Hecho por: Omar Emilio Durán Cortez // 26780 (C8)

Examen Final

ICE191 Software Architecture / Cloud Computing

1. Describe los pros y contras de una arquitectura de software de microservicios.

En una arquitectura de software basada en microservicios las partes de un sistema están divididos en varias partes que trabajan de forma independiente. Es la forma contraria de hacer software que es tenerlo todo junto (monolítico). A continuación, listaré cosas buenas y malas que esto conlleva en ningún orden en particular.

Bueno: Como todas las partes del sistema están dividas en partes, si alguna de estas llega a fallar es fácil identificar cual fallo. Es mucho más fácil investigar el problema cuando se hay el servicio de notificaciones (por ejemplo) que investigar el problema cuando se nos cayó el monolito entero.

Bueno: Al estar todo separado, subir cambios a X ambiente es mucho más rápido porque no se está subiendo todo. Esto ayudar a reducir tiempos de caída (*down-times)* y a solucionar errores mucho más rápido cuando hay un error en el *deploy*. No estaría chido que falle del *deploy* luego de esperar 23 horas.

Malo: Se añade mucho *overhead* conforme crece el sistema. Aunque los microservicios son muy buenos en general, no son perfectos. Cada servicio tiene que estar siendo vigilado por logs, reglas, monitoreos. Adicionalmente, en algún lugar se tienen que levantar esos servicios sean contenedores, instancias de máquinas virtuales, *clusters* de instancias.

Estas son cosas que se van acumulando a lo largo del crecimiento del sistema que también se deben tomar en consideración para que el sistema funcione de forma óptima, no vaya a ser que le diste casi nada de uso de CPU al contenedor del

microservicio que más lo necesita. En un sistema monolítico solo hay que lidiar con un servicio.

Bueno: Existe independencia de tecnologías. Hace unas tareas se mencionó el caso de Khan Academy y su monolito de Python para su sitio web. En la arquitectura de microservicios eso no pasa. Un servicio puede utilizar A tecnologias y otro puede usar B tecnologías siempre y cuando sepan comunicarse el uno con el otro, así se pueden satisfacer necesidades distintas.

Un ejemplo sería que un servicio necesita analizar datos grandes científicos, entonces usar alguna librería científica de Python vendría bien, este entonces podría comunicarse con algún *framework* de *frontend* (los cuales usan TS o JS generalmente).

Bueno: Dos caras de la misma moneda. Aunque existe más *overhead* por tanto tener que manejar muchos microservicios, esto puede ser útil en las manos adecuadas porque cada servicio es escalable de forma independiente. Con esto cada uno puede trabajar con poco o mucho dependiendo de lo necesario. Esto no es posible en arquitectura de monolito.

Bueno: Al estar los códigos separados en múltiples *codebases* los desarrolladores tendrán un trabajo un poco más fácil en implementar nuevas funciones a cada servicio. Esto también facilita la distinción de trabajo a través de equipos/desarrolladores.

Malo: El testing se dificulta a nivel *end-to-end.* De acuerdo con FreeBootCamp (s.f) las pruebas se dificulta más conforme crece la aplicación porque se incrementa la cantidad de llamadas entre servicios que se tienen que probar, y esto es de forma exponencial. En un monolito todas las comunicaciones ocurren consigo mismo.

2. Explica la arquitectura de Publish and Subscribe

La arquitectura se suscripción y publicación es una forma de hacer comunicaciones entre software con software y software con personas (por medio de software también). Es un proceso asíncrono que permite a algún cuerpo central hacer un envió de información a otros múltiples interesados (ej.: enviar una notificación global por celular a todos los usuarios de una aplicación).

En comunicaciones entre componentes de software, es bueno tener comunicación asíncrona porque así hay una separación clara entre componentes, al hacerse asíncrona se evitan bloqueos para proceder con el flujo de trabajo.

De acuerdo con documentación oficial de Azure (s.f), esta arquitectura existe bajo la premisa de que usar un *queue* no escala bien cuando se trata de muchísimos usuarios (muchos recursos), y, que los usuarios solo están interesados en cierta parte de la información (usar un queue tradicional implica mandar X datos hasta llegar al que verdaderamente se quiere).

De acuerdo con la misma documentación existen 5 miembros del sistema. El primero es el publicador o enviado. Siendo aquel software/servicio/sistema que se quiere comunicar. El enviado utiliza un canal de entrada en vez de mandar los mensajes hacia un *message broker* (usaré los términos mensajes, objetos y datos como sinónimos en esta respuesta) en vez de hacerlo directamente.

Un *message broker* es el encargado en este patrón para enviar los mensajes por un canal de salida hacia los suscriptores. De acuerdo con IBM (s.f). este es el encargado de ser el entremedio de comunicación entre componentes de un sistema y se adapta a distintos protocolos, o sea que dos sistemas que no se conocen pueden comunicarse entre sí siempre y cuando sepan comunicarse con este (similar a una API). Los *brokers* validan, almacenan y enviar mensajes a los suscriptores.

El suscriptor es otra cosa interesante aquí, este se suscribe a los tópicos de su interés por medio de *endpoints*. Un tópico por así decirlo es el lugar donde el *broker* mete los mensajes, ahí es donde los suscriptores interesados en su información los reciben de forma relativamente inmediata en comparación a los queues donde se tienen que enviar todos los mensajes previos.

3. Crea una Lambda + API Gateway que reciba como parámetro un rango de fechas con formato YYYYMMDD y regrese la cantidad de días, horas y minutos entre dichas fechas.
    a. Por ejemplo: http://elamorlopuedetodo.com/rango/20230303-20230404
    b. La respuesta debe ser el URL del end point y los ARN de los recursos creados

URL (fecha incluida): http://2xu8nyvink.execute-api.us-east-1.amazonaws.com/dev/rango/20001226-20230429
ARNS de recursos:

- Lambda: arn:aws:lambda:us-east-1:292274580527:function:rango_fechas_omar
- Método GET de API: arn:aws:execute-api:us-east-1:292274580527:2xu8nyvink/*/GET/rango/*
- ARN API GateWay : arn:aws:execute-api:us-east-1:292274580527:2xu8nyvink

Mi cumpleaños es el 26 de diciembre, en un año no bisiesto hay 365 días. Entonces cuando sea el 26 de diciembre del siguiente año a ese (también hacia otro año no bisiesto) ocurrieron 365 días. Por lo tanto, el rango de fechas inicia en el momento de la media noche del primer día, y, acaba al ocurrir el último segundo del último minuto de la última hora del día anterior al de la fecha final.

Entonces, el rango entre un día por ejemplo 5 de abril y el día 7 de abril es 2 días o 48 horas o 2880 horas o 172800 segundos. Así es como tomaré en cuenta los rangos de fechas.

Nota: Usé la librería “datetime” de Python para hacer las cuentas porque las hace como especifiqué en los párrafos de ahí arriba.

Comenzando desde el final, este es el URL de la API
Ok, ahora de vuelta al inicio, el procedimiento es muy parecido al de mi tarea 6 por no decir igual con cambios a los nombres de cosas y el archivo de python que usa la lambda

Comando 1:

```
aws lambda create-function \
    --function-name rango_fechas_omar \
    --runtime python3.9 \
    --zip-file fileb://lambda_rango_fechas_omar.zip \
    --handler main.lambda_handler \
    --role arn:aws:iam::292274580527:role/lambda_ice191
```

Esto crea una lambda, se le dio nombre, *runtime* de Python 3.9, el archivo zip contiene solo el script de python porque no usé librerías ajenas a boto (ni boto en realidad), el *handler* se le dio el nombre de archivo y función son su sintaxis correspondiente (más detalles en tarea 5 y 6). Y, el rol de lambdas para la clase.

Comando 2:

```
aws apigateway create-rest-api --name rango_fechas_omar
```

Este comando crea la API en gateway, se le fio nombre y la ID está ahí abajo
ID API: 2xu8nyvink

Comando 3:

```
aws apigateway get-resources --rest-api-id 2xu8nyvink
```

Este comando lista los recursos de la API, se debe hacer para conseguir la ID del recurso raíz.
ID root: s4mgqrr6ra

Comando 4:

aws apigateway create-resource --rest-api-id 2xu8nyvink --parent-id s4mgqrr6ra --path-part rango

Este comando crea un recurso hijo para el padre, se llamará rango porque lo considreo un nombre apropiado.
ID rango: c58vxh

Comando 5:

aws apigateway create-resource --rest-api-id 2xu8nyvink --parent-id c58vxh --path-part {dateRange}

Este comando crea un recurso proxy para el recurso "…/rango/", aquí el usuario pondrá el rango de fechas cuando escriba la solicitud.

Explicación de APIs proxy de mi tarea 5:

Este comando es para crear un recurso en nuestra API, es necesario porque ahí se alojará el endpoint, necesita ID de la API, ID del recurso padre, y el lugar en donde poner el endpoint, el valor de ahí es para crear una API proxy. Una API proxy hará que la lambda según documentación oficial de AWS (s.f) tenga acceso al cuerpo, headers y tipo de la solicitud, necesario para cumplir el ejercicio 3 y gran parte de este ejercicio también.

ID recurso proxy: 0q7l1b

Comando 6:

aws apigateway put-method --rest-api-id 2xu8nyvink --resource-id 0q7l1b --http-method GET --authorization-type NONE

Este comando le pone al recurso proxy un método GET, esto es porque solo se obtendrá información de la API, no se le dará nada. Como no se pidió que se le diera forma de autorización no se le puso.

Comando 7:

```
aws apigateway put-integration \
    --rest-api-id 2xu8nyvink \
    --resource-id 0q7l1b \
    --http-method GET \
    --integration-http-method POST \
    --type AWS_PROXY \
    --uri "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:292274580527:function:rango_fechas_omar/invocations"
```

Este comando creó la integración entre recurso del gateway API y la lambda. Necesitaba ID de la API, ID del recurso, método HTTP que se quiere crear integración, el método de integración (siempre es POST), el tipo de integración (debe de ser PROXY para hacer funcionar el recurso PROXY), y, el URI con esta sintaxis (obtenido de mi tarea 5)

En URI (identifcador del recurso) va la siguiente sintaxis:

arn:aws:*nombreDeServicioQueLlamaLaLambda:region*:lambda:path/2015-03-31/functions/arn:aws:lambda:*region*:*IDdeUsuario*:function:*NombreDeLambda*/invocations

Comando 8:

```
aws lambda add-permission \
    --function-name "rango_fechas_omar" \
    --statement-id autorizadorDeLaLambdaDelExamenDeOmar \
    --action lambda:InvokeFunction \
    --principal apigateway.amazonaws.com
```

Este comando le da permiso a la API de API gateway a invocar a la lambda. Necesita el nombre de la lambda, la acción que hará (en este caso invocar) y especificar qué servicio de AWS llamará a la lambda. Statement es como un comentario, o sea opcional.

Comando 9:

```
aws apigateway create-deployment \
    --rest-api-id 2xu8nyvink \
    --stage-name dev \
    --description "deploy de API del examen de OMAR"
```

Como pasó final se deployea la API, necesita la ID de la API, algún nombre para el stage del deployment (el nombre es libre), y de preferencia alguna descripción.

¿Y el código del paso 1?

Aquí está:

```
import re
from datetime import datetime #Libreria para restar fechas


# Lamda hanlder es la funcion prinipal que corre mi lambda


def lambda_handler(event, context):
#Se necesita una respuesta de tipo json con estos 4 valores en APIS proxy para su funcionamiento correcto
baseResponse = {
```

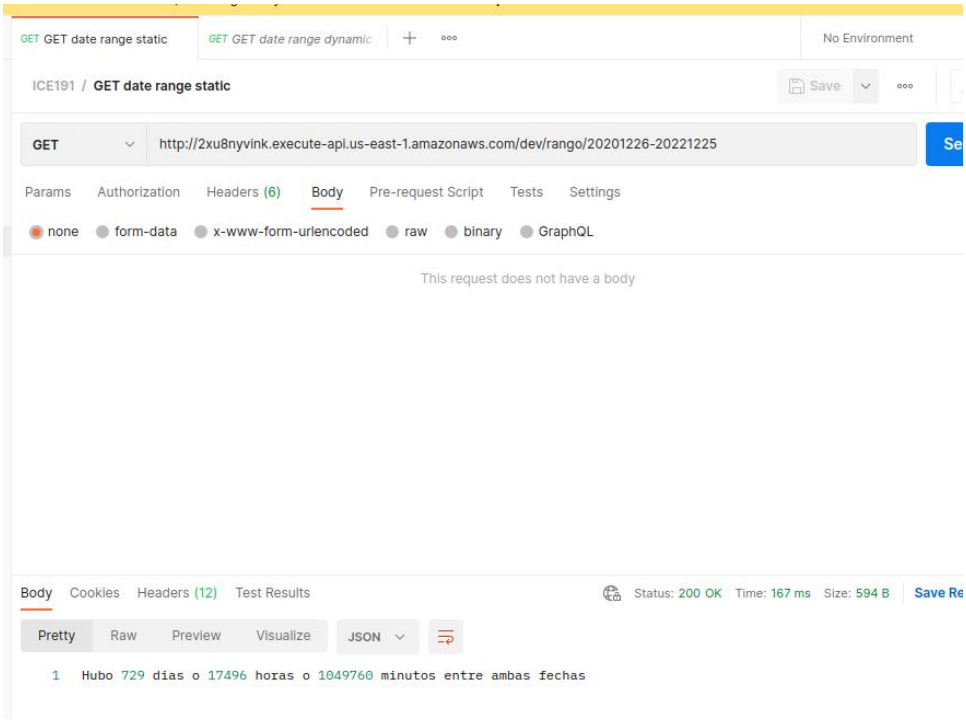
```
"isBase64Encoded": False,
"statusCode": 200,
"headers": { "uuuh": "ya vamonos no?!"},
"body": {}
}


# La fecha se encuentra en los path parameters con la llave del recurso proxy de la API
dateRange: str = event['pathParameters']['dateRange']


# Revisar que el usuario llamó a la API siguiendo mi sintaxis deseada
# usando una regex (YYYYMMDD-YYYYMMDD es lo que quiero)
if(re.match("[0-9]{8}-[0-9]{8}", dateRange)):
# Separar y convertir la fecha de string a objetos fecha
startDateStr, endDateStr = dateRange.split('-')
startDate = datetime.strptime(startDateStr, '%Y%m%d')
endDate = datetime.strptime(endDateStr, '%Y%m%d')


#La fecha menor debe de ser la izquierda en esta API
if(endDateStr < startDateStr):
baseResponse["statusCode"] = 500
baseResponse["body"] = "Fecha final es menor que la de inicio, favor de escribir primero la fecha inicial"
else:
# Hacer calculos de fechas
days = (endDate-startDate).days
hours = days * 24
minutes = hours * 60
answer = f"Hubo {days} dias o {hours} horas o {minutes} minutos entre ambas fechas"
baseResponse["body"] = answer
else:
baseResponse["statusCode"] = 500
baseResponse["body"] = "Rango de fechas no sigue la sintaxis YYYYMMDD-YYYYMMDD"


return baseResponse
```

Funciona el código de la siguiente forma. Todos los retornos regresan un objeto que necesitan las APIs proxy para funcionar, debe incluir cuerpo, encabezados, código de estado, y especificar si se encodifica en base 64 (repito de nuevo, más detalles en mis tareas 5 y 6.

El procedimiento es verificar si se llamó el *endpoint* con la sintaxis correcta del rango de fechas, luego las separa en dos y las convierte en objetos de fechas, luego las resta, y al final regresa los valores solicitados.

Puede fallar y regresar código 500 si: no se escribe bien la fecha (que se verifica con una REGEX), o, la fecha de la derecha es menor que la de la izquierda.

Hay copia de este script en mi repo de la materia y en mi sitio web de la materia.
Imagen con resultados entre el mi cumpleaños del 2020 y la navidad del 2022



Cabe mencionar que como la respuesta es texto y no un JSON con el valor de respuesta, algunos browsers como Firefox mostrarán un error al llamar al endpoint. Esto se debe al encabezado de “content-type” configurado como JSON, a pesar del error, pedirle al browser que muestre el “Raw content” mostrará una respuesta como la de la imagen.

Ese *header* configurado diferentemente es la siguiente respuesta.

4. Replica la respuesta anterior, pero en este nuevo end point regresa HTML, creando un dynamic web site.

Nota: Para facilitar mi trabajo y reducir la complejidad de la respuesta, este *endpoint* será un *endpoint* nuevo en la misma API de la respuesta anterior. Favor tomar eso como base/contexto.

URL (fecha incluida): https://2xu8nyvink.execute-api.us-east-1.amazonaws.com/dev/rango-dinamico/19221226-20221225

ARNs

- Lambda: arn:aws:lambda:us-east-1:292274580527:function:rango_fechas_omar_dinamico
- API gateway: arn:aws:execute-api:us-east-1:292274580527:2xu8nyvink
- Método GET de API: arn:aws:execute-api:us-east-1:292274580527:2xu8nyvink/*/GET/rango-dinamico/*

El primer paso es crear una lambda nueva. Pero ahora explicaré su código nuevo al inicio.

```
import re
from datetime import datetime #Libreria para restar fechas


# Lamda hanlder es la funcion prinipal que corre mi lambda


def lambda_handler(event, context):
#HTML STRING: Aquí irán unos strings con html
htmlHeadString: str = "<!DOCTYPE html><html><head><title>Rango de fechas
dinámico</title></head><body><p>"
htmlFooterString: str = "</p></body></html>"
htmlReturnString: str = ""


#Se necesita una respuesta de tipo json con estos 4 valores en APIS proxy para su funcionamiento
correcto
baseResponse = {
"isBase64Encoded": False,
```

```
"statusCode": 200,
"headers": {
"uuuh": "ya vamonos no?!",
"content-type": "text/html"
},
"body": {}
}


# La fecha se encuentra en los path parameters con la llave del recurso proxy de la API
dateRange: str = event['pathParameters']['dateRange']


# Revisar que el usuario llamó a la API siguiendo mi sintaxis deseada
# usando una regex (YYYYMMDD-YYYYMMDD es lo que quiero)
if(re.match("[0-9]{8}-[0-9]{8}", dateRange)):
# Separar y convertir la fecha de string a objetos fecha
startDateStr, endDateStr = dateRange.split('-')
startDate = datetime.strptime(startDateStr, '%Y%m%d')
endDate = datetime.strptime(endDateStr, '%Y%m%d')


#La fecha menor debe de ser la izquierda en esta API
if(endDateStr < startDateStr):
baseResponse["statusCode"] = 500
baseResponse["body"] = "Fecha final es menor que la de inicio, favor de escribir primero la fecha
inicial"
else:
# Hacer calculos de fechas
days: int = (endDate-startDate).days
hours: int = days * 24
minutes: int = hours * 60
answer: str = f"Hubo {days} dias o {hours} horas o {minutes} minutos entre ambas fechas"
htmlReturnString = htmlHeadString + answer + htmlFooterString
baseResponse["body"] = htmlReturnString
else:
baseResponse["statusCode"] = 500
baseResponse["body"] = "Rango de fechas no sigue la sintaxis YYYYMMDD-YYYYMMDD"


return baseResponse
```

El código en sí es lo mismo, pero hay dos diferencias, uno es que ahora el body de la respuesta es una combinación de tres strings. Uno con la primera parte del HTML, otro

con la respuesta (rango de tiempo), y otra parte con la parte final del HTML. Eso lo agarra de las tres variables nuevas casi al inicio.

Lo segundo nuevo está en la parte de los encabezados, ahora hay uno nuevo que no es nuevo en realidad, sino que sobrescribirá el predeterminado de AWS que es el de 'content-type' el cual por defecto tiene valor de "application/json" y lo que quiero es "text/html" para que así mi browser reconozca la respuesta del endpoint como HTML y no un JSON.

Ahora los pasos, que cabe destacar que son muy parecidos a los de la pregunta anterior, pero sin la creación de la API porque se recicló esa y solo se hizo un nuevo endpoint)

Comando 1:

```
aws lambda create-function \
    --function-name rango_fechas_omar_dinamico \
    --runtime python3.9 \
    --zip-file fileb://lambda_rango_fechas_omar_dinamico.zip \
    --handler main.lambda_handler \
    --role arn:aws:iam::292274580527:role/lambda_ice191
```

Crear nueva función lambda con un nombre nuevo, pero aun así similar para facilitar su identificación por mi parte.

Comando 2:

```
aws apigateway create-resource --rest-api-id 2xu8nyvink --parent-id s4mgqrr6ra --path-part rango-dinamico
```

Crear recursos dentro del recurso padre de la API, este será rango dinamico porque me parece un nombre apropiado dado que debemos retornar HTML y eso hace dinamico el sitio.
ID rango-dinamico resultante: qb5ey5

Comando 3:

```
aws apigateway create-resource --rest-api-id 2xu8nyvink --parent-id qb5ey5 --path-part {dateRange}
```

Crear recurso proxy para el recurso creado en al paso pasado
ID recurso proxy resultante: i3xtrm

Comando 4:

```
aws apigateway put-method --rest-api-id 2xu8nyvink --resource-id i3xtrm --http-method GET --authorization-type NONE
```

Poner método GET al recurso proxy del paso anterior

Comando 5:

```
aws apigateway put-integration \
    --rest-api-id 2xu8nyvink \
    --resource-id i3xtrm \
    --http-method GET \
    --integration-http-method POST \
    --type AWS_PROXY \
    --uri "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:292274580527:function:rango_fechas_omar_dinamico/invocations"
```

Integración del recurso PROXY a la lambda

Comando 6:

```
aws lambda add-permission \
    --function-name "rango_fechas_omar_dinamico" \
    --statement-id autorizadorDeLaLambdaNumero2DelExamenDeOmar \
    --action lambda:InvokeFunction \
    --principal apigateway.amazonaws.com
```

Darle permisos a API gateway de utilizar la lambda de esta pregunta

Comando(s) 7:

```
aws apigateway delete-stage --rest-api-id 2xu8nyvink  --stage-name dev
```

```
aws apigateway delete-deployment --deployment-id mb0nrt --rest-api-id 2xu8nyvink
```

Borrar el stage y deployment del ejercicio 3

Comando 8:

```
aws apigateway create-deployment \
    --rest-api-id 2xu8nyvink \
    --stage-name dev \
    --description "deploy #2 de la API del examen de OMAR"
```

Crear un nuevo deployment para incluir ambos endpoints del ejercicio 3 y este (el 4)

5. Describe la diferencia entre los S3 object storage class standard, infrequent access y Glacier.

S3 tiene múltiples formas de almacenar información, estas están hechas para distintas necesidades, pero según su descripción oficial (s.f) los factores de decisión son respecto a frecuencia, costos, y “resiliencia”.

El tipo “standard" es el tipo “normal” por así decirlo. O sea que es data que se accede frecuentemente, ¿Qué tan frecuente es frecuente? Múltiples veces por día. Sus casos de uso recomendados son para aplicaciones de la nube, sitios dinámicos, videojuegos, análisis de datos. Tiene disponibilidad de 99.99%, y, es de baja latencia, o sea que cuando se solicita un objeto (información) almacenado ahí se puede acceder rápidamente por no decir al instante.

El acceso “infrequent” u oficialmente conocido como el Standard-Infrequent Access y está dirigido para gente que no accederá tan frecuentemente a la información almacenada pero que aun así la requiere rápidamente cuando lo vaya a hacer. Sus casos de uso recomendados son para almacenamiento de datos a largo plazo como respaldos. La disponibilidad aquí baja de 99.99% a 99.9%

Existe una variante de esta última llamada “One Zone-Infrequent Access” que en vez de estar almacenados los datos en tres zonas de disponibilidad como mínimo, solo se hace en una. Es más barato, por cierto.

Como últimos *tiers* se tienen a los de tipo “glacier” que están hechos para almacenaje a muy largo plazo a muy bajo costo. Según la misma descripción oficial (s.f). Los casos de uso son para datos que no serán tocados por semanas, meses o años, cosas hechas para archivar, como, por ejemplo: una digitalización de archivos físicos viejos en una empresa.

Existen tres tipos de “glacier”, el primero es “instant retreival” y como se menciona es de acceso rápido, sus casos de uso recomendados son contenidos generados por usuarios en internet o de medios de noticias como imágenes. La información se consigue igual de rápido que el tipo estándar pero la disponibilidad es la misma que de acceso

infrecuente. Adicionalmente se requiere que el objeto almacenado sea de 128Kb como mínimo, como referencia, este documento no llega a ese tamaño.

El otro tipo glaciar es el “flexible retreival”, de acuerdo con su descripción oficial antes este era conocido como el tipo “glacier” solamente. AWS dice que los datos almacenados aquí se acceden entre una y dos veces al año, o sea que no es para datos que se requieren de inmediato cuando surja la necesidad. Por esto mencionado, los tiempos para adquirir objetos de ahí rondan entre los minutos hasta las horas. La disponibilidad es la misma que de acceso infrecuente y está hecho para casos de uso similares a este, lo distinto es que en este tipo de S3, es más barato el almacenamiento.

El último tipo de glaciar es el “Deep archive”, aquí su intención es almacenar objetos por objetos que serán accedidos con muy baja frecuencia (como documentos para cumplir estándares o normas gubernamentales) por un largo tiempo como una década, por ejemplo. Se vende como alternativo a almacenamiento en cintas magnéticas, este tipo de almacenamiento consiste en almacenamiento en un medio más duradero que discos duros, y esto se refleja en el servicio el cual requiere hasta 12 horas para acceder a los archivos, similar a la lentitud de las cintas. Es el más barato de todos los tipos de almacenamiento.

6. Explica cómo hacer route de un subdominio de Route 53 a un static web site en S3.

Nota: La siguiente respuesta es la misma que la 2 de mi tarea dos con sus correcciones pertinentes, por lo tanto, se asume que

- El sitio web (que será el subdominio) ya existe
- El sitio web dominio ya existe

El primer paso fue ir a la documentación oficial de Route53 para ver cómo hacerlo con la consola de AWS

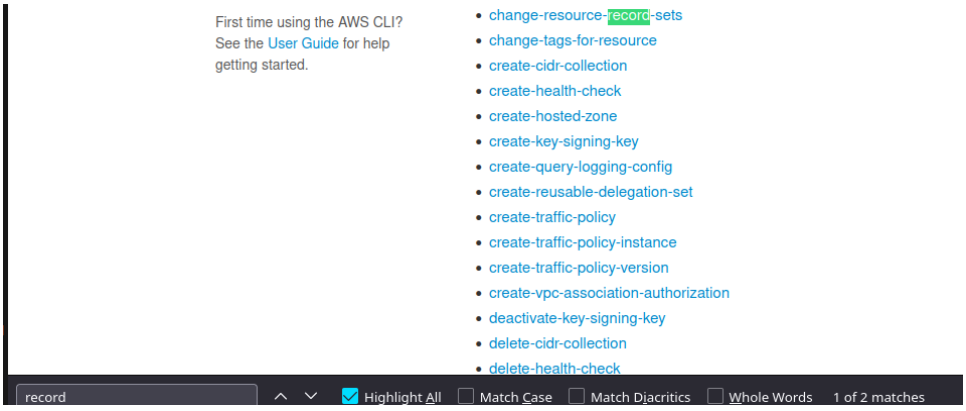
Configuring Amazon Route 53 to route traffic to an S3 Bucket

To configure Amazon Route 53 to route traffic to an S3 bucket that is configured to host a static website, perform the following procedure.

To route traffic to an S3 bucket

1. Sign in to the AWS Management Console and open the Route 53 console at https://console.aws.amazon.com/route53/.
2. In the navigation pane, choose **Hosted zones**.
3. Choose the name of the hosted zone that has the domain name that you want to use to route traffic to your S3 bucket.
4. Choose **Create record**.

Aquí dice que debo crear un récord. Aquí es cuando me moví a la documentación del CLI para buscar comando relacionados a récords



Las únicas dos menciones eran creación y listado de récords, por lo tanto, el de creación debía de ser el correcto

Sí es el comando correcto.

```
{
  "Comment": "optional comment about the changes in this change batch request",
  "Changes": [
    {
      "Action": "CREATE"|"DELETE"|"UPSERT",
      "ResourceRecordSet": {
        "Name": "DNS domain name",
        "Type": "SOA"|"A"|"TXT"|"NS"|"CNAME"|"MX"|"PTR"|"SRV"|"SPF"|"AAAA",
        "TTL": time to live in seconds,
        "ResourceRecords": [
          {
            "Value": "applicable value for the record type"
          },
          {...}
        ]
      }
    },
    {...}
  ]
}
```

Este *string* que acabo de copiar y pegar de la documentación del CLI (s.f) es un "resource record set" básico, usaré este porque solamente quiero redirigir el tráfico de mi cubeta a un subdominio, nada de cosas sofisticadas.

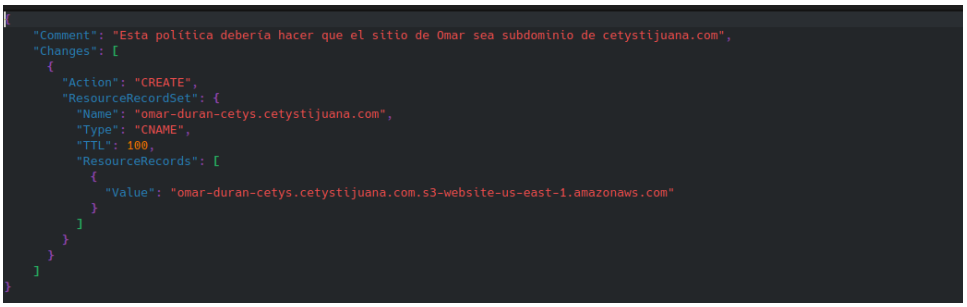
En “comment” va un comentario que describa el récord, es libre la decisión, en “Changes” va una lista de cambios, cada cambio va dentro de *brackets*, en “Action” va si se crea, borra o actualiza un récord, en “ResourceRecordSet” van unos *brackets* con datos respecto al récord.

En “ResourceRecordSet” en “Name” va el nombre del récord, o sea el URL **completo** al que se redirigirá el tráfico. En “Type” va el tipo de récord, aquí depende de lo que quieras hacer, si vas a redirigir a un recurso de S3 debe de ser A según la documentación del CLI, pero sí es un sitio web entonces esto no funcionará porque ya lo probe y dice que la dirección IPV4 es invalida (el tipo A redirige a direcciones IP), dado que será un subdominio deberemos usar CNAME. En TTL es el tiempo que vive el récord en cache antes de que los DNSs lo deban leer de nuevo.

¿Pero qué significa CNAME? De acuerdo con Cloudfladre, CNAME significa “Canonical Name”, un récord de tipo A asigna un dominio a una dirección IP, mientras que un récord de tipo CNAME asigna un subdominio a un dominio. O sea, que si tengo el dominio queso.com necesito un récord A para que se asigne a una IP que yo quiera, y necesito un récord CNAME si quiero ponerle un subdominio como omar.queso.com. Cambios a una dirección IP no afectan a un récord CNAME porque estos están vinculados al dominio padre y no a la IP, cambios de IP son asunto de un récord A.

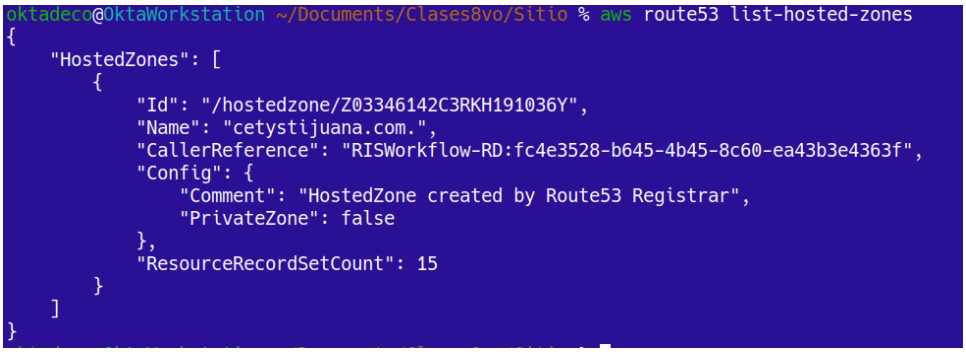
¿Ok pero que es un récord? Cuando digo récord me refiero a un récord de DNS (o archivo de zona), y esos, según Cloudflare (s.f) son archivos con múltiples instrucciones que especifican como se relaciona un dominio en la red a una dirección IP y que se deben hacer con las solicitudes. Un DNS contiene muchísimos de estos récords y funciona como directorio telefónico vinculando X dominio a Y dirección para que los usuarios no tengan que memorizarse IPs.

Ok, volviendo al tema de mi JSON …Dentro de “ResourceRecordSet” van unos *brackets* con un solo para llave valor, siendo el del lugar de origen del tráfico, el opuesto al “Name” de “ResourceRecordSet”.



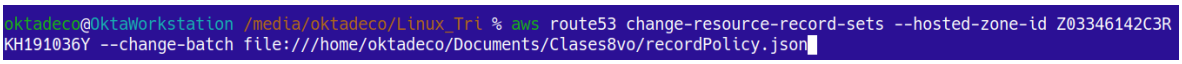
Así es como quedó mi política. En el caso de “Name” en “ResourceRecordSet” es importante destacar que debe coincidir con el nombre del dominio el final del nombre, o sea que si quiero un subdominio para lol.com, mi cubeta de S3 debe llamarse "NombreDeSubdominio.lol.com".

Ahora explicaré como es que lo subí.

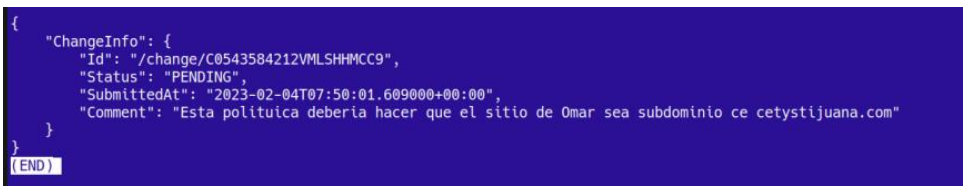


El comando para hacer cambios de récord requiere como parámetro el "hosted-zone-ID" este es un ID que según documentación de Route53 (s.f) aloja información respecto a récords de un dominio padre (en este caso el padre sería cetystijuana.com). Para conseguir este ID corrí el comando en la imagen, el cual lista en el servicio de route53, todas las "hosted zones" a disposición del usuario. Ahí, dentro de la única entrada presente estaba el ID que necesitaba.

Una vez con esta información pude correr el siguiente comando:



De aquí lo único que me falta explicar de este comando es el "–change-batch", es cual es el parámetro que contiene los cambios a hacer (el json mostrado)



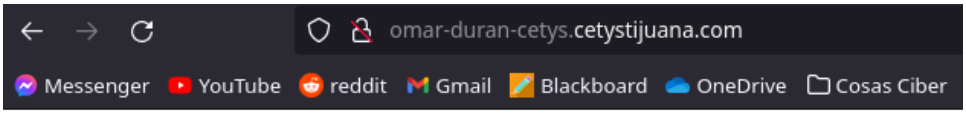
Al correrlo esto fue lo que me salió, dice “PENDING” porque es una solicitud y hay que esperar a que se hagan los cambios, esto ocurre porque gran parte de los flujos de las APIs de AWS son asíncronas, o sea que en una llamada simplemente haces una solicitud para que se haga algo, y eso significa que no necesariamente justo después. Pueden pasar milésimas, decimas, segundos u otros periodos de tiempo dependiendo de la acción que querías hacer.

```
oktadeco@OktaWorkstation ~/Documents/Clases8vo/Sitio % aws route53 list-resource-record-sets --hosted-zone-id Z03346142C3R
KH191036Y
```

Este comando nuevo ("list-resource-record-sets") muestra los récords de un hosted Zone ID indicado, en este caso del del dominio de cetystijuana.com.


```
        {
            "Name": "omar-duran-cetys.cetystijuana.com.",
            "Type": "CNAME",
            "TTL": 100,
            "ResourceRecords": [
                {
                    "Value": "omar-duran-cetys.cetystijuana.com.s3-website-us-east-1.amazonaws.com"
                }
            ]
        },
```


Y como se puede apreciar ahí sale mi récord con mi cubeta de S3 siendo apuntada por el subdominio mío de cetystijuana.com



Hola este es el sitio de Omar Duran

Éxito total, lo probe justo después de correr el comando y mi sitio ya era un subdominio del sitio padre.

7. Escribe el código en python para leer y borrar un mensaje de SQS

Nota: El siguiente código está hecho bajo la suposición de que el *queue* de SQS ya ha sido creado y está configurado para quien corra este script puede leer de tal.

```
import boto3
import time

def process_messages():
#Usaremos el cliente de SQS, es de más bajo nivel y permite hacer
# cosas más granulares, pero para fines prácticos y tomando en cuenta la
# difucltuad de este ejercicio, esto se puede lograr tanto con
# cliente como recurso
sqs_client = boto3.client("sqs")
# Obtener el url del queue de omar
queue_url = sqs_client.get_queue_url(QueueName='test_omar')['QueueUrl']
print(queue_url)

#Obtener el mensaje del url del queue de Omar y guardarlo en una variable, su recibo igual
print("Queue URL: {0}".format(queue_url))
message = sqs_client.receive_message(QueueUrl=queue_url)
#Message ["Messages"][0]["Body"] porque solo quiero visualizar el mensaje
# y no todo lo demas
messageStr = message["Messages"][0]["Body"]
print("Message received: {0}".format(messageStr))
receipt = message["Messages"][0]["ReceiptHandle"]

#Esperar un segundo para darse tiempo (yo como persona) para leer el mensaje
time.sleep(3)

#Borrar el mensaje recibido y enviarlo a una variable para rastreo
response = sqs_client.delete_message(
QueueUrl=queue_url,
ReceiptHandle=receipt
)
print("Message deleted: {0}".format(response))

process_messages()
```

8. Explica las diferencias entre SNS y SQS

SNS y SQS son dos servicios de AWS que son utilizados comúnmente para comunicaciones asíncronas entre servicios, componentes, programas, entre otros. De acuerdo con un artículo oficial de AWS (s.f) lo que diferencia a SNS de SQS es que SNS utiliza el paradigma de “publish and subscribe” (detalles más a fondo en su pregunta correspondiente) mientras que SQS se basa en *queues*.

¿Pero eso que significa? aquí los “publicadores” envían mensajes al tópico y los subscriptores (que por cierto también pueden ser *queues* de SQS, pero eso es una tangente) los reciben ya sea por HTTP, correo, notificaciones de teléfono, mensaje de texto, entre otros medios de comunicación. El suscriptor no hace nada al respecto mientras que en SQS aquel que quiera recibir información del *queue* debe leer del *queue* a mano. Un sistema automatizado ajeno a AWS también lo consideraré como a mano en este caso.

Otra diferencia, que es un poco más enfocada al servicio en sí que las arquitecturas, es el tiempo en el que vive la información, en SQS se puede vivir entre uno y catorce días de acuerdo con el FAQ de SQS, pero para SNS el tiempo para vivir (TTL) predeterminado es 4 semanas, más que lo que permite SQS.

Otra diferencia es cuánta gente recibirá el mensaje, en SNS ya se dijo que es pub-sub, entonces puede haber N suscriptores sin problema, pero en SQS no es así porque es un *queue*, y los *queues* son filas, y solo pasa un elemento a la vez. En SQS al adquirir un mensaje, se debe mandar una solicitud de retorno diciendo que el mensaje llegó correctamente, esto hará que se borre el mensaje.

Antes de borrar un mensaje en SQS, pero después de enviarlo existe un periodo de tiempo llamado “visibility timeout” que según documentación oficial el mensaje no puede ser visto por nadie más que tenga acceso a la fila a pesar de que no se haya borrado, por lo tanto, la comunicación es serial y no en paralelo. Si este tiempo expira se manda de vuelta al final de la fila. Y, si se hace la configuración respectiva se puede mandar a un *dead letter queue* los mensajes que sean enviados múltiples veces, pero nunca se les envía la solicitud de borrado, este *queue* muerto sería algún tópico de S3.

Así que, para fines prácticos, es mejor utilizar SNS cuando hay comunicación de 1 a muchos, y SQS para comunicación de 1 a 1. SNS es más para notificaciones hacia

personas, envió de mensajes (*broadcast),* comunicación hacia varias aplicaciones o servicios. Mientras que SQS sería mejor utilizado en situaciones como comunicar una aplicación/servicio con otro para tareas asíncronas, esto último no es para decir que SNS sea síncrono, sino que en SNS estos llegan lo más pronto posible, mientras que en SQS los mensajes deben esperar un tiempo determinado en una fila.

9. Explica el ciclo de TTD

Para contestar a esta pregunta me basaré en el artículo de Grant Steinfield de IBM (2020)

El primer paso antes de iniciar el *development* es que exista una necesidad para el programa, esto es clásico en las metodologías de desarrollo. Se adquieren los requisitos, se resuelven dudas, se hace un diseño para resolver la problemática y una vez que ya se entiende lo que se va a resolver y como se puede proceder.

Bueno, una vez que quedó todo el diseño y adquisición de requisitos completada ahora sigue la falta de programar … las pruebas. El chiste de TDD es programar primero las pruebas y luego su implementación así que se inicia con pruebas que por defecto fallarán porque no existe aún la implementación, este proceso es hacer las pruebas unitarias y demuestran funcionalidades individuales de las partes del programa, por defecto se debería tener un altisimo *code coverage* al final porque el código que se haga a futuro se hará para que las pruebas hechas sean consideradas como pasadas.

Así es, lo que sigue es escribir el código para que esas pruebas escritas pasen, el punto es que todas pasen para que tu implementación cumpla con los requisitos. Algo que se debe tomar en cuenta es que sí al acabar este paso hay pruebas que fallan, lo normal es que debas cambiar tu implementación para que pasen en vez de cambiar las pruebas. Esto solo podría perdonarse si de plano la prueba estaba mal programada. Pero el punto es que el desarrollo debe cumplir a las pruebas y no las pruebas al desarrollo.

Lo que sigue es hacer la limpieza del código. Ahora, posiblemente algunas bibliografías dirían que el código debería a ver de quedado lo mejor posible desde un inicio y que hacer “refactors” no deberían ocurrir luego de haber codificado …. Pero asumiendo que así no fue el caso ahora sigue limpiar el código sin romper implementación. Que se siga haciendo lo mismo, pero de forma más clara o optimizada, por lo que para verificar que esto se cumpla es bueno volver a correr las pruebas y verificar el *code coverage* y porcentaje de pruebas exitosas.

El siguiente paso es repetir el proceso, las pruebas ya pasaron, el código es bueno, y se cumple el requisito, por lo que ya se puede pasar a la siguiente tarea/subtarea/requisito.

10. Explica si el amor lo puede todo y por qué.

Una pregunta super filosófica profesor, la verdad. Ojalá esto no haya sido 100% en serio (o una búsqueda a alguna referencia que no he captado en estos tres meses) porque si es así entonces no sé qué es lo apropiado… así que diré algo para que no se vea vacía la hoja:

El amor no lo puede todo porque es intangible, se relaciona a las emociones, por si solo es incapaz de lograr algo. No puedes ir a la escuela con amor, puedes ir a la escuela y no te gusta, por defecto eso descontaría el **todo**, y podría seguir así por un rato.

Pero…. Lo que sí creo es **con amor se puede lograr todo**. A lo que me refiero con esto es a la actitud. Si se tiene buena actitud hacia algo con pasión y esfuerzo se puede lograr todo. El tenerle amor a tu trabajo, por ejemplo, significa que le vas a echar ganas y por lo tanto te irá mejor y podrás lograr tus metas cuales sean.

A lo que quiero llegar a todo esto es que el amor si lo puede todo si se usa como un catalizador para llegar a tus metas porque te servirá para pasar todos los obstáculos en tu camino y aprovechar todo lo que este a tu disposición para logarlo.

Hecho por: Omar Emilio Durán Cortez // 26780 (C8)

Materiales utilizados:

- https://docs.aws.amazon.com/sns/latest/dg/sns-ttl.html
    - TTL de SNS explicado
- https://aws.amazon.com/sqs/faqs/
    - FAQ de SNS explicado
- https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-configure-subscribe-queue-sns-topic.html
    - Explicación de suscripciones a tópicos en SNS
- https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-visibility-timeout.html
    - Explicación de Timeout de visibilidad
- https://developer.ibm.com/articles/5-steps-of-test-driven-development/
    - Pasos de TDD según IBM
- https://aws.amazon.com/s3/storage-classes/
    - Clases de almacenamiento en S3 por AWS
- https://aws.amazon.com/s3/storage-classes/glacier/
    - AWS S3 glacier explicado por AWS
- https://www.cloudflare.com/learning/dns/dns-records/dns-cname-record/
    - Explicación de Cloudflare de DNS y CNAMEs
- https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber
    - Explicación de Azure para Pub & Sub
- https://www.ibm.com/topics/message-brokers
    - Explicación de IBM de los message brokers
- https://www.freecodecamp.org/news/these-are-the-most-effective-microservice-testing-strategies-according-to-the-experts-6fb584f2edde/
    - Explicación de porque testing en microservicios se difculta