Examen Final

Describe los pros y cons de una arquitectura de software de microservicios.

Los microservicios sirven como un conjunto de pequeños servicios que se ejecutan de manera autónoma e independiente, por eso el software es más facil de que se acople a los cambios.

Ventajas:

- Escalabilidad: en necesidad de necesitar más o menos recursos, estos pueden cambiar conforme a estas necesidades.
- Modularidad/sencillés: Al no estar todo ligado en un mismo lugar es facil de implementar estos cambios, y en el caso de un error, estos no afectan a todo el software. Asimismo, el codigo puede ser reutilizado en sus diferentes componentes.
- Seguridad ante caidas: Al no estar en contenedores, si alguna falla, esto no hara que toda la aplicación muera y el resolver este problema no sera tan costoso.
- Curva de aprendizaje: es más fácil entrar y empezar a ser productivo como desarrollador, ya que se tiene que entender el funcionamiento de varios servicios pequeños en lugar de uno grande.

Desventajas:

- Diseño: Antes de iniciar un microservicio se deben de diseñar muy bien el como cada servicio dependerá de otros y que hara cada cosa.
- Memoria: Porque cada servicio esta separado se necesita mas consumo de memoria y CPU.
- Testing: debido a que los componentes estan distribuidos es dificil realizar las pruebas.
- Se requiere un control exhaustivo de las las versiones activas para no sobreescribirlas.

No siempre se necesita usar microservicios, se debe de evalaur muy bien el proyecto para ver si realmente sera una facilidad para el equipo en corto y largo plazo, o si esto traera problemas mayores.

Explica la arquitectura de Publish and Subscribe.

Pub/Sub

Esto ayuda a que cuando hay muchos microservicios asincronos, estos sigan sin tener que interdepender entre si, la documentación de AWS lo recomienda implementar en Amazon SNS para que los usuarios recivan una notificación en forma de mensaje cuando por ejemplo necesiten una comprobación de un pago realizado. Una aplicación publish/subscribe da una serie de servicios, a los usuarios, para gestionar, almacenar y presentar información, y otros usuarios se suscriben a ella para ser avisados de nuevas actualizaciones o informaciones.

Estos son usado mucho en el area de IoT y las redes sociales, ya que se espera que los que hacen **publish** envíen un mensaje cuando los que estan **subscritos** reciben estos mensajes al hacer cierta acción.

De esta forma cualquier mensaje publicado en un tema es recibido inmediatamente por todos los suscriptores del tema. La parte del Publish Subscribe permite que los mensajes se transmitan a diferentes partes de un sistema de forma asíncrona, asi el publisher no necesita saber quién está usando la información que está transmitiendo, y los subscribers no necesitan saber de quién proviene el mensaje.

Ventajas

- Menor acoplamiento.
- Más modularidad y más manejables.
- Permite la flexibilidad entre publicadores y y suscriptores.

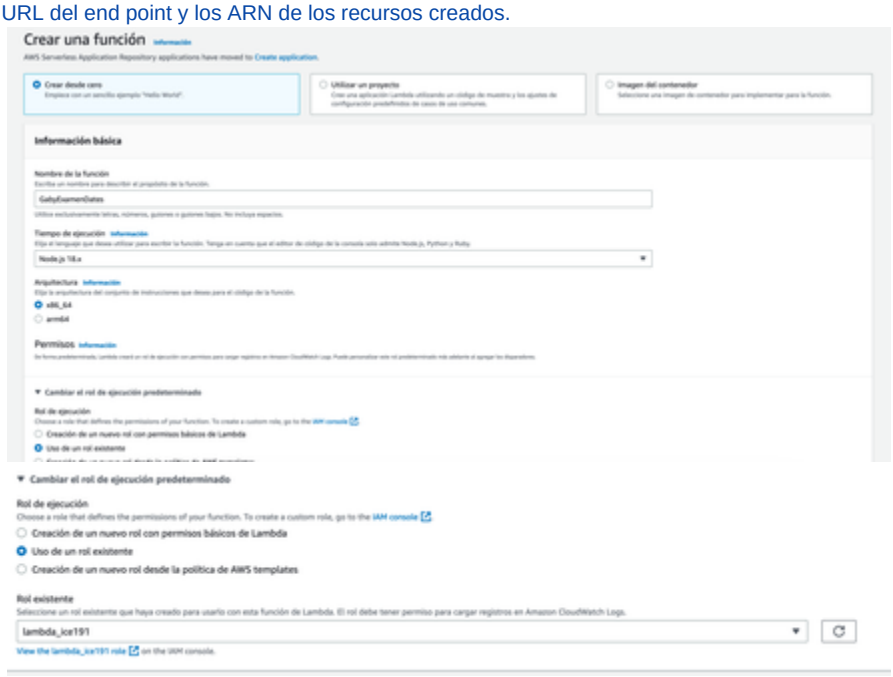
Desventajas

- Al no estar acoplado, no se sabe con seguridad el órden con el que se ejecutarán las cosas.
- Puede generar problemas de acumulación de eventos.

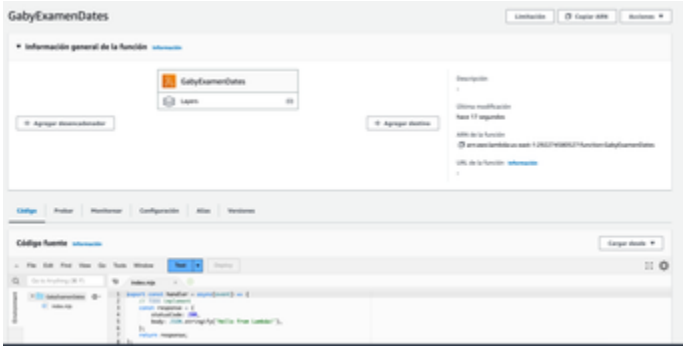
Crea una Lambda + API Gateway que reciba como parámetro un rango de fechas conformato YYYYMMDD y regrese la cantidad de días, horas y minutos entre dichas fechas.

Por ejemplo: http://elamorlopuedetodo.com/rango/20230303-20230404

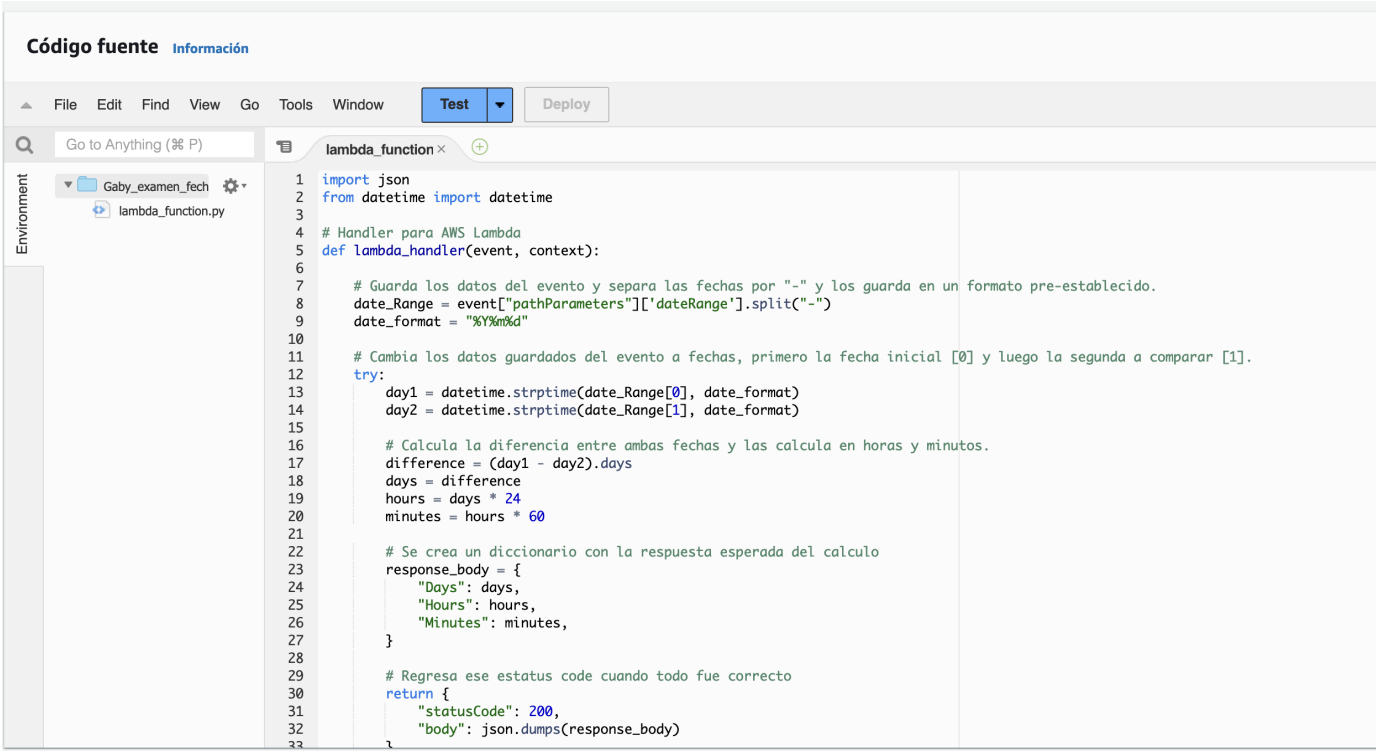
La respuesta debe ser el URL del end point y los ARN de los recursos creados.



Primero ingresamos a AWS con nuestros datos y damos click en **crear una función**, a partir de ahí llenamos los datos como en las imagenes anteriores y creamos la Lambda, a continuación deberá de verse una página como la siguiente:



Esta página muestra todo lo relacionado con nuestra Lambda, aqui mismo tenemos que ingresar el código para que regrese el request del API Gateway y le damos Deploy para guardar los cambios:



Este es el código:

```
import json
from datetime import datetime

# Handler para AWS Lambda
def lambda_handler(event, context):

    # Guarda los datos del evento y separa las fechas por "-" y los
guarda en un formato pre-establecido.
    date_Range = event["pathParameters"]['dateRange'].split("-")
    date_format = "%Y%m%d"

    # Cambia los datos guardados del evento a fechas, primero la fecha
inicial [0] y luego la segunda a comparar [1].
    try:
        day1 = datetime.strptime(date_Range[0], date_format)
        day2 = datetime.strptime(date_Range[1], date_format)

        # Calcula la diferencia entre ambas fechas y las calcula en
horas y minutos.
        difference = (day1 - day2).days
        days = difference
        hours = days * 24
        minutes = hours * 60

        # Se crea un diccionario con la respuesta esperada del calculo
        response_body = {
            "Days": days,
            "Hours": hours,
            "Minutes": minutes,
        }

        # Regresa ese estatus code cuando todo fue correcto
        return {
            "statusCode": 200,
            "body": json.dumps(response_body)
        }

    # Si hubo algun error regresa este StatusCode
    except ValueError as error:
        return {
            "statusCode": 422,
            "body": json.dumps({"message": "Wrong input formart."})
        }
```

Este código crea una función de Python llamada `lambda_handler` ; cuando se invoca, la función espera un objeto de evento y un objeto de contexto como entradas, la función espera que un parámetro llamado `dateRange` esté presente en el campo `pathParameters` del objeto de evento. Se espera que el parámetro `dateRange` sea una cadena que consta de dos fechas separadas por un guión ("-"), las cuales después se separan por el `.spli()`

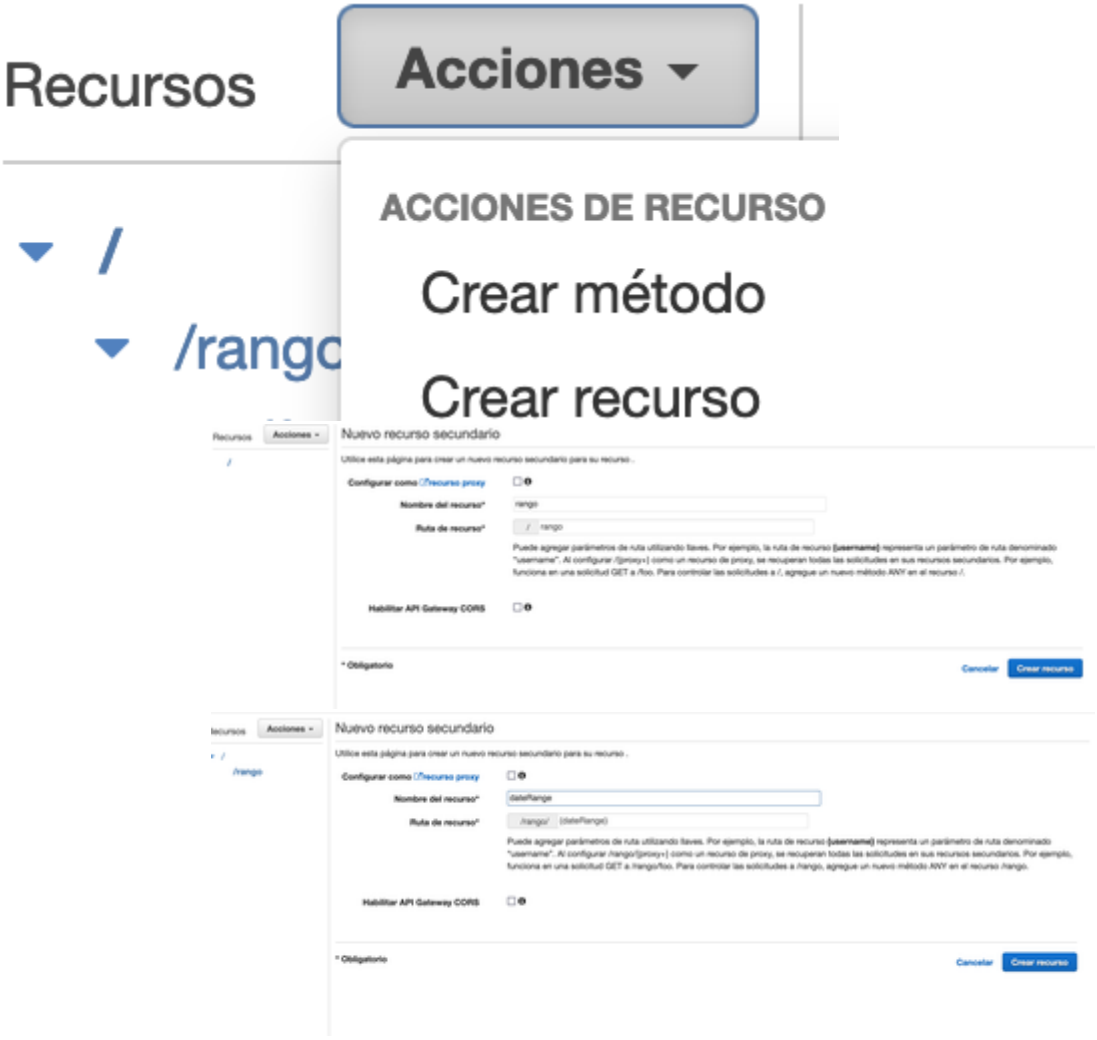
Luego, la función intenta analizar las dos fechas del parámetro `dateRange` utilizando el método `datetime.strptime()` del módulo datetime. Si tiene éxito, calcula la diferencia entre las dos fechas en días, horas y minutos y devuelve una respuesta en formato JSON que contiene esos valores como un diccionario.

Si se produce un error durante el análisis de las fechas, como un formato incorrecto, la función devuelve un mensaje de error con formato JSON con un código de estado 422.

Luego se hace la parte del **API Gateway**.

Le picamos en *crear API.*

Luego le ponemos en acciones y crear recurso, ahi se abrira una pestaña donde ponemos el nombre de recurso *rango* y bajo rango, se crea otro recuerdo con *{dateRange}* y bajo eso se llama un método *GET*.



Se hace deploy al GET en fase de DEV:

Después de eso se crea una prueba para verificar que la API este conectada mi Lambda como dev.

Este es el ejemplo de la prueba:

Se da la fecha **20230213-20220624** y se regresa la siguiente información:

```
{"Days": 234, "Hours": 5616, "Minutes": 336960}
```

ARN de la función:

```
arn:aws:lambda:us-east-1:292274580527:function:Gaby_examen_fechas
```

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

import json
from datetime import datetime

Handler para AWS Lambda

def lambda_handler(event, context):

```
# Guarda los datos del evento y separa las fechas por "-" y los guarda
en un formato pre-establecido.
date_Range = event["pathParameters"]['dateRange'].split("-")
date_format = "%Y%m%d"

# Cambia los datos guardados del evento a fechas, primero la fecha
inicial [0] y luego la segunda a comparar [1].
try:
    day1 = datetime.strptime(date_Range[0], date_format)
    day2 = datetime.strptime(date_Range[1], date_format)

    # Calcula la diferencia entre ambas fechas y las calcula en horas y
minutos.
    difference = (day1 - day2).days
    days = difference
    hours = days * 24
    minutes = hours * 60

    # HTML donde se muestra la diferencia de fechas generada.
    html = f"""<!DOCTYPE html>
    <html>
        <head>
            <h1>Days: {days}</h1>
            <h1>Hours: {hours}</h1>
            <h1>Minutes: {minutes}</h1>
        </head>
    </html>"""

    # Regresa un statusCode:200 con la respuesta en HTML pasada como
cuerpo y el tipo de contenido establecido en 'text/html'.
    return {
        "statusCode": 200,
        "body": html,
        "headers": {
            'Content-Type': 'text/html',
        }
    }

# Si sucede un error, lanza la siguiente excepcion:
except ValueError as error:
    html = f"""<!DOCTYPE html>
    <html>
        <head>
            <h1>Wrong input format.</h1>
```

```
            <h1>Input date should be YYYYMMDD.</h1>
        </head>
    </html>"""

    # Muestra un error 422: que el servidor entiende la peticion pero
hay un error semantico y no se puede procesar.
    return {
        "statusCode": 422,
        "body": html,
        "headers": {
            'Content-Type': 'text/html',
        }
    }
```

Este código hace una función AWS Lambda que toma un evento y un contexto como entrada y calcula la diferencia entre dos fechas en días, horas y minutos.

La función primero importa los módulos necesarios, `json` y `datetime`.

Luego se cuando se corre el handler este espera que tenga un diccionario que contenga la clave 'pathParameters', que a su vez es un diccionario con una clave 'date_Range' y se almacenan los datos en una lista llamada `dateRange`.

El método `datetime.strptime` se usa para convertir las dos cadenas de fecha en objetos `datetime` usando el `date_format` que en las lineas anteriores se especifico que el formato debe de ser: '%Y%m%d'.

Luego se hace un `try` donde se calcula la diferencia entre las dos fechas en días, horas y minutos. Luego genera una cadena HTML que contiene los resultados del cálculo, que se devuelve en el cuerpo de la respuesta.

Si ocurre una excepción durante el bloque de prueba (por ejemplo, si la fecha de entrada no está en el formato esperado), el código genera una cadena HTML de mensaje de error y la devuelve en el cuerpo de la respuesta con un código de estado de 422.

API Gateway

Se crea lo mismo que en el ejericio pasado, pero en otra API en mi caso se llama **API_gabyHTML**


```
<!DOCTYPE html>
        <html>
            <head>
                <h1>Days: 234</h1>
                <h1>Hours: 5616</h1>
                <h1>Minutes: 336960</h1>
            </head>
        </html>
```

ARN de la función:

`arn:aws:lambda:us-east-1:292274580527:function:GabyExamenHTML`

Describe la diferencia entre los S3 object storage class standard, infrequent access yGlacier.

S3 nos ayuda a guardar objetos al menor costo en un mismo lugar con seguridad y escalabilidad…

Standard: es la forma de almacenamiento clásica, estos datos pueden accederse de inmediato, se recomienda usarse cuando los datos son revisados o accedidos frecuentemente, ya que puede llegar a ser muy costoso, dentro de las características más importantes de esto es que tiene baja latencia y alto procesamiento.

Infrequent access: En este almacenamiento los datos se accesan con menos frecuencia, pero que cuando se haga una solicitud, esta sea rápida, este almacenamiento es más barato que el standar, pero si se pasa el número límite de solicitudes, se carga una tarifa extra.

Glacier: Este almacenamiento es más lento al momento de generar una solicitud, pueden llegar a tardar desde 1 hora hasta días despues de la solicitud; debido a esto es la opcion más económica. Estos se dividen en 3:

- *Instant retrieval:* El costo es muy bajo y tiene el mismo rendimiento y acceso en milisegundos que las clases de almacenamiento S3 Standard; es ideal para los datos de archivo que requieren acceso inmediato.
- *Flexible Retrieval:* Se usa para los datos de archivo que no requieren acceso inmediato, pero necesitan la flexibilidad de recuperar grandes conjuntos de datos sin costo alguno y es más barato que el Glacier anterior.
- *Deep Archive:* Es lo más económico, se usa por ejemplo “en negocios financieros, sanidad y sectores públicos, que retienen los conjuntos de datos durante un periodo de 7 a 10 años o más para cumplir los requisitos de conformidad normativa”

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

DNS es un servicio web utilizado para convertir direcciones web *cetystijuana.com* a direcciones ip ejemplo: 102.19.2.168, para poder iniciar primero se necesita tener un bucket de S3 donde se le hara host al static website con el mismo nombre del nuevo subdominio que deseemos; asimismo debemos habilitar el alojamiento de sitios web estáticos en el bucket de S3.

Al igual que en las primeras actividades en esta clase necesitamos un JSON para que lo redirija al nuevo subdominio del dominio padre:

```
{ "Comment":
  "CREATE record ",
  "Changes": [{"Action": "CREATE",
    "ResourceRecordSet": {
    "Name": "http://gueritagabs.cetystijuana.com ",
    "Type": "CNAME",
    "TTL": 300,
      "ResourceRecords": [{ "Value":
      "http://gueritagabs.cetystijuana.com.s3-website-us-east-1.
amazonaws.com "}]
    }}]
}
```

Así se podrá redirigir a mi subdominio, ya con el JSON, se corre el siguiente comando para crear el record del subdominio:

```
aws route53 change-resource-record-sets --hosted-zone-id hostedzone
/Z03346142C3RKH191036Y --change-batch file:{pathJSON}
```

Se usa AWS S3 para poder indicar el hosted zone y el file del JSON que se usará para la redirigir la dirección.

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

En este caso use lo que ya teniamos en clases pasada para leer los mensajes, solo agregue el borrado y el regresar algo en caso de que no se encontrara ningún mensaje a leer o borrar.

```
import boto3
import json

# Crea un cliente SQSsqs_client = boto3.client("sqs")
queue_url = sqs_client.get_queue_url(QueueName='gabyTest')['QueueUrl']

def read():
  response = sqs_client.receive_message(QueueUrl=queue_url)
    # Checa si existen mensajes en el queue
    if "Messages" in response:
    message = response["Messages"][0]
        # lee el mensaje
        readed_message = json.loads(message["Body"])["Message"]
        print(readed_message)
        return message
    else:
    # Si no hay mensajes regresa el sig mensaje
    print("No messages in queue")
        return None
def erase():
message = read()
    # Checa si hay algun mensaje
    if message:
    receipt_handle = message["ReceiptHandle"]
        # Borra el mensaje
        response = sqs_client.delete_message(
            QueueUrl=queue_url,
            ReceiptHandle=receipt_handle
        )
        # Muestra que el mensaje fue borrado
        print("Message Deleted: {0}".format(response))
    else:
    # Si no hay mensajes muestra lo sig
    print("No message to delete")
```

La función **read()** comprueba si hay algún mensaje en la cola antes de intentar leer el primer mensaje. Si no hay mensajes, imprime un mensaje en la consola y devuelve None. De lo contrario, recupera el primer mensaje y extrae el cuerpo del mensaje.

La función **erase()** llama a la función read() para recuperar el primer mensaje de la cola. Si hay un mensaje, extrae el valor ReceiptHandle y lo usa para eliminar el mensaje. Si no hay mensaje, imprime un mensaje a la consola.

Explica las diferencias entre SNS y SQS.

SNS (Simple Notification Service) : Envia mensajes (push-based delivery) de forma asíncrona a los subscriptores de una campaña por email, Lambda, HTTPS… (from publishers to subscribers) no al reves; esto permite enviar mensajes individuales o mensajes masivos a un gran número de destinatarios a los subscribers.

Solo recibe archivos tipo JSON y XML y los precios varian dependiendo del número de mensajes que se publiquen a los subscriptores.

SQS (Simple Queue Service) : Envia y recibe (pull-based delivery) mensajes en cola, todos los mensajes se ingresan en la cola, hasta que son procesados por un consumidor y salen de la cola, en otras palabras si los subscriptores quieren ver un mensaje lo tiene que jalar de la cola.

Recibe cualquier formato de archivo, aquí los precios varían dependiendo del número de peticiones que recibe el servicio, cuantos de estos mensajes están guardados y cuantos son enviados.

Explica el ciclo de TDD.

Test Driven Development

En este caso los tests se desarrollan primero los tests y después se crea el codigo, enfoque iterativo para el diseño de software, cuando escribimos software usando este enfoque, el producto final es desconocido; pero proporciona la capacidad de obtener comentarios rápidos sobre el diseño del software.

1ero. Necesitas crear unit test super especificos.

2do. Se crea el codigo y cuando algún test falle, se tiene que mejorar hasta que el código sea exitoso.

3ero. Se refactoriza el código para evitar redundancias y mejorar el rendimiento de la aplicación.

Esto tiene grandes beneficios como el poder entender correctamente que es lo que se necesita en ese momento y que es lo que cliente espera realmente, no hacer código extra, por ende también la productividad del equipo es mejor.

Explica si el amor lo puede todo y por qué.

Hora de sacar mi lado obscuro (el cursi 😜)

Yo soy fiel creyente de recordar las cosas buenas y si nos efocamos en lo malo de las personas/situaciones seriamos muy amargados. Y el que el amor lo puede todo yo creo que si, lo vemos en el amor de las madres a sus hijos, de una persona a sus mascotas, novios …

Uno cuando ama busca la felicidad del otro, sin importar que, esto a veces nos puede perjudicar como individuos, porque dejamos de pensar en nuestra propia felicidad con tal de darsela a alguien más, pero si uno tiene buenas bases de cuanto vale como persona esto no debería de pasar.

Ahora bien, por más que uno ame a una persona, a veces no es lo mejor para nosotros (no le voy a contar la historia con mi ex, bueno si quiere chismecito pues si jajaja), pero hay cosas que van más allá de nuestras manos, yo creo que la gente si puede cambiar a otras personas, pero solo si estas lo desean. Y hay veces que uno no hace click con la forma de vida de otra persona, por más que uno lo desee y por más amor que tengas hacia alguien más, va a llegar un momento en que las cosas negativas empiezan a sobrepasarte, pero si sales a tiempo y lo manejas con “amistad” las cosas pueden seguir bien y no quedar peleados.

Ahora el amor influye en todo, no solo en la pareja, por ejemplo si uno estudia algo que no le gusta va a trabajar en algo que no le gusta y ser infeliz, pero si estudias algo que amas, vas a querer mejorar y siempre seguir aprendiendo.

Asimismo las muestras de amor de las personas pueden variar, puede ser un abrazo, un mensaje, una canción dedicada … Pero hay cosas que no vemos, conversaciones que no oimos y debemos de confiar en que lo que sucede en nuestra vida es por algo y debemos de ver el lado bueno de todo.

Tambien el amor puede generar dolor, pero quiero pensar que todo ese dolor es pasajera y es mejor recordar lo bueno, uno puede inspirar a los demás demostrando ese amor y así poco a poco cambiar la forma de pensar de otros. Y si uno realmente ama a alguien o a una cosa, buscará la forma de lograr estar sincronía.

La verdad se me ocurre muchas cosas que decir de este tema pero siento que ni el tiempo ni mi mente puede desarrollar todas ahorita, pero si creo que el amor lo puede todo, solo es cuestión de ver que tanto estás dispuesto a dar a cada persona/situación con tal de que no te afecte de forma negativa. ❤️