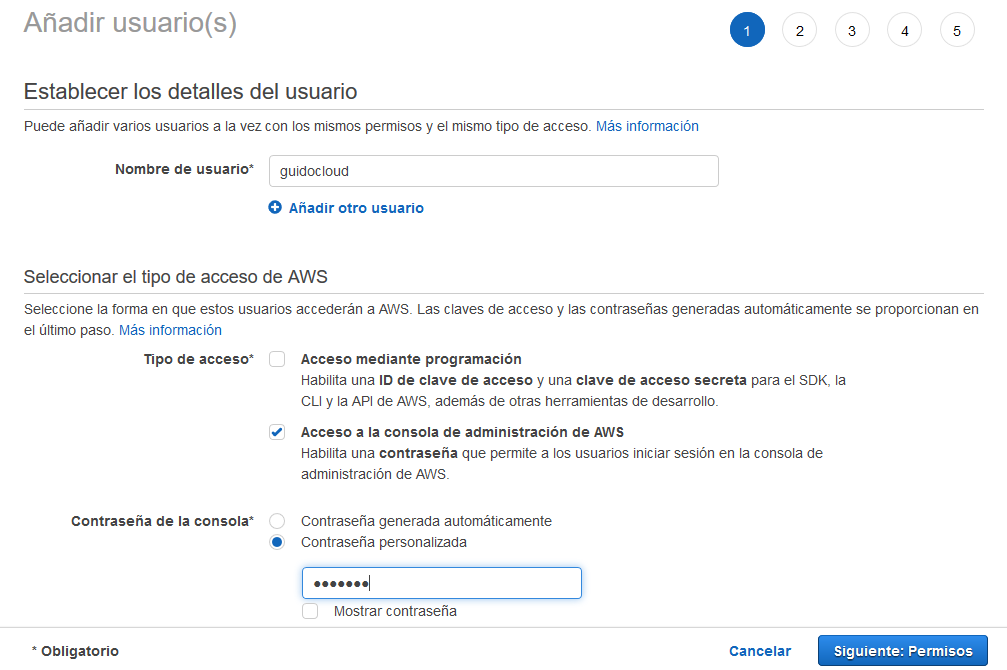
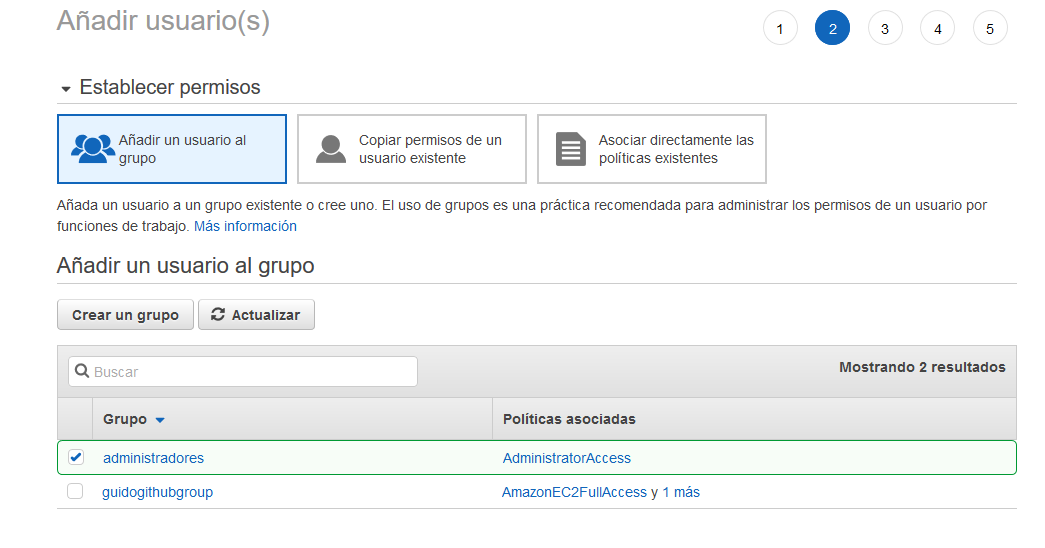
Como buena práctica no se debe usar el usuario root, en este caso crearemos un usuario que administrara y creara infraestructura, instancias, bases de datos, buckets, etc. Lo agregaremos en un grupo administradores, el cual tendrá las politicas correspondientes.

1. Crear usuario IAM



1. Creación de grupo



\*Agrego el usuario al grupo creado administradores, en este caso no utilizo la política de least privileges principle, la cual indica que los permisos deben ser los mínimos indispensables para cada usuario. Esto quedara así a fin de agilizar el trabajo en este entorno no corporativo.

Las políticas se definen con formato JSON (aunque en la UI se pueden ver y agregar de forma manual).

En este caso es bastante sencillo, wildcard \* para todos los permisos y todos los recursos (ARN)

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": "\*",

"Resource": "\*"

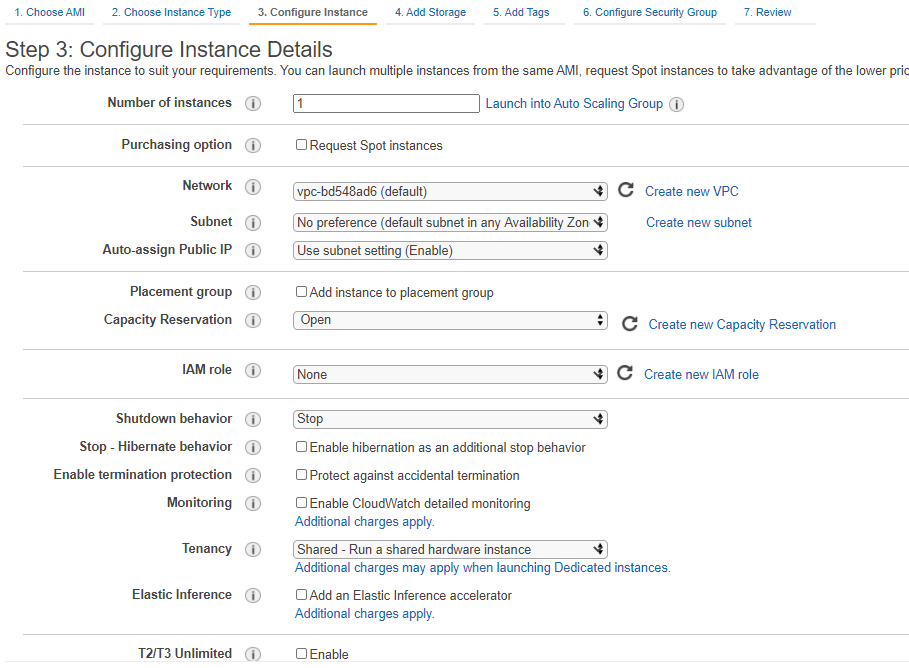
}

]

}

Una vez creado el usuario salgo de la cuenta root y utilizare el nuevo admin guidocloud, junto al id de la cuenta x para ingresar.

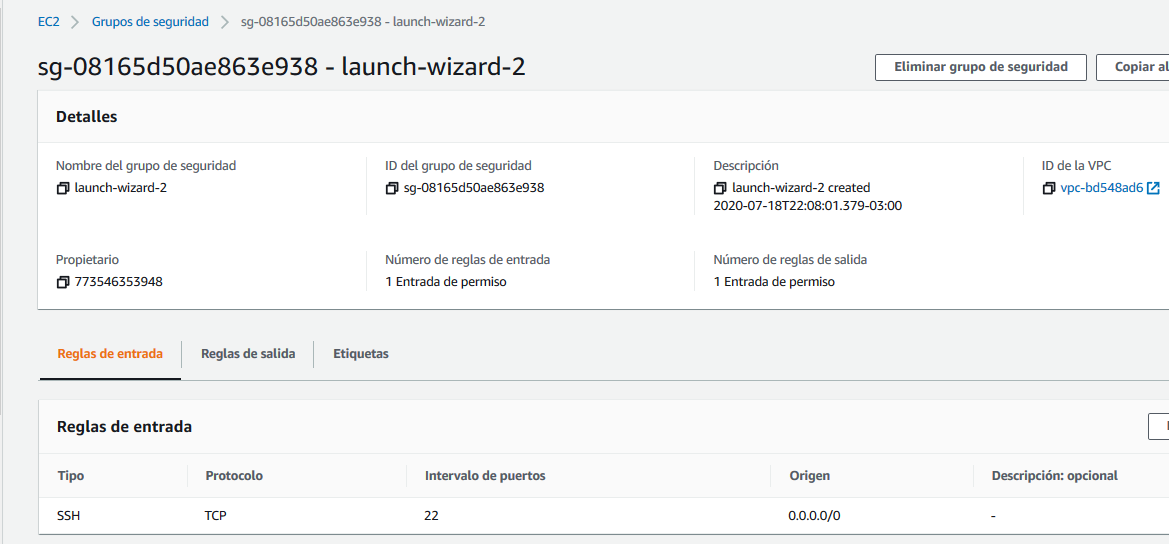
1. Creación de instancia EC2 (de prueba)



Utilizare la red default en un principio, tanto como la subnet. Elegimos la TIER gratuita de instancia, y como storage un volumen type Root (el cual se elimina al eliminar la instancia por default, al contrario de los otros tipos de disco ‘delete on termination’).

Sera de 8gb, y gp2 (SSD general purpose), el cual tiene una cantidad de IOPS suficiente, pero inferior a la de otro tipos de discos más costosos. Por otro lado no tendrá encriptación (esta opción se selecciona al crearse, luego no se puede cambiar).

1. Security Groups



Al crearse una instancia se crea un security group default, con inbound rules y outbound rules. Esta trae por default el puerto 22 (ssh) parece. Si es un servidor web habrá que abrir el puerto 80, 443, y asi.

\*Un security group está relacionado a una instancia ya que los grupos de seguridad están asociados a interfaces de red. (Trabaja como un firewall, filtra puertos, ips tráfico de entrada/salida)

1. Conectarse a la instancia (3 opciones)

.En este caso ingresare desde el navegador web con la opción de AWS (la cual crea en la instancia un token temporal para que funcione, sin que tengamos que hacer nada).

.Tambien se puede utilizar el Putty (ssh), convirtiendo el pem en ppk.

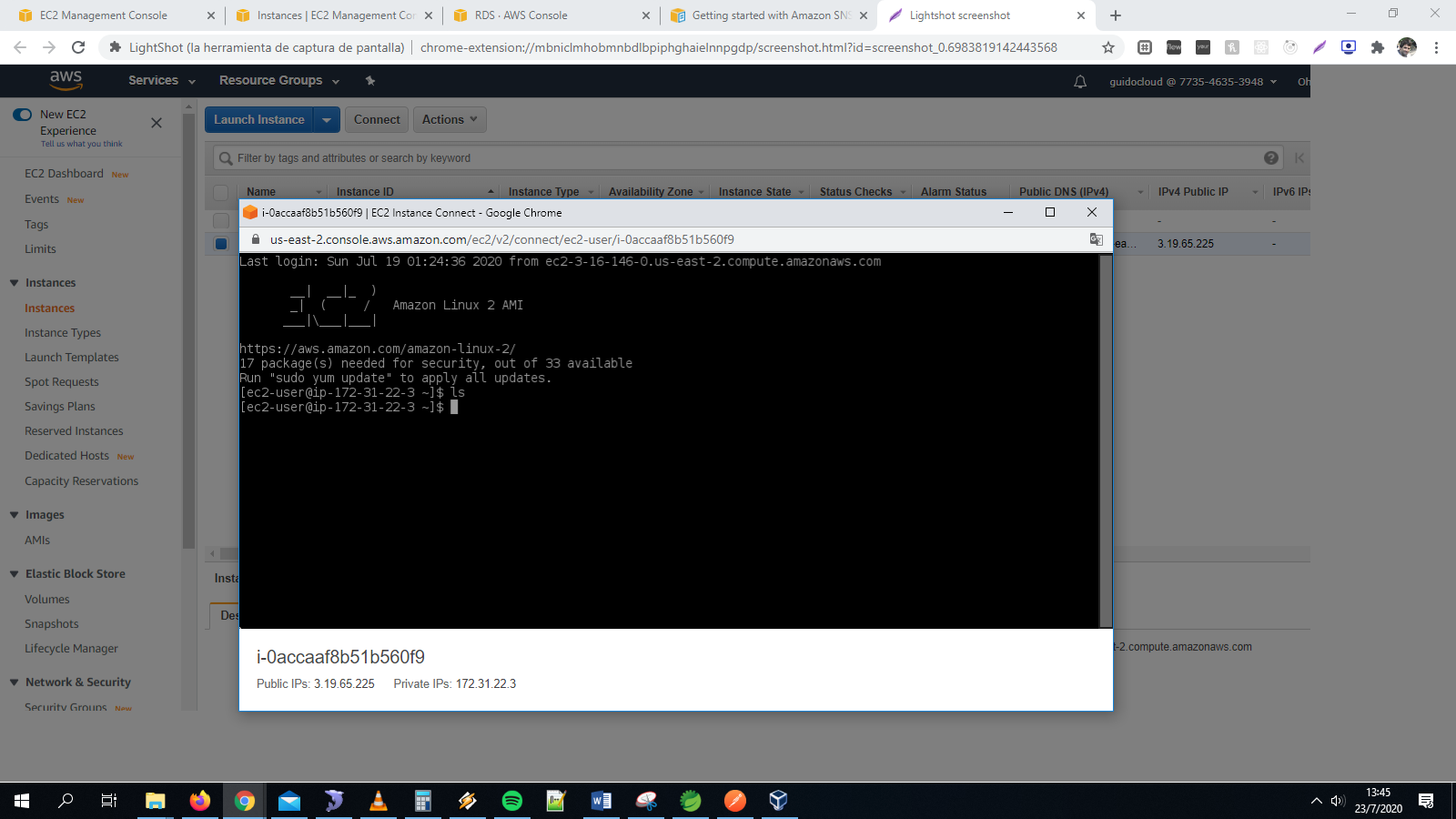
.Consola (SSH): ojo que cambia ip publica cuando reiniciamos la instancia para conectarse.

En Windows 10 y Linux se puede usar el comando SSH desde la terminal/powershell

Ej: Ssh –i c:/clave.pem [ec2-user@35.233.452.111](mailto:ec2-user@35.233.452.111)

Esto puede dar error de permisos, hay que meter un chmod 0400 al .pem o en permisos de Windows (full control)

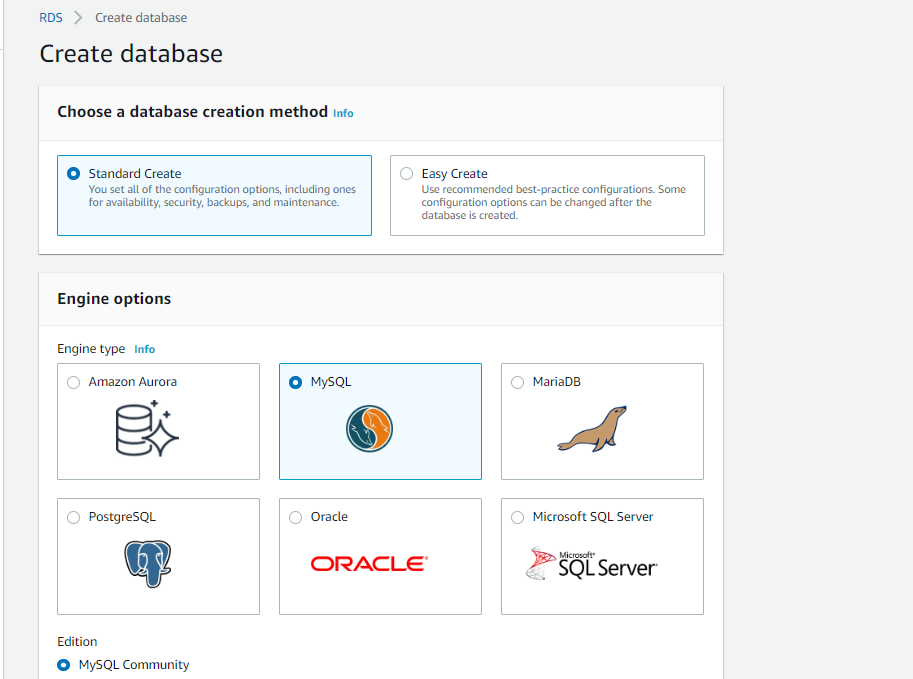
Via navegador:

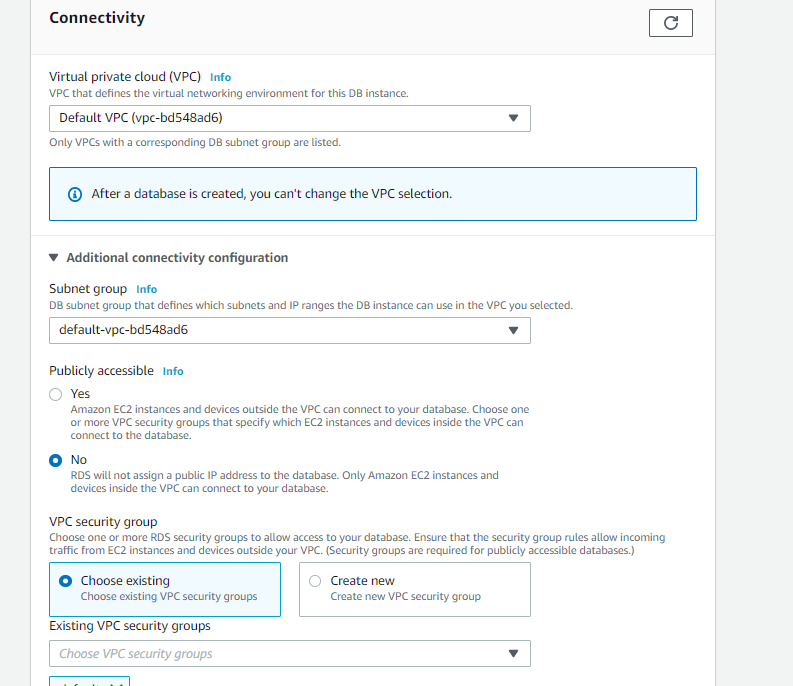


1. Base de datos RDS: databaseHispano

Master username: \*

Sin acceso público, accederemos solo desde nuestra instancia y el puerto correspondiente.

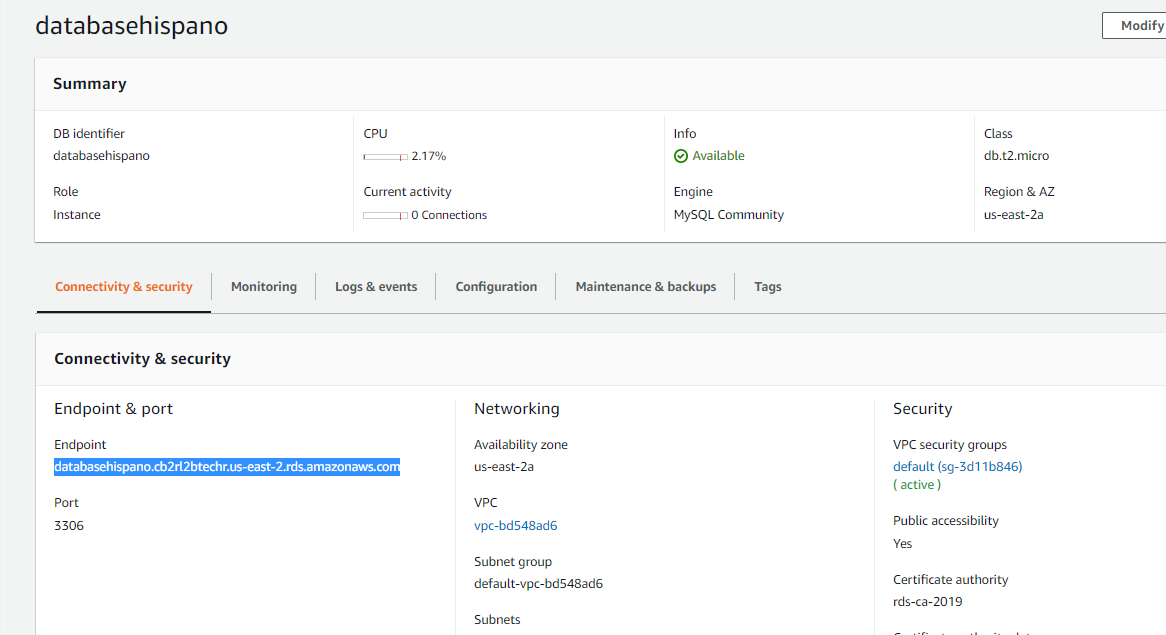


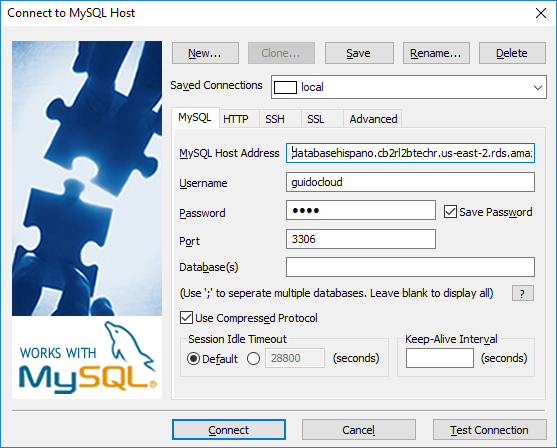


Para importar datos y estructura de la base local, realizo un sql dump, y para subirla a RDS que configuro con acceso público de la base de datos en AWS, que luego deshabilito nuevamente. Esto de cambia desde RDS-Databases-Modify-Additional connectivity configuration (tarda unos minutos al cambiar la opción)

La conexión la hice con el SQLYOG, pero podría hacerla manualmente por consola desde la instancia EC2 que tiene acceso a la base.

Recordar que también tiene que tener el inbound del security group con mi IP o todas las IP para acceder.





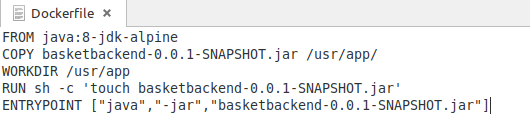
databasehispano.cb2rl2btechr.us-east-2.rds.amazonaws.com

**Para deployar en docker en local**

1. Tome proyecto Spring Boot y en vez de exportar un .war, hacer un maven install y tomar el .jar de la carpeta target (fat jar, con librerías incluidas) y dejarlo en la misma carpeta donde está el dockerfile. \*Este proyecto está configurado para acceder a la base de AWS de RDS, en una imagen, en un repositorio privado. No así el código fuente, donde no se guarda esta información.

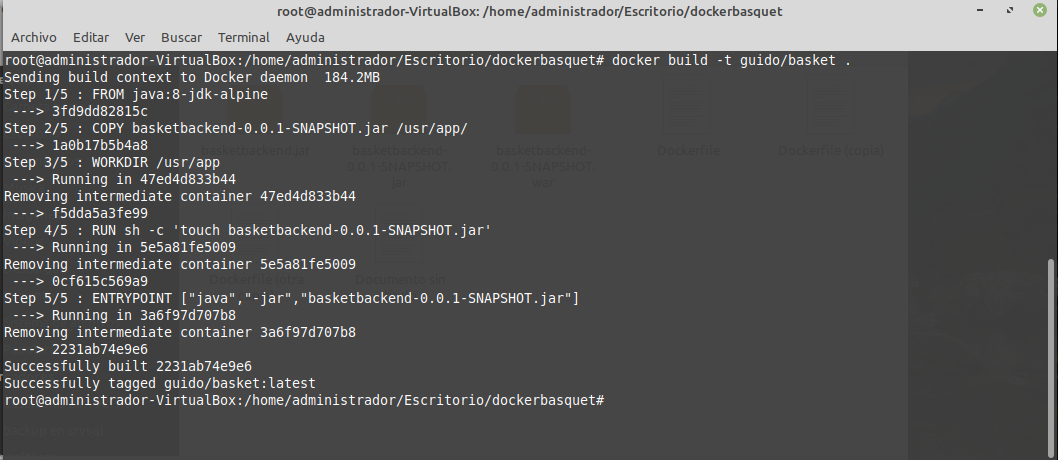
2. Crear una imagen de nuestra aplicación, esto se realiza con un archivo Dockerfile (archivo de texto, el cual tiene las indicaciones para levantar la imagen con las dependencias y SO necesarias, en este caso, sería solamente la máquina virtual de java, más una distribución de Linux muy pequeña).

Archivo Dockerfile (sin .txt ni nada).



3. Correr el comando que construye la imagen (se corre desde el directorio donde está el dockerfile, y el archivo .jar):

docker build -t guido/basket .



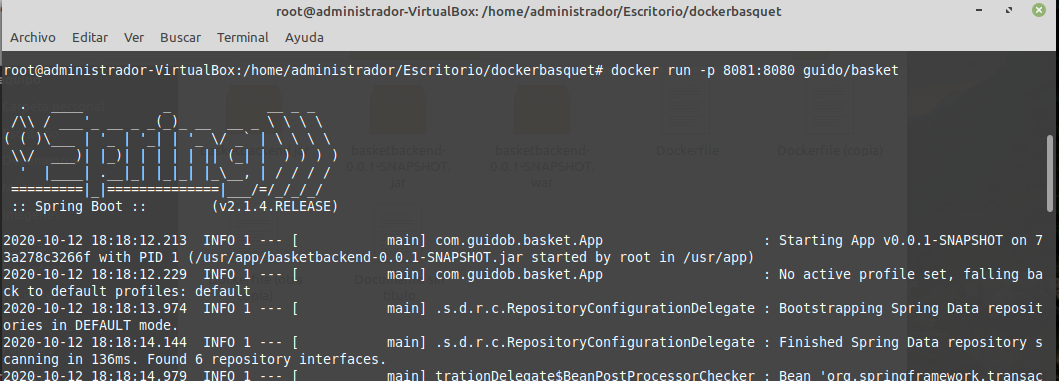
(ejemplo):

docker build -t my\_docker\_hub\_username/my\_image\_name:my\_image\_version .

Ahora podemos ver en nuestro repositorio que la imagen existe (comando docker images).

4. Ejecutar comando para correr el contenedor docker, acá vemos que se le pasa parámetros donde indicamos que Puerto escucha y por ejemplo la dirección de la base de datos que va a utilizar:

docker run -p 8081:8080 guido/basket



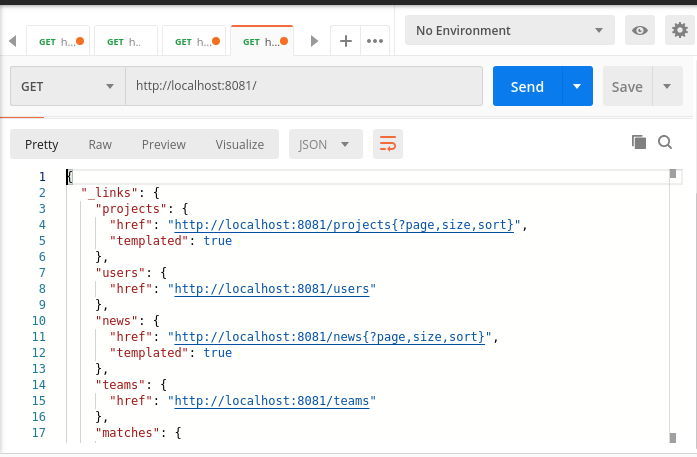
Aca se le pega al puerto 8081 y docker lo redirige al 8080 que tiene la aplicación configurado.

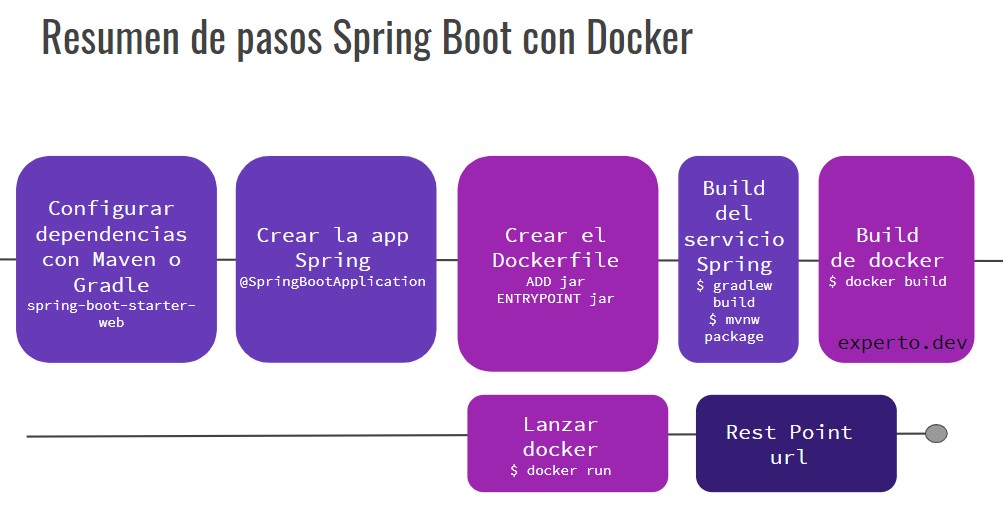
(Este ejemplo deje la apuntando el archivo application.properties con la base de RDS)

(ejemplo):

docker run -p 8080:8080 --env SPRING\_PROFILES\_ACTIVE=docker my\_docker\_hub\_username/my\_image\_name:my\_image\_version

5. Ya estamos corriendo en docker:





Para entrar al container que fue creado:

docker exec -it e559262b3578 bash (donde el numero es el ID del contenedor)

Para iniciar si fue cerrado:

docker start e559262b3578

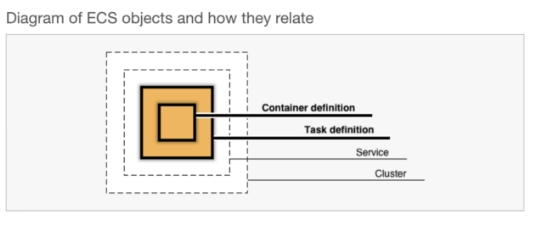
\*\*Los contenedores stopeados no consumen recursos, solo espacio de disco. Por ello hay que eliminarlos si no los usaremos más.

docker ps --size --filter "status=exited"

Si bien el tamaño de la imagen original se comparte, el adicional no.

**Deployar docker en AWS**

Utilizo AWS ECS (Elastic Container Service) en el cual creamos un **Cluster** (con Fargate), este cluster crea una o varias tareas, donde se correrá docker **(evito utilizar instancias EC2 como lo describi al principio** asociadas a un AGENT de ECS para administrar).

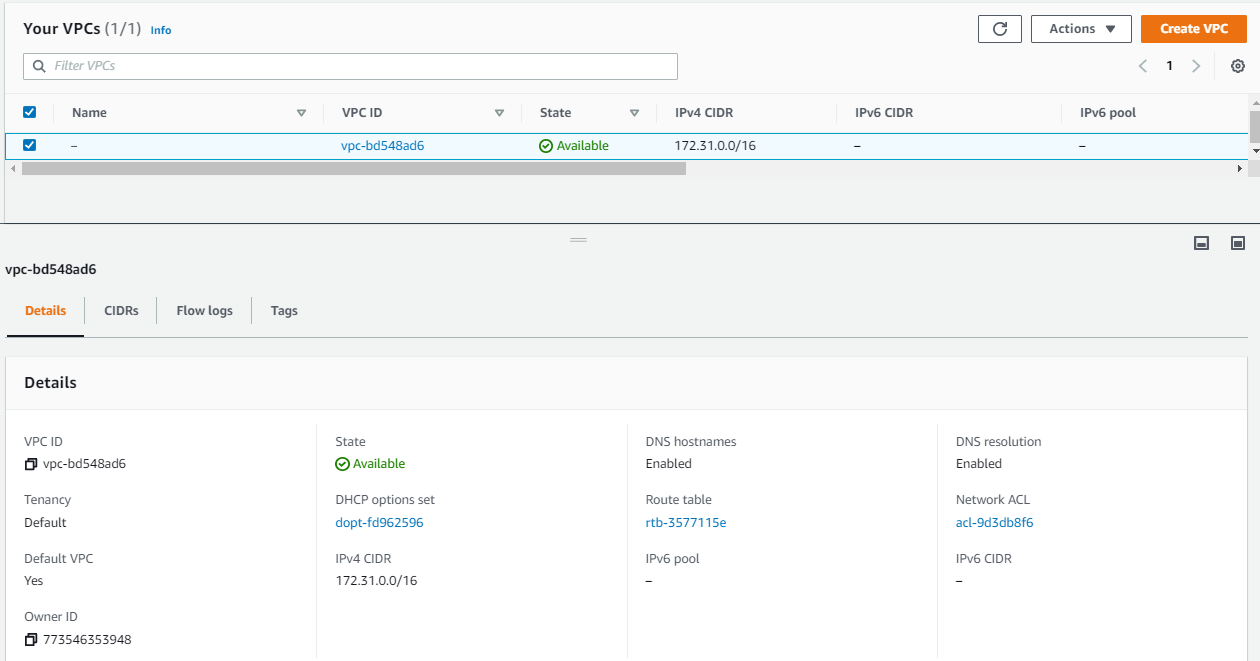




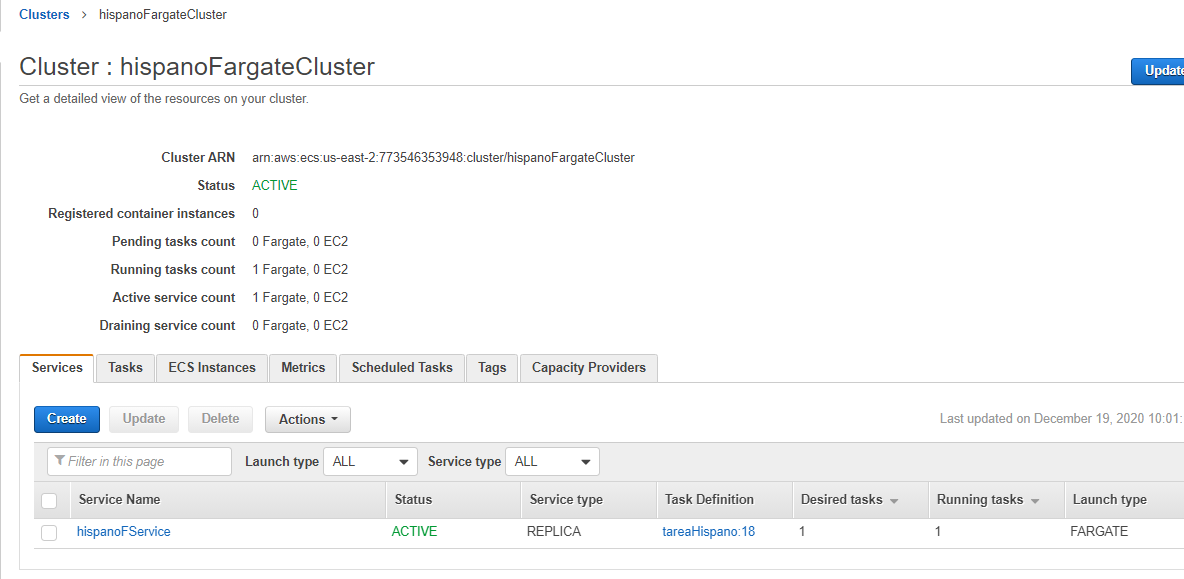
Cuando creamos un cluster se asocia o crea un ASG (auto scalling group), y este es el que crea las instancias EC2/Fargate Task que definí arriba.

Para correr todo esto, necesitamos un cluster, un service, una task, un Aplication Load Balancer, un Auto Scaling Group, un target group y dos Security Group (para los dos últimos) y VPC.

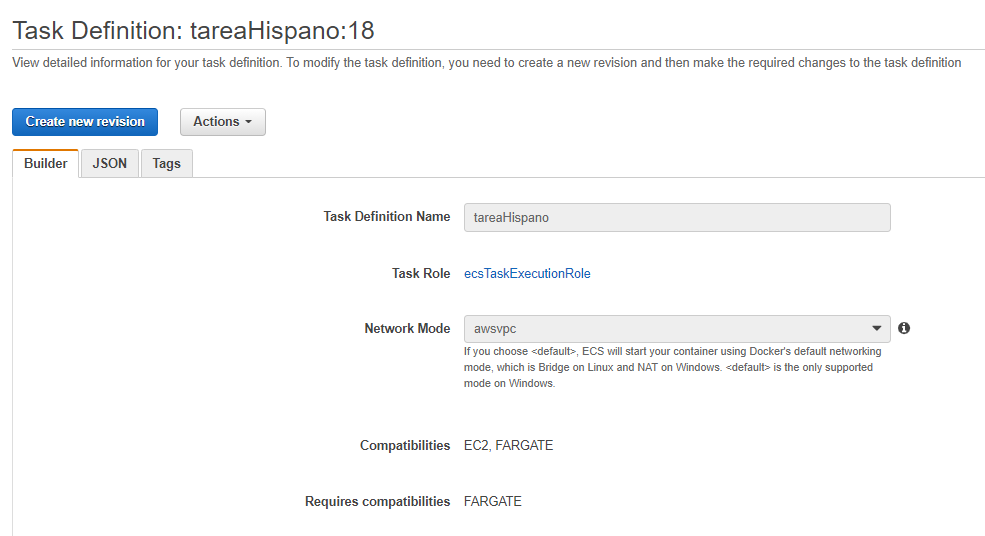
Primero, trabajamos todo en una misma VPC (virtual private cloud):

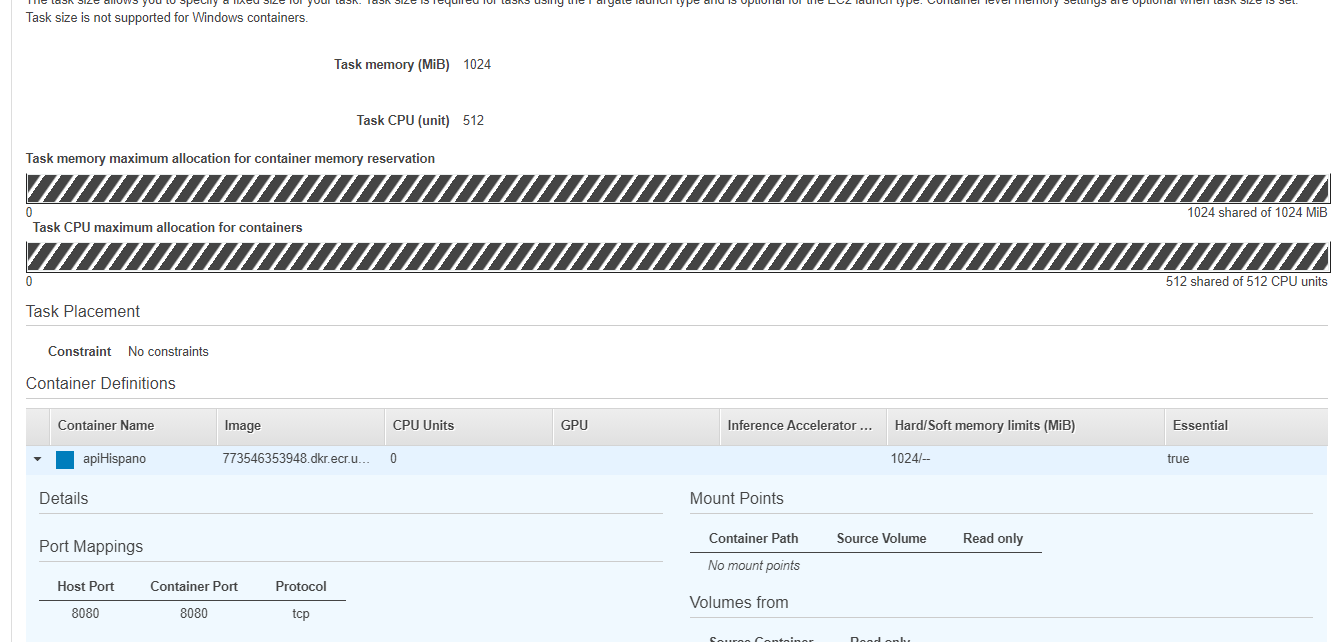


Cluster:



Hay que crear una **task definition**, donde le especificamos la ram y cpu a usar **por cada** conteiner, como también la imagen que utilizara y el puerto, entre otras configuraciones. (Seria como el run de docker a mano)



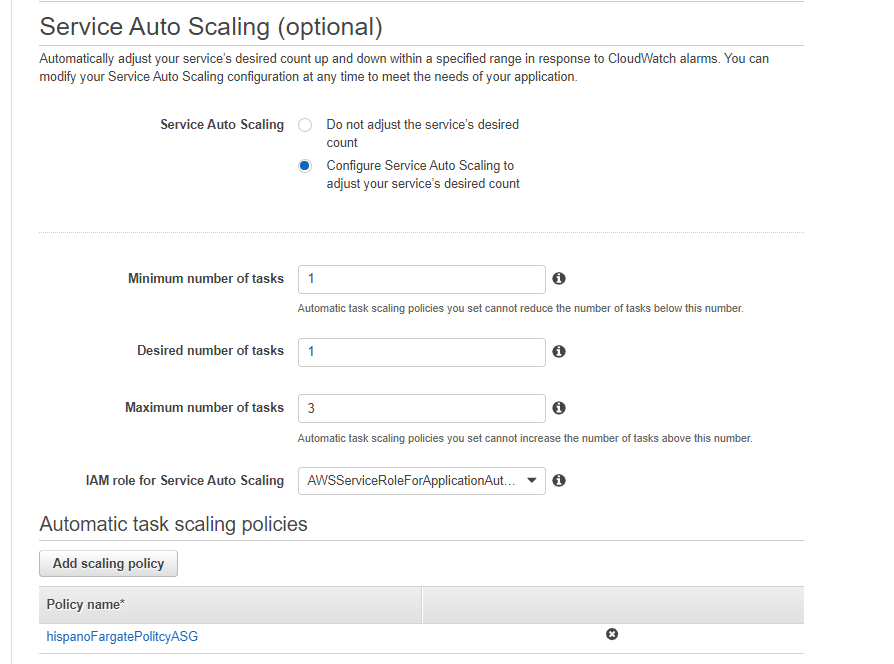


\*Lo minimo para fargate es 1GB, y para servicio de Spring son 500MB (si fuese en una EC2 se podría configurar, acá queda como 1GB).

Además hay que crear un **service** (desde el clúster creado), con este se define cuantos contenedores lanzaremos (que task y cuantas de ellas).

“A service lets you specify how many copies of your task definition to run and maintain in a cluster. You can optionally use an Elastic Load Balancing load balancer to distribute incoming traffic to containers in your service.”

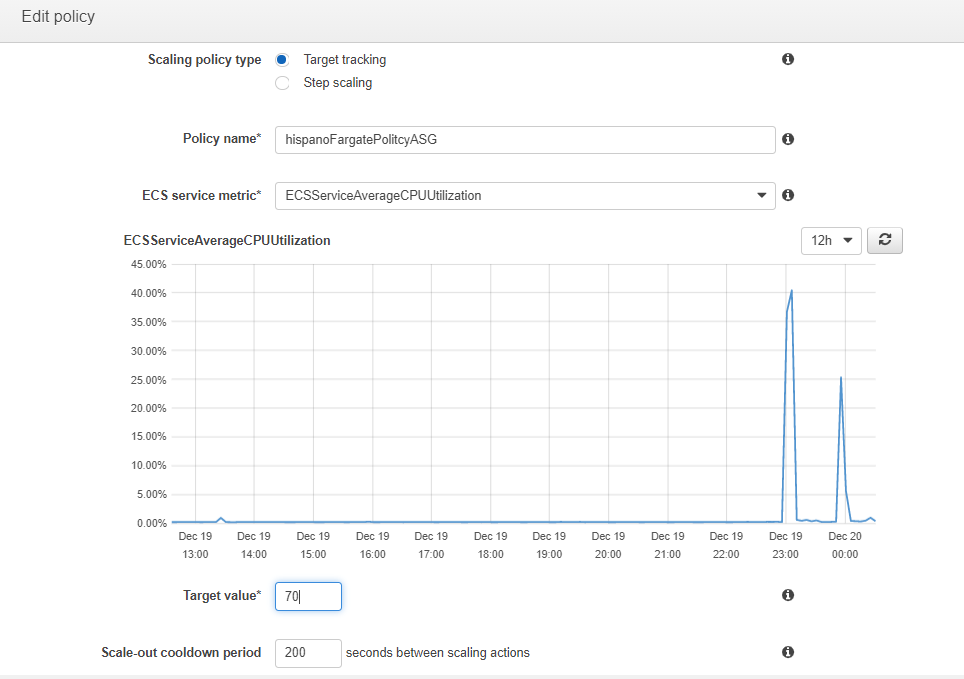
En el servicio definiremos la cantidad minima de tasks, la máxima, y la deseada (la deseada es por si queremos cambiarla en algún momento, pero esta se sobrescribirá cuando el ASG aumente o decrezca la cantidad de task en función de lo que definiremos en la alarma de CloudWatch).



En la política definiremos que métrica utilizar para escalar, y el target (valor) que tiene que llegar para que se ejecute.

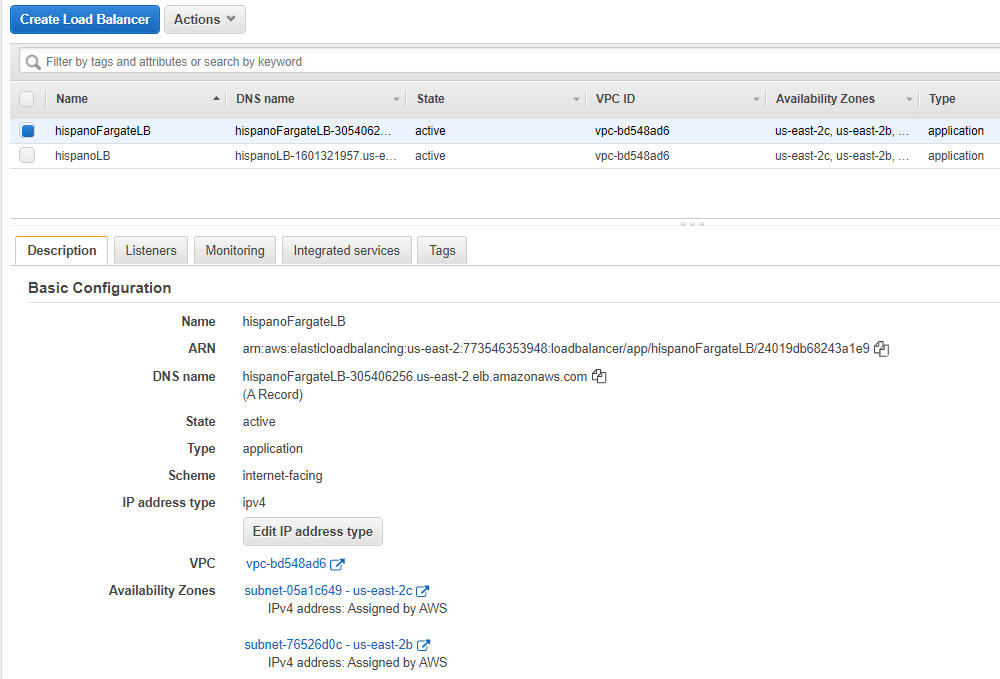
Tambien el tiempo Scale-out y Scale-in (es decir el tiempo que se bloquea el ASG para no iniciar mas de una o cerrar mas de una):

*“However, with a cooldown period in place, the Auto Scaling group launches an instance and then blocks scaling activities due to simple scaling policies until the specified time elapses. (The default is 300 seconds.)”*

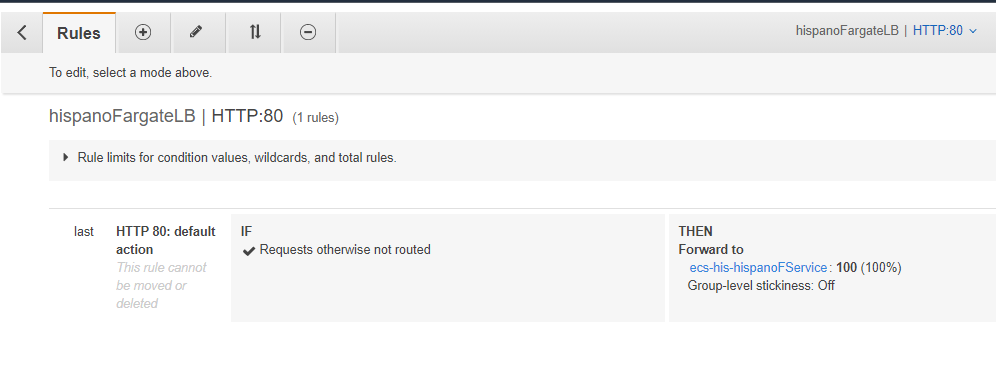


En nuestro caso haremos que escale a máximo tres tareas. Cuando pase el 70% de CPU, que cree la otra.

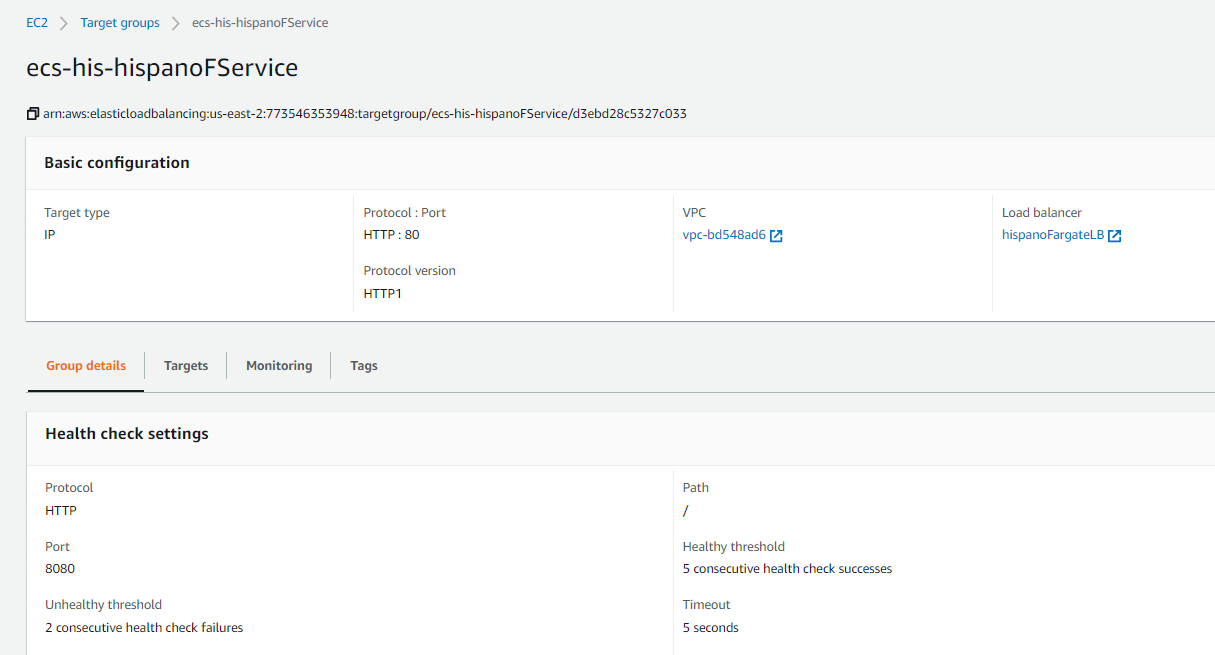
Se debe crear un Aplication Load Balancer (ALB) el cual dirija los request que llegan a las distintas tareas que se van creando.



El ALB también tiene un security group en el cual debemos especificar que esta abierto el puerto 80, (y también luego en otro SG, aceptar que este ALB tenga acceso a las task).

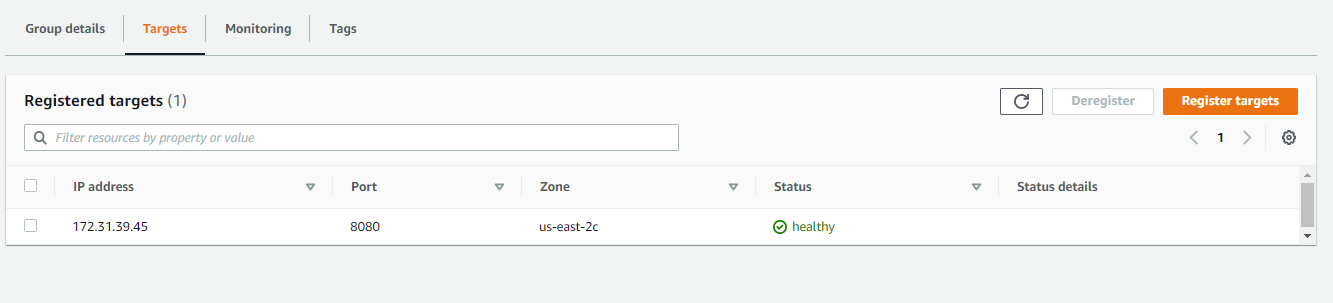


Este ALB tiene un listener que apunta los pedidos que llegan al puerto 80, hacia el target group que se crea (cuando se crea el load balancer).

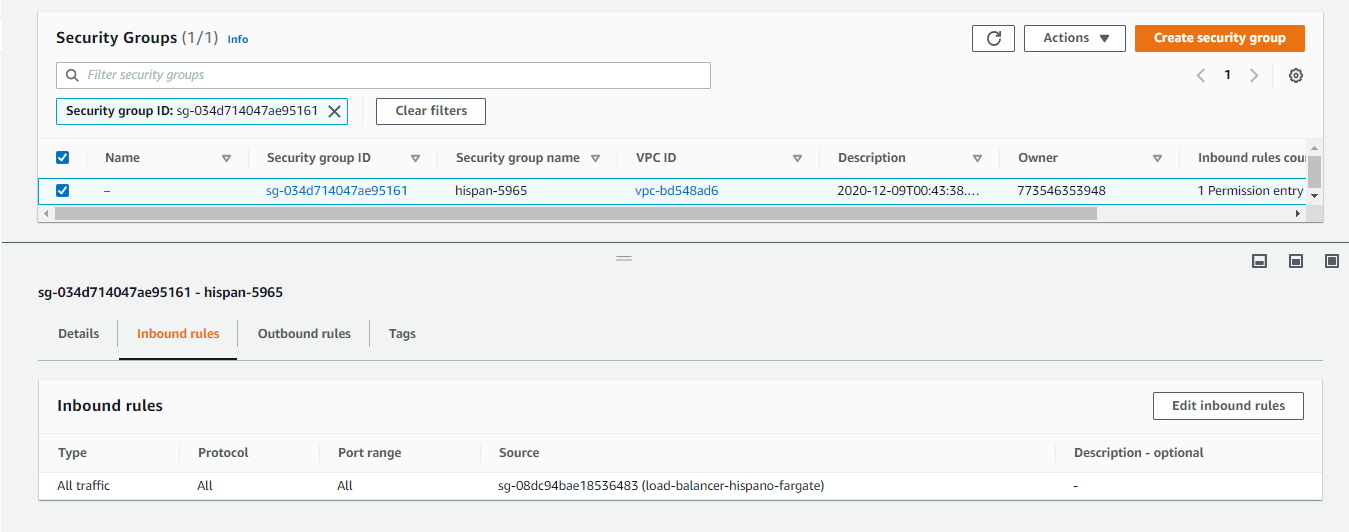


\*Hay que configurar el heath check sobre el puerto que está escuchando la task (8080).

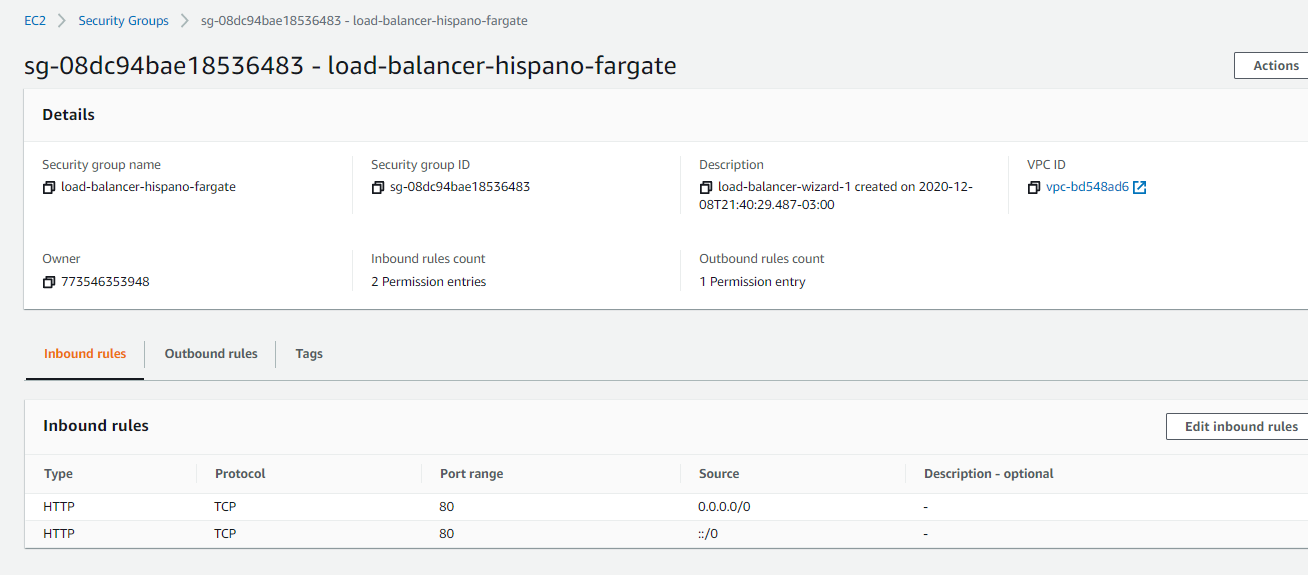
\*En la solapa target vamos a ver las task registradas en el momento, escuchando al ALB para responder. Si tenemos 3 task, tienen que aparecer las 3 aca, en este caso hay una sola deployada.



Security Group del Service:

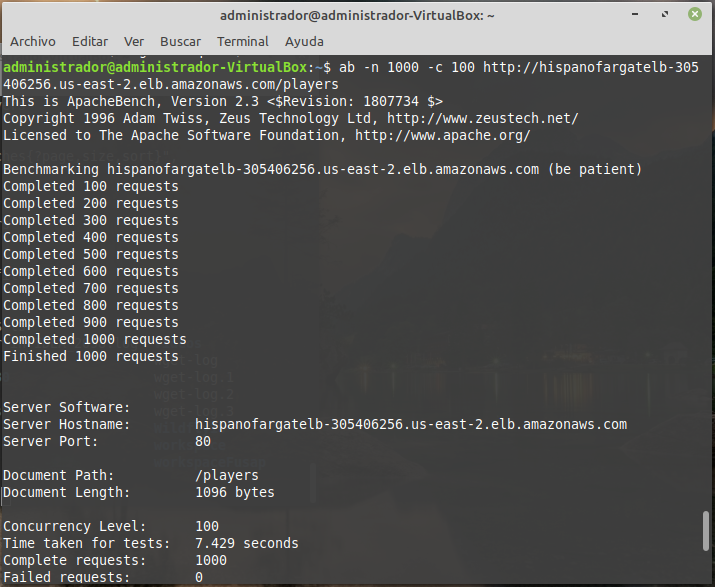
\*vemos todos los puertos abiertos, ya que el ALB los asigna dinámicamente para pegarle a los containers

Security Group del ALB:



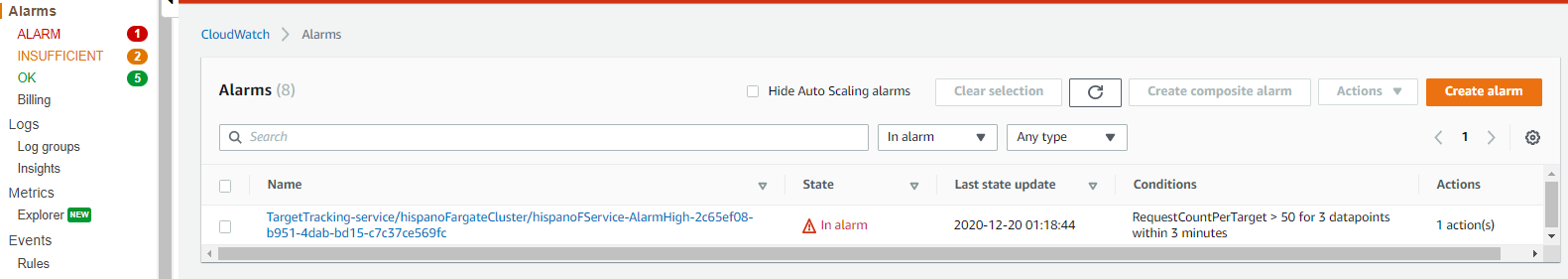
Para demostrar el correcto funcionamiento del autoscalling utilizaremos la herramienta **apachebench** (y cambiaremos el tipo de alarma a mas de 50 request por minuto, durante 3 minutos seguidos, a fines demostrativos rapidos)(también se podría dejar al 70% de utilización del CPU y forzar procesos complejos para que llegue al máximo):

*“RequestCountPerTarget > 50 for 3 datapoints within 3 minutes”*

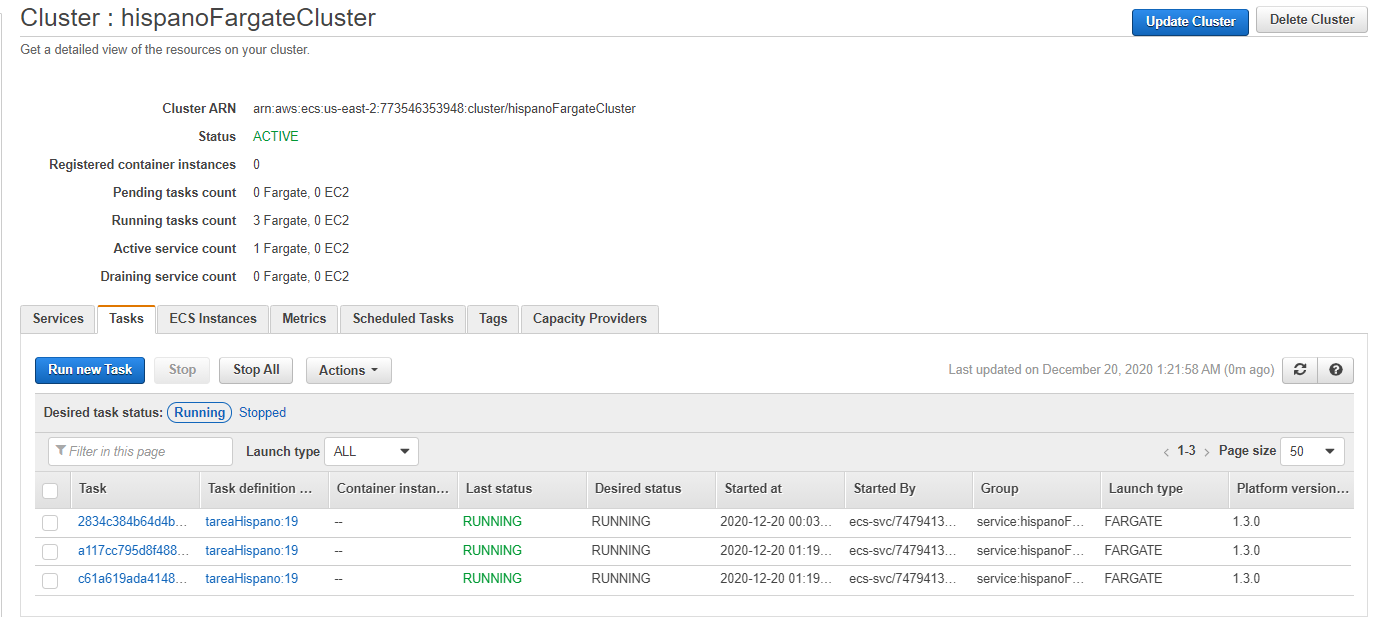


Ejecutando el comando en pantalla, hacemos 1000 request con concurrencia de 100 procesos al mismo tiempo, hacia la dirección del load balancer, elevando la carga del CPU a mas de 20%. Lo que deberá deployar una nueva task con nuestro conteiner.

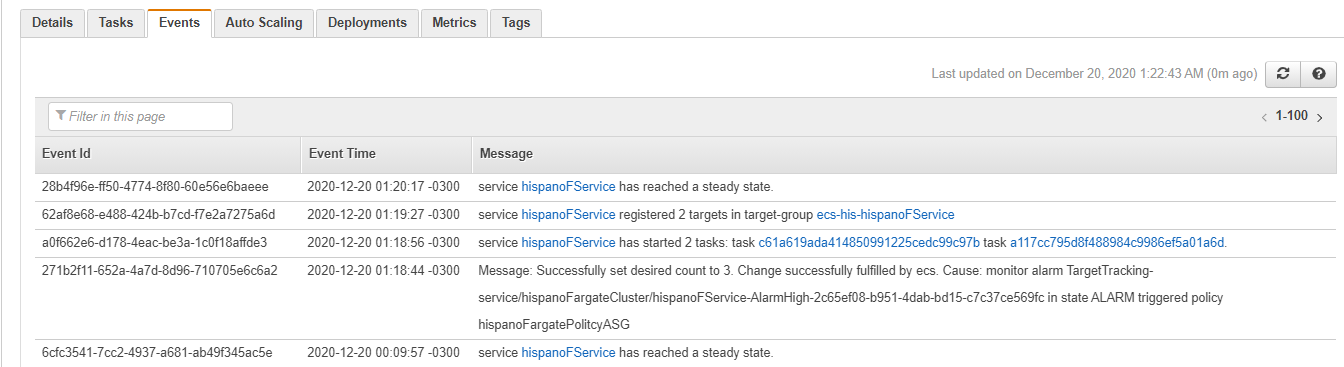
Vamos a cloudwatch y vemos a una alarma disparada en rojo:



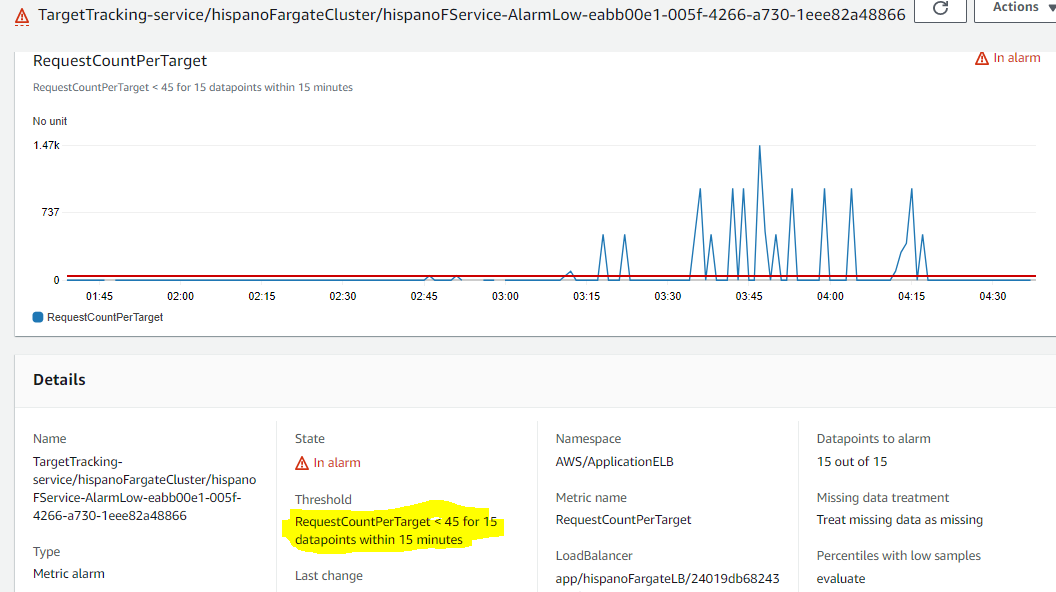
Y vemos que se levantan 3 instancias para atender la demanda:

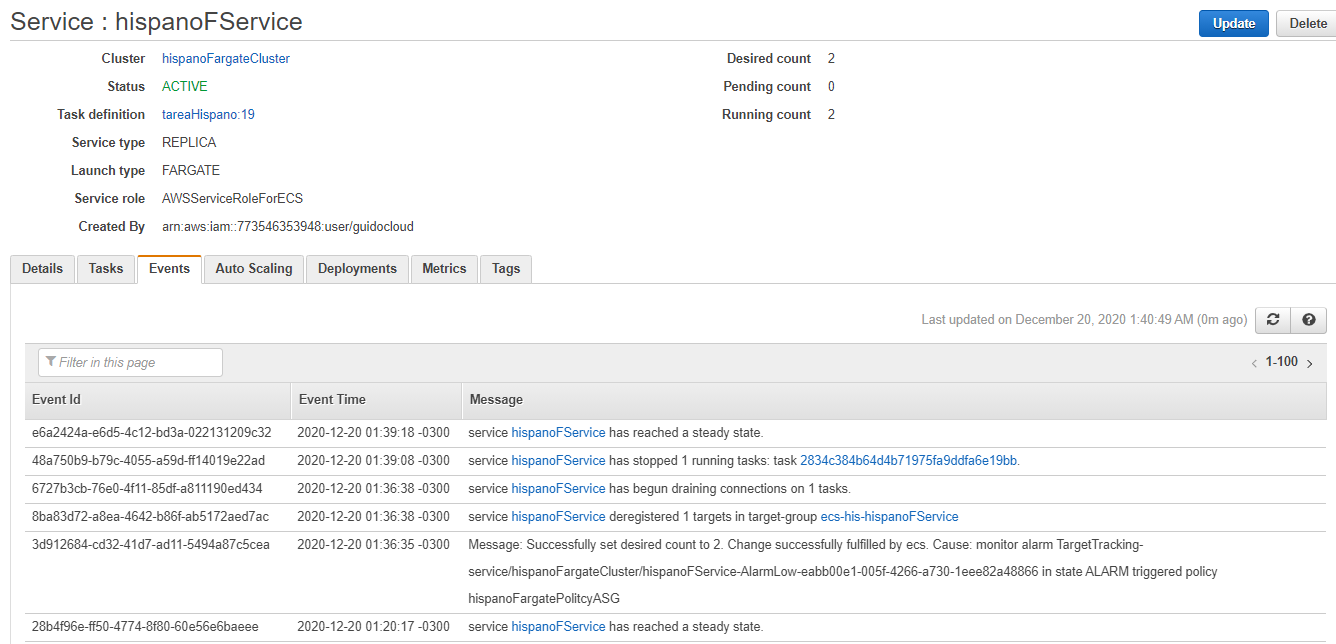


Vemos los eventos dentro del servicio:

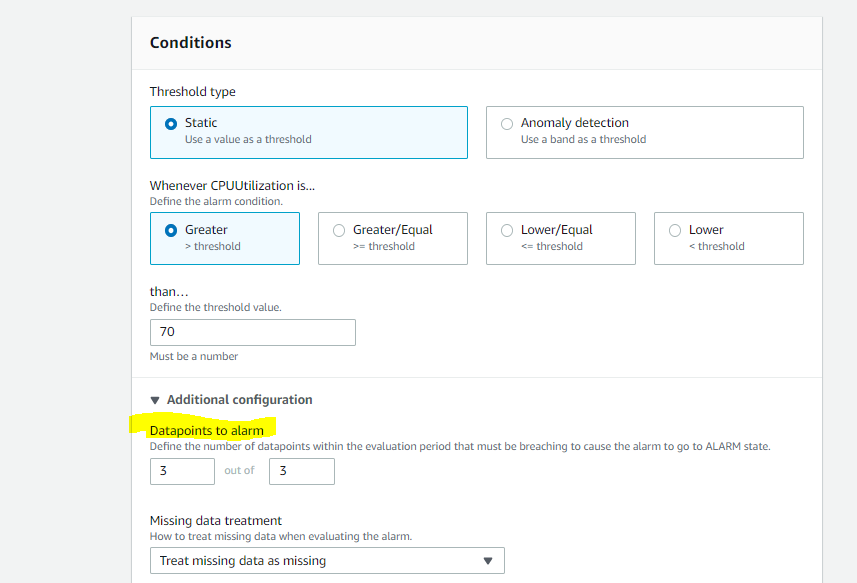


Y luego cuando los request bajen por los 200 configurados, se eliminaran las instancias automáticamente. (Primero una, luego otra).





Como observación, en alarmas se pueden configurar/editar, cuando se debe disparar, durante cuantos minutos sucede que excede el límite, si es una suma, un promedio, etc:



**Docker Images y Repositorio**

Para subir una imagen de docker hay que usar la awscli, actualmente la última es la 2, y se puede instalar tanto en Linux o Windows.

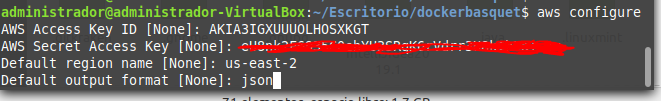
Se configura con aws configure y hay que pasarle las credenciales pertinentes (se toman de IAM managment, Security Credentials)

**Access key ID:**

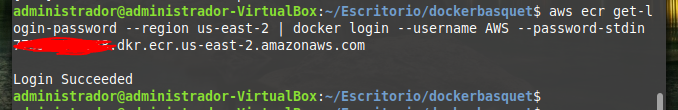
AKIA3IGXUUUOLHOSXKGT

**Secret access key:**

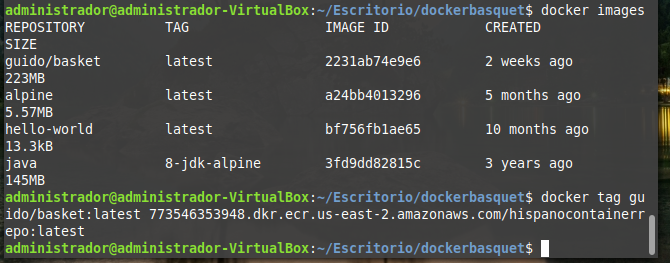
**esta en archivo guidocloud\_access.csv**



Para conectarse a ECR y administrar imágenes de docker:



(Tomar comando desde UI de **ECR** view push commands, acá lo que hace es tomar el password del primer comando que es hasta el pipe |, y luego dejarlo en la segunda parte para el logueo)

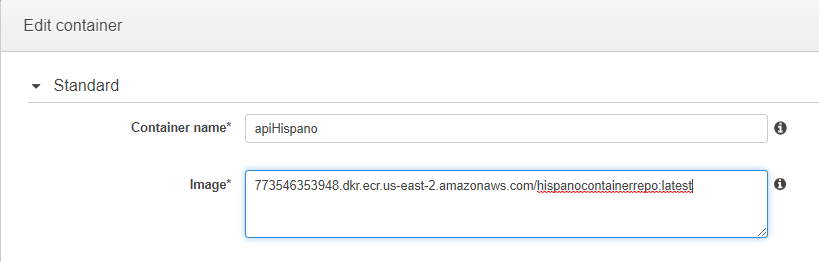


Tagueo imagen que ya tenía compilada con docker y luego ejecuto comando para pushear:

docker push 773546353948.dkr.ecr.us-east-2.amazonaws.com/hispanocontainerrepo:latest

(En este caso tome la imagen con nombre guido/basket y la subi como hispanocontainerrepo, pero podria tener nombres distintos)

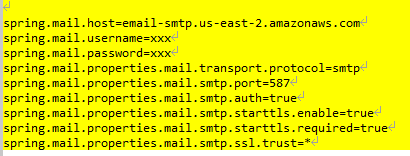
Luego en task apuntaremos a esta imagen que subimos:



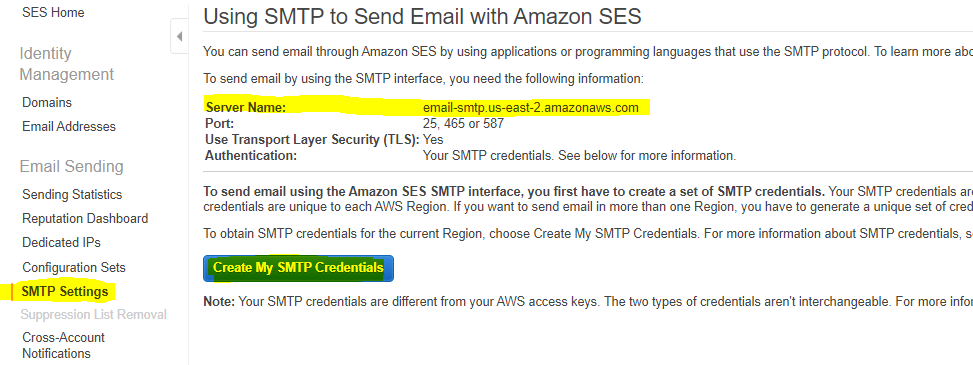
Para buscar el nombre de la imagen podemos buscarla **en ECR->Repositories.**

**Envio de Emails desde aplicación**

Utilizamos dependencia spring-boot-starter-mail en el .pom proyecto de Spring. Como también los parámetros de configuración en el application.properties.



El username y password son las credenciales que tenemos que crear desde la consola de AWS SES.

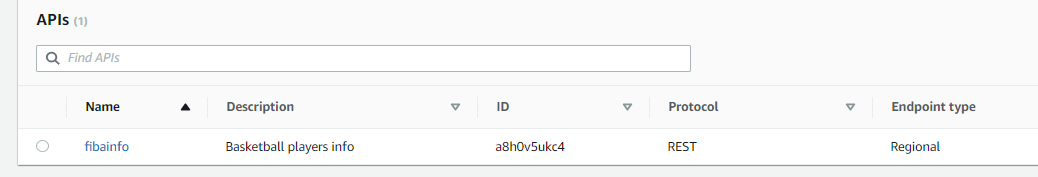


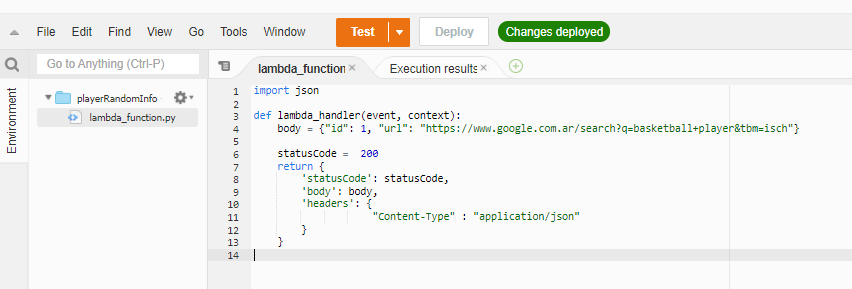
Las cuentas de email deben estar registrados para recibir mails, hasta pedir que el servicio de SES este productivo.

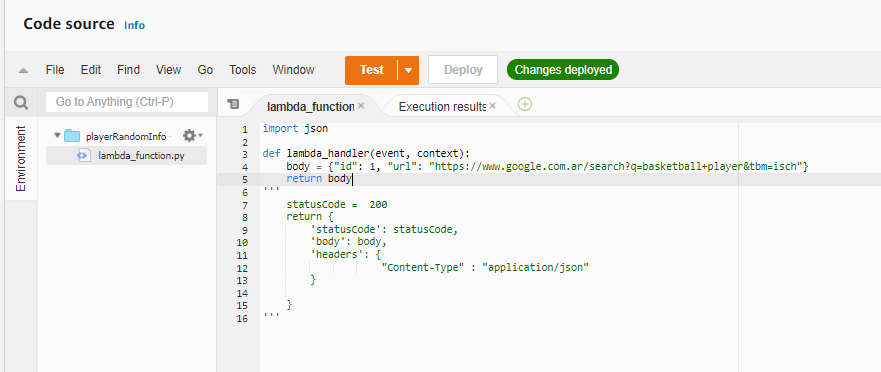
Para este caso hemos desarrollado un pequeño servicio que envía el email, y una clase model del mail, con el from, to (usuario registrándose), subject y body para utilizarla como log u para informar algo en particular.

**API Gateway / Lambda**

Devuelvo en un endpoint mio, con una función lambda alguna foto que encuentre en google, esto a modo practico.

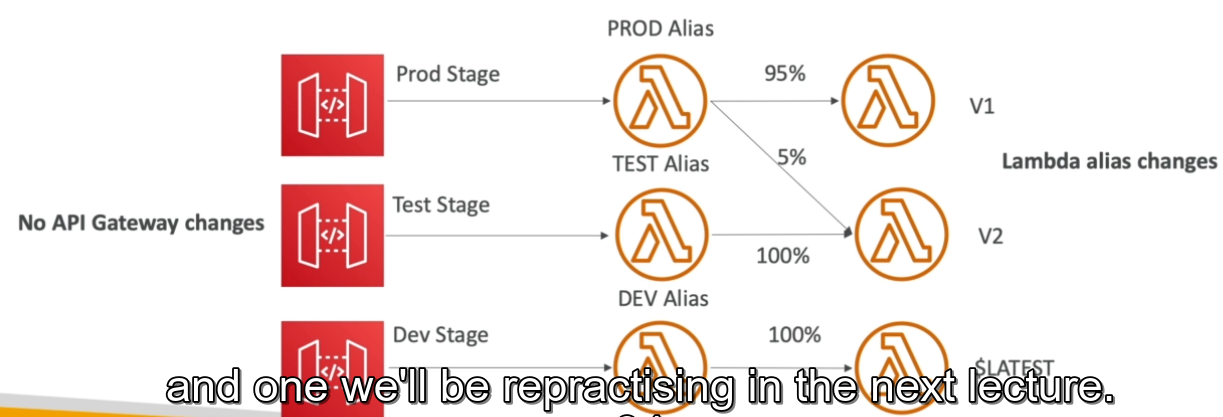






<https://a8h0v5ukc4.execute-api.us-east-2.amazonaws.com/dev/player>

Se podría pensar en un escenario de múltiples entornos para versionar y para derivar el tráfico que queramos a versiones de las funciones lambdas que queramos, como se ve en el gráfico, utilizando alias:



Sin embargo utilizaremos solo un endpoint a fines de practicidad para consultar desde la aplicación de un api hacia otra api, utilizando la librería reactiva Webclient en vez del restTemplate.

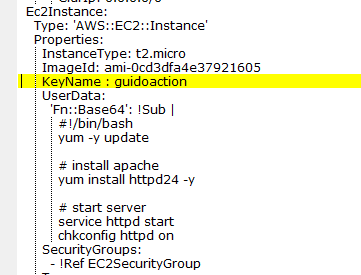
<https://github.com/chuchip/webClientExample>

Con esto podremos realizar tanto peticiones sincrónicas como asincrónicas de ser necesario, permitiendo que continúe el flujo del programa sin bloquearlo en caso de ser necesario.

En el ejemplo veremos una clase por dominio al que nos comunicaremos, y el dto es bien sencillo, un id y una url que será la respuesta de esta api.

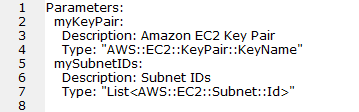
**CloudFormation**

Se le puede pasar al código del stack una key ya creada para poder acceder a la ec2 que querramos, ejemplo:

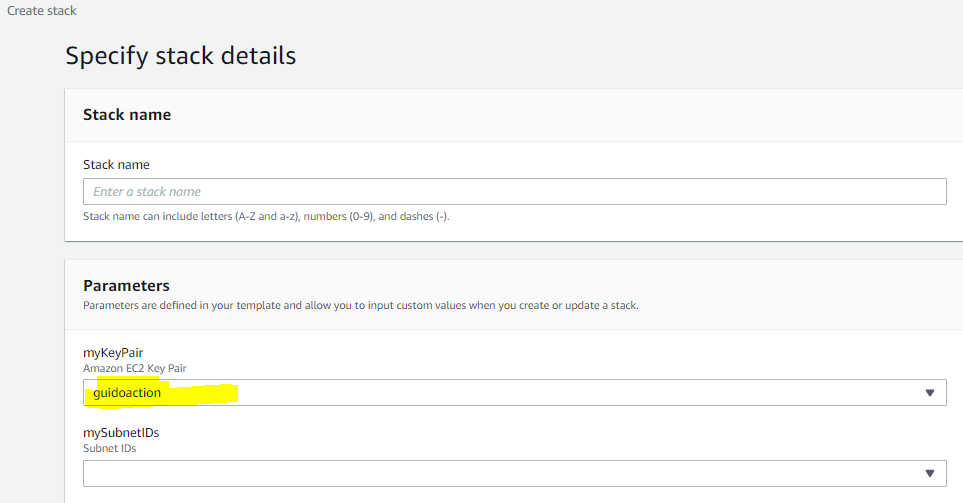


\*De esta forma queda hardcodeada en el template, también se puede poner para seleccionar por parámetro al crear un stack, dándole mas reusabilidad.

De la siguiente forma:



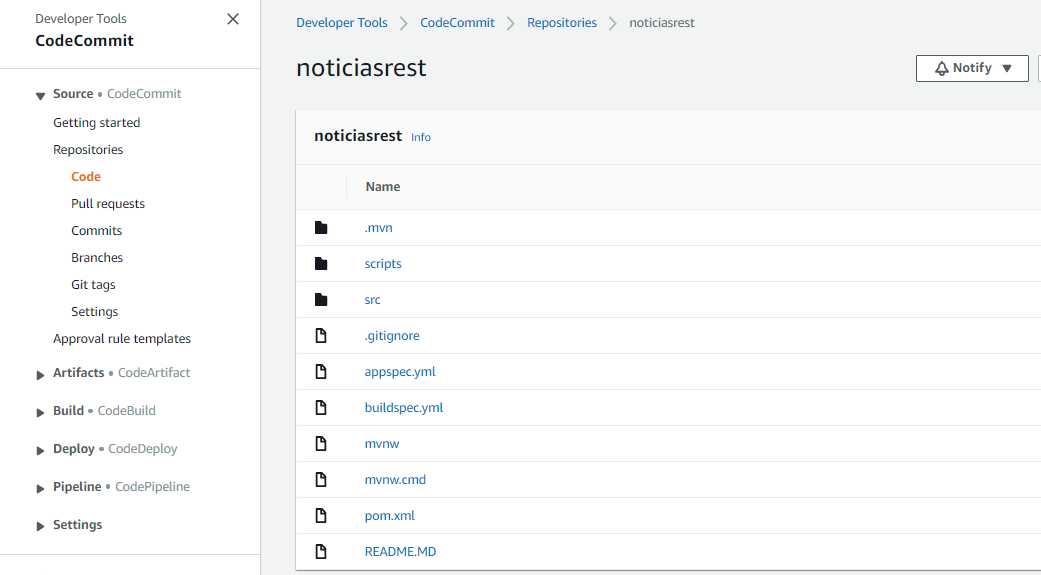
Y vemos:



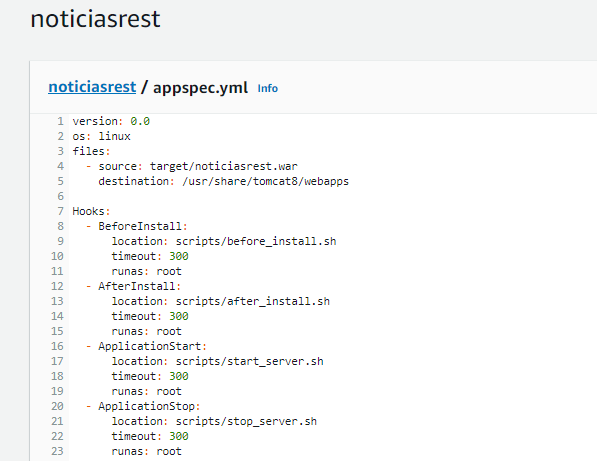
Agreando la key-value, podremos acceder via ssh con nuestra clave privada, ya que la instancia tiene asociada la clave pública esa que creamos (siempre que tengamos el puerto 22 abierto)

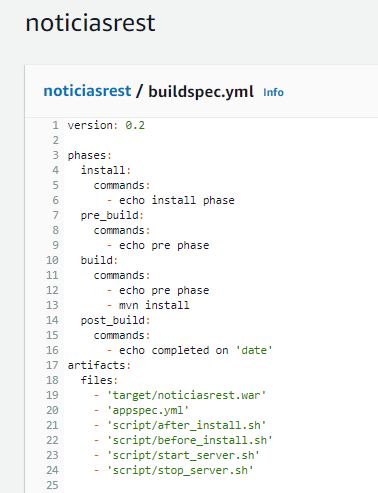
**CI/CD**

Repositorio con código que devuelve una noticia, en caso que en la base del proyecto basket no tenga ninguna cargada.



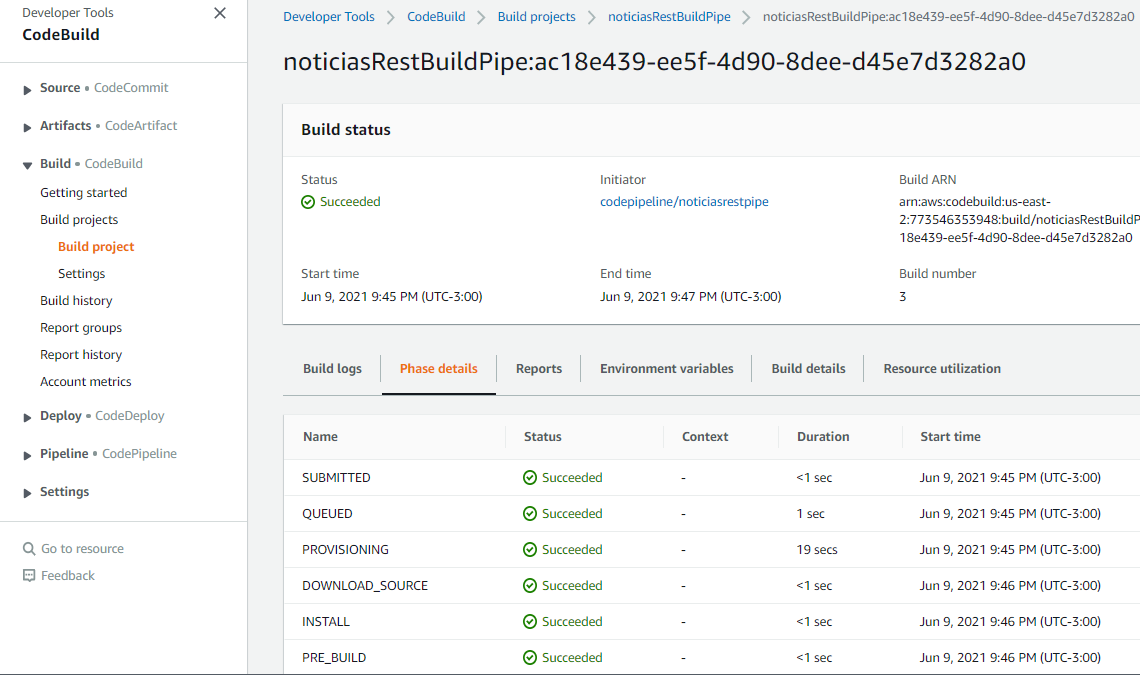
\*Aca podemos ver el buildspec.yml y el appsec.yml, junto a los scripts que se ejecutaran en casa fase.



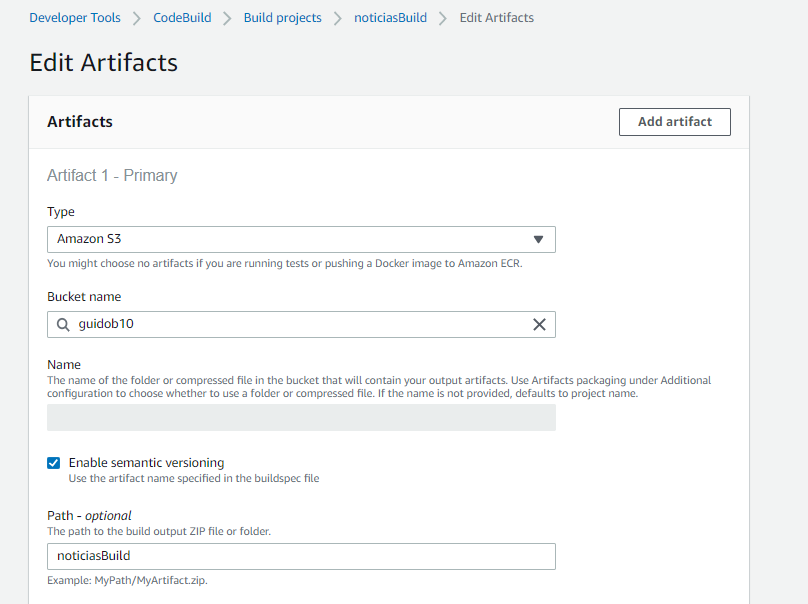


Guardara estos archivos en S3, que luego serán los input del build y deploy.

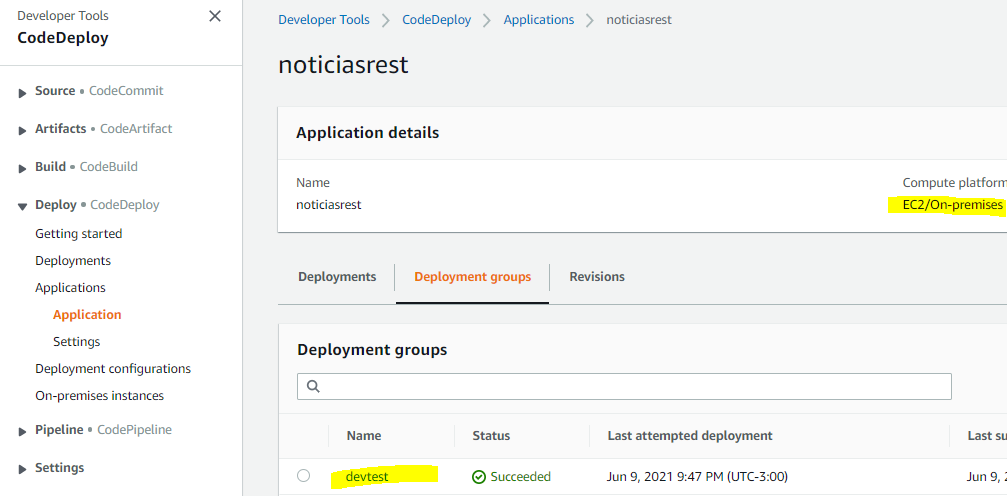
El build con sus fases



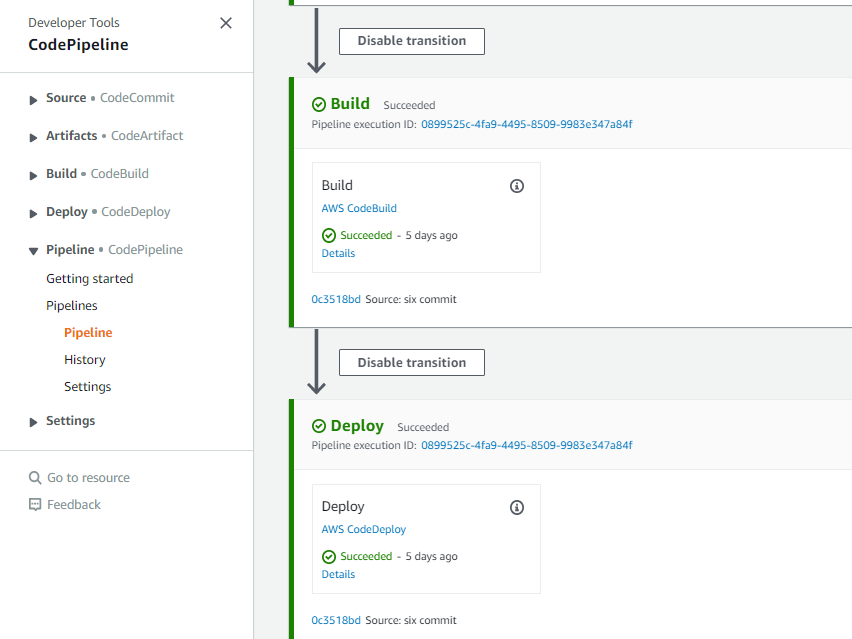
Se configure en el build, de que repositorio tomara el codigo, y donde quedara guardado lo generado.



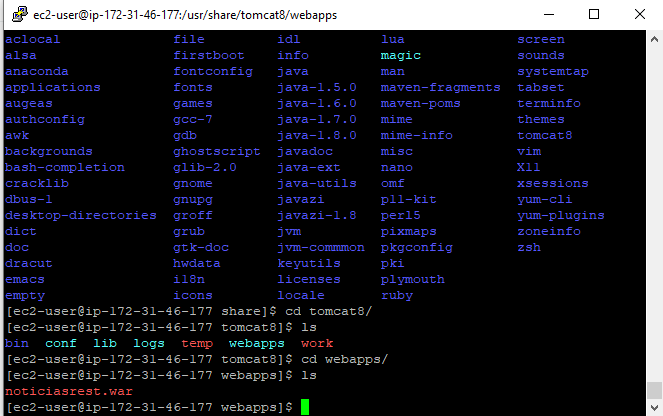
En codedeploy se selecciona en que group se deployara, en este caso sera la EC2 generada con cloudformation, con el tag devtest



El pipeline que se ejecutara al subir un código al repositorio.



App deployada desde pipeline a cloudformation ec2, al restartear se creara.



Agregar detalle en readme y template de cf. Ya no lo puedo probar arriba. Mostrar q deployo y fue

**Cosas que quedaron pendientes:**

.Cognito user pools para api gateway

.Logs importantes en Dynamo (los costos de writes son altos, solo para cosas importantes)

.Agregar disco para poder persistir logs de task en fargate

.Poner replica a la base (hacer reads contra una réplica)

.Poner cache para algunas querys a base

.Agregar un CloudFront edge, meter fotos en cache y restringir por país.

.ACM, certificado ssl para https, se pone en LB

.Agregar swagger para api