imagen de posts-api
- Pushear dicha imagen al repositorio

Para taguear 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
```

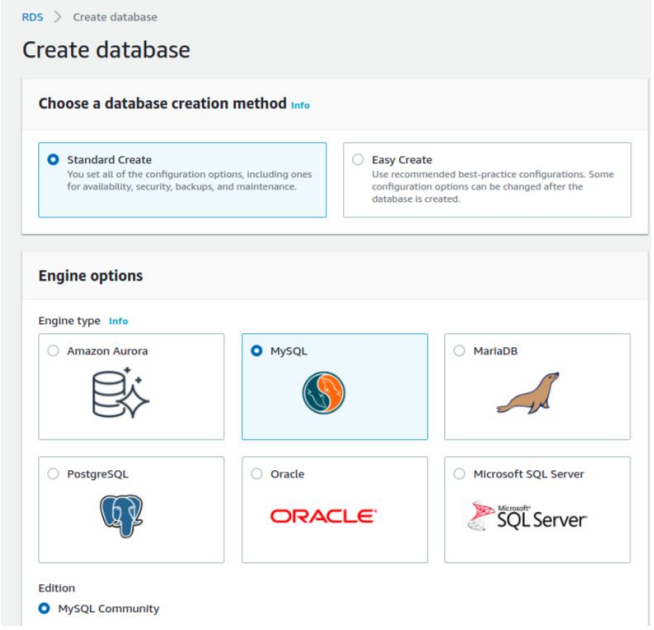
Una vez ejecutados estos comandos, podremos acceder al detalle del registry a través de la consola de AWS, donde nos listará la imagen recién subida



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 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

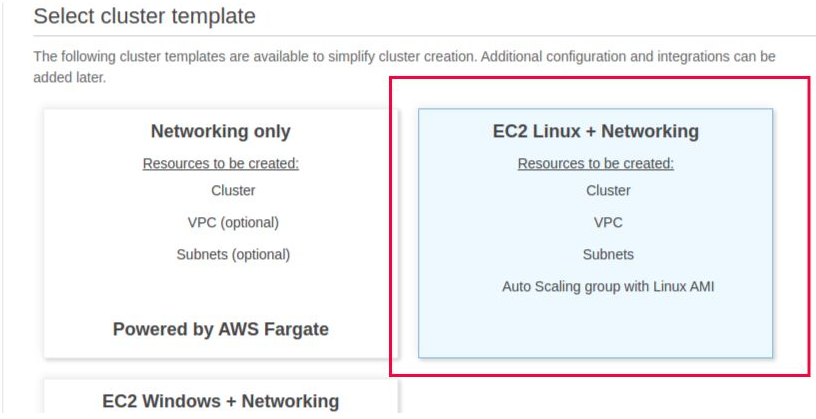
Nuestro próximo paso será deplegar la imagen en un cluster de Elastic Container Service

En esta sección, buscaremos:

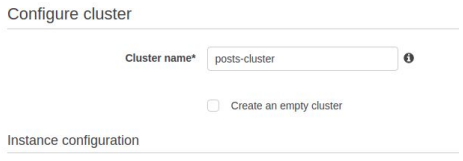
- Crear un cluster ECS de 2 instancias
- Crear un load balancer a partir de ELB
- Crear un servicio junto con una task definition que haga uso de nuestra imagen

Creación del cluster

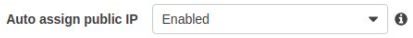
Al momento de elegir un template para crear nuestro cluster, seleccionamos EC2 Linux + Networking, ya que es el que nos ofrece las features que buscamos



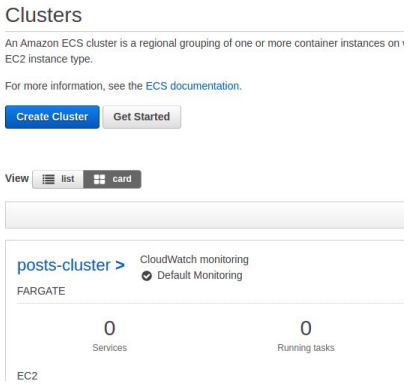
Luego configuraremos nuestro cluster con las especificaciones que consideremos.
En este ejemplo lo configuramos para usar 2 instancias, y para que sea parte de la VPC con la que ya cuento



Configuración de red: Importante que este atributo se encuentre habilitado



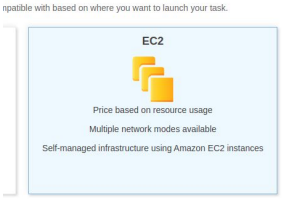
Una vez configurado, procedemos a “Crearlo”. Esto lanzará el proceso, que puede demorar unos segundos, y al finalizar podremos ver el cluster.



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

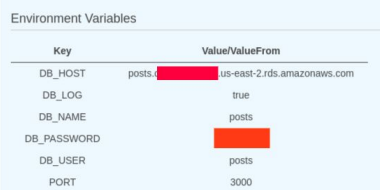


Lo importante en la task definition es que agreguemos el container con el que trabajará. La URI de nuestra imagen la obtenemos de nuestro ECR.

Además, debemos prestar especial atención a las siguientes configuraciones:

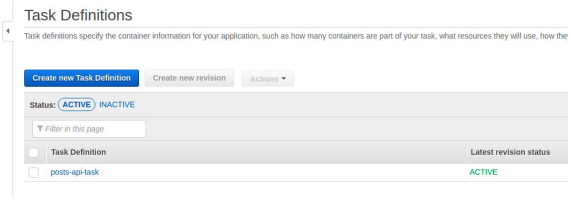
- Variables de entorno: Son las propias a ejecutar en este contenedor. Son análogas a las que configuramos localmente en el archivo ‘.env’, pero para nuestro despliegue

en ECS.



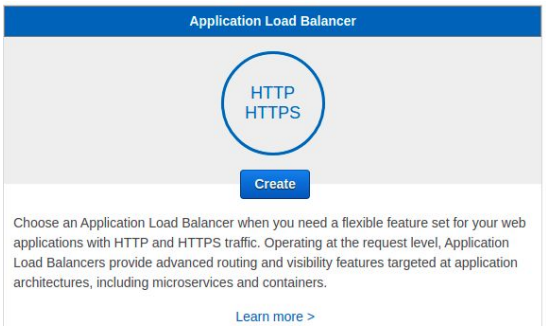
- Mapeo de puertos: A fines de la configuración que luego haremos con nuestro ELB, debemos mapear el puerto 80 del host, hacia aquel puerto que hayamos configurado como variable de entorno en el contendor (3000 en el ejemplo)

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

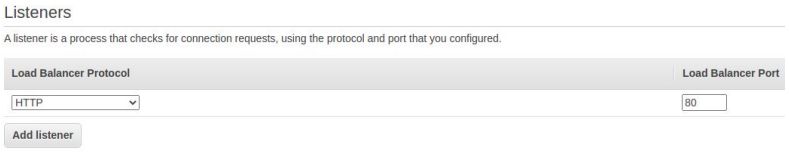


Creando nuestro balanceador de carga

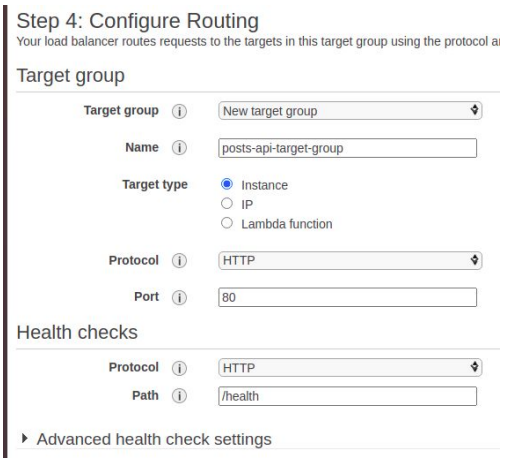
Antes de continuar con la creación de nuestro servicio para ser ejecutado en el cluster, vayamos a EC2 para crear nuestro balanceador de carga.
Para esto usaremos Elastic Load Balacing, haciendo uso del tipo “Application Load Balancer”



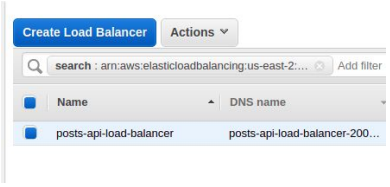
Allí, ingresamos un nombre para identificar el balanceador de carga, y seleccionamos la VPC sobre la que funcionará. También indicamos el puerto donde se ejecuta, el cual dejamos en su default (80)



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



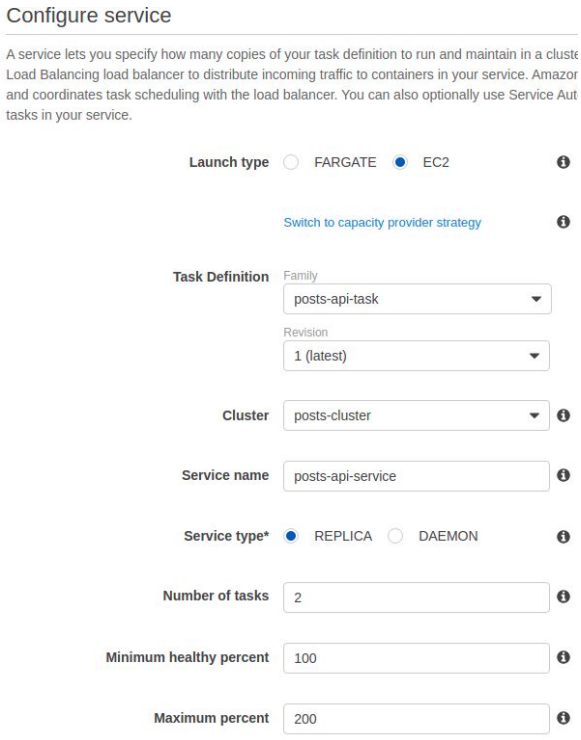
Luego pedirá registrar los targets. Por el momento obviamos este paso. Con esto finalizamos el proceso y tenemos nuestro load balancer listo.



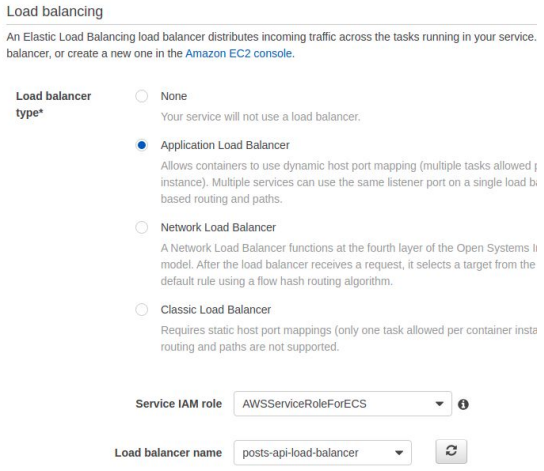
Creando nuestro servicio para ECS

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

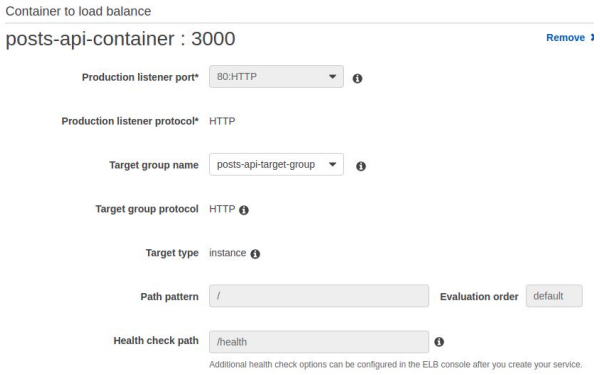
Primero configuramos atributos básicos de un servicio, en donde se incluye cuál es la task que lo compone



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.

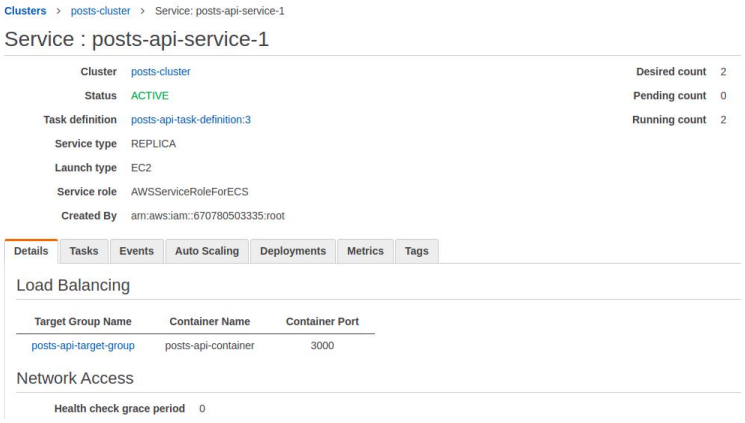


También indicamos cuál es el container sobre el cual se hará load balancing. En este ejemplo tenemos uno solo, pero podríamos contar con varios que trabajen en diferentes endpoints (ej. aplicación backend + frontend)

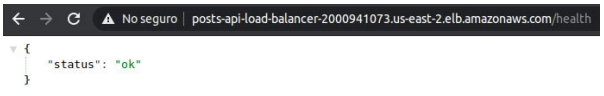


Opcionalmente puedo sumar alguna política de auto-scaling. Esto permitirá hacer uso de la elasticidad de nuestras instancias, logrando así un uso eficiente de los recursos de acuerdo a la política que configuremos (CPU, Memoria, Requests en el ELB, etc)

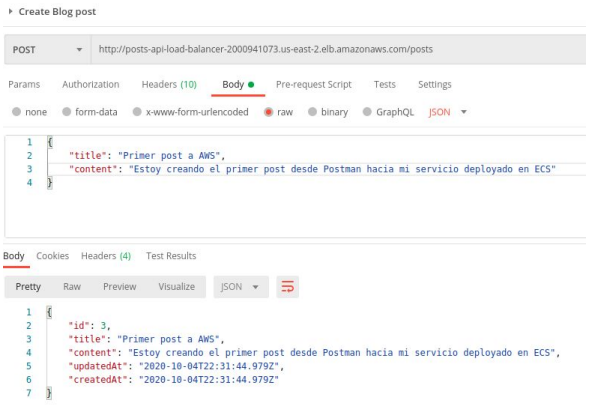
Luego de estos pasos, podemos ver el servicio creado con los parámetros configurados

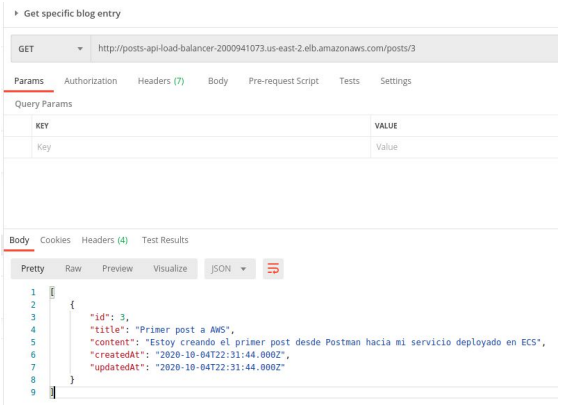


Podremos verificar que se encuentra corriendo accediendo al dominio expuesto por nuestro load balancer



Probemos además otras operaciones, como crear o listar recursos:





¿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