Integrantes: Roberth Lara y Joel Cuascota

Proyecto de Computación en la Nube

Aplicación Serverless utilizando AWS

1. Introducción

En el presente reporte se detalla el desarrollo y despliegue de una aplicación serverless utilizando los servicios de Amazon Web Services (AWS). La arquitectura serverless ofrece una solución escalable, eficiente y rentable para aplicaciones modernas, eliminando la necesidad de gestionar infraestructura tradicional. Este enfoque permite concentrar los esfuerzos en el diseño de la lógica de negocio y la experiencia del usuario, dejando en manos de AWS la gestión de recursos, escalabilidad y alta disponibilidad.

El proyecto se implementó utilizando los siguientes servicios principales de AWS:

- **Amazon S3**: Para almacenar y servir archivos estáticos como HTML, CSS y JavaScript, proporcionando una interfaz web dinámica y fácil de escalar.
- **AWS Lambda**: Para ejecutar funciones de backend en respuesta a eventos, eliminando la necesidad de servidores dedicados y permitiendo un escalado automático bajo demanda.
- **Amazon API Gateway**: Para gestionar las interacciones entre los usuarios y las funciones Lambda mediante una API REST segura y de fácil configuración.
- **Amazon DynamoDB**: Para almacenar y consultar datos de manera eficiente, garantizando un rendimiento de baja latencia independientemente de la carga de trabajo.

La aplicación que se despliega en este proyecto es una plataforma de reservas de hotel, diseñada para facilitar la búsqueda, selección y reserva de habitaciones de manera rápida y eficiente. El propósito principal de esta aplicación es ofrecer una experiencia de usuario intuitiva y fluida para los clientes que desean encontrar y reservar hospedaje en diversas ubicaciones.

El objetivo de este proyecto es proporcionar una solución accesible y moderna para los viajeros, mientras que también permite a los administradores de los hoteles gestionar fácilmente las reservas y mantener actualizada la disponibilidad de las habitaciones en tiempo real.

2. Diseño de la arquitectura de la aplicación

El diseño arquitectónico de la aplicación de reservas de hotel sigue el enfoque serverless, aprovechando los servicios escalables y gestionados de AWS. Esta arquitectura permite que la aplicación se ejecute de manera eficiente, flexible y rentable, sin necesidad de gestionar infraestructura física o virtual, lo que mejora la disponibilidad, seguridad y rendimiento de la plataforma.

A continuación, se detallan los componentes principales y los servicios de AWS utilizados en la arquitectura:

Amazon S3 (Simple Storage Service)

Amazon S3 se utiliza para almacenar y servir los archivos estáticos de la aplicación, como los archivos HTML, CSS, JavaScript y las imágenes. Al estar completamente gestionado, S3 asegura que los archivos estén siempre disponibles y sean fácilmente accesibles por los usuarios a través de la web, además de permitir un alto nivel de escalabilidad para manejar picos de tráfico sin comprometer el rendimiento.

AWS Lambda

AWS Lambda es el servicio que permite ejecutar la lógica de backend de la aplicación sin necesidad de gestionar servidores. Las funciones Lambda son invocadas por eventos generados por las solicitudes de los usuarios a través de la API Gateway. Estas funciones realizan operaciones clave, como procesar las reservas de hotel, gestionar la autenticación de los usuarios y manejar la lógica de negocio relacionada con las fechas de reserva y la disponibilidad de habitaciones. Lambda garantiza un escalado automático y un tiempo de respuesta rápido en función de la demanda.

Amazon API Gateway

Amazon API Gateway actúa como un intermediario entre el cliente (el navegador

web) y las funciones Lambda. Es responsable de exponer la API RESTful que los usuarios utilizan para interactuar con la aplicación. API Gateway gestiona las solicitudes HTTP, las valida y las redirige a las funciones Lambda correspondientes.

Amazon DynamoDB

Amazon DynamoDB es una base de datos NoSQL completamente gestionada que almacena los datos de la aplicación. DynamoDB está optimizado para manejar grandes volúmenes de datos con baja latencia, lo que permite consultas rápidas y eficientes, independientemente del tamaño de la base de datos o la carga de trabajo. Además, su escalabilidad automática asegura que la aplicación pueda crecer sin preocuparse por el rendimiento de la base de datos.

3. Implementación detallada

La implementación de la aplicación de reservas de hotel se llevó a cabo utilizando exclusivamente los servicios de AWS mencionados: **Amazon S3**, **AWS Lambda**, **Amazon API Gateway** y **Amazon DynamoDB**. Creamos una cuenta en AWS para tener una experiencia más cercana al entorno de AWS y desplegar la aplicación. La solución se configuró completamente a través de la consola web de AWS.

Amazon S3: Almacenamiento de archivos estáticos

- **Creación de un bucket en S3**:
  Primero, se creó un bucket en Amazon S3 para almacenar los archivos estáticos de la aplicación (HTML, CSS, JavaScript e imágenes).
  - Se accedió a la consola de S3 desde el panel de AWS.
  - Se creó un nuevo bucket con un nombre único.
  - Se configuraron las políticas de acceso para permitir la lectura pública de los archivos estáticos, lo que permite que los usuarios accedan a los recursos sin autenticación.
- **Subida de archivos estáticos**:
  Los archivos de la aplicación fueron subidos al bucket, incluyendo los archivos HTML para la interfaz, los archivos CSS para el diseño y JavaScript para la funcionalidad dinámica de la página web.
- **Configuración del hosting web estático**:
  Se habilitó la opción de "Hosting web estático" en el bucket de S3, lo que permite que S3 sirva los archivos estáticos cuando se accede a la URL proporcionada por AWS. Se configuraron las opciones para especificar los archivos de índice (index.html).

AWS Lambda: Ejecución de la lógica de backend

- **Creación de funciones Lambda**:
  Se crearon varias funciones Lambda para gestionar las operaciones principales de la aplicación.
    - Se accedió a la consola de Lambda y se creó una función para cada tarea clave. Por ejemplo:
        - getAvailability: Esta función consulta todas las reservaciones en la base de datos DynamoDB.
        - createReservation: Esta función maneja la creación de nuevas reservas y actualiza en DynamoDB.
        - deleteReservation: Esta función elimina una reserva y actualiza la tabla de DynamoDB
- **Configuración de permisos**:
  Cada función Lambda se configuró con los permisos necesarios para interactuar con DynamoDB y otros recursos de AWS. Esto se logró a través de roles de IAM (Identity and Access Management), asignando permisos adecuados para permitir la lectura y escritura en DynamoDB. Para eso fuimos a IAM y creamos un rol que solo de acceso a DynamoDB.
- **Desarrollo del código**:
  El código de las funciones Lambda se implementa directamente en la consola de Lambda utilizando el editor en línea. Por ejemplo, el código para la función getAvailability puede verse de la siguiente manera:

```python
import boto3
import json

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('HotelReservations')

def lambda_handler(event, context):
    date = event['queryStringParameters']['date']
    response = table.scan(
        FilterExpression="date = :date",
        ExpressionAttributeValues={":date": date}
    )
    reservations = response['Items']
    return {
        'statusCode': 200,
        'body': json.dumps(reservations),
        'headers': {'Content-Type': 'application/json'}
    }
```

Amazon API Gateway: Exposición de la API

- **Creación de una nueva API REST**:
  Se utilizó Amazon API Gateway para crear una API RESTful que sirviera como interfaz entre los usuarios y las funciones Lambda.
    - En la consola de API Gateway, se creó una nueva API REST y se configuraron los recursos necesarios, como /availability para la consulta de reservas y /reservation para realizar una reserva o eliminar una.
    - Cada recurso se configuró con un método HTTP (GET, POST o DELETE), y se vinculó a la función Lambda correspondiente.
- **Integración con Lambda**:
  En la configuración de los métodos de API Gateway, se integraron las funciones Lambda como las fuentes de backend para cada recurso. Se configuró el tipo de integración como "Lambda Function" y se seleccionaron las funciones Lambda previamente creadas.
- **Configuración de CORS**:
  Para permitir que la aplicación web accediera a la API desde diferentes dominios, se habilitó el soporte de CORS (Cross-Origin Resource Sharing) en los recursos de la API.
- **Despliegue de la API**:
  Una vez configurada la API, se creó un entorno de despliegue (por ejemplo, prod) y se desplegó la API. AWS generó una URL pública que los usuarios pueden utilizar para interactuar con la API.

Amazon DynamoDB: Almacenamiento de datos

- **Creación de la tabla DynamoDB**:
  Se creó una tabla en DynamoDB para almacenar los datos y las reservas. La tabla HoteleReservations tiene claves de partición como reservation_id y atributos adicionales como usuario, fecha_reserva y estado.
    - En la consola de DynamoDB, se configuraron las claves primarias y los índices secundarios necesarios para realizar consultas rápidas sobre las habitaciones disponibles y las reservas.
- **Población de la tabla**:
  Se cargaron datos de ejemplo a través de la consola de DynamoDB o mediante scripts Lambda.
- **Interacciones con DynamoDB**:
  Las funciones Lambda utilizan el SDK de AWS para interactuar con DynamoDB, realizando operaciones como get_item, put_item y query. Por ejemplo, al realizar una reserva, la función Lambda actualiza la disponibilidad de las habitaciones en DynamoDB.

4. **Simulation Approach and Setup Instructions**

Este proyecto fue desarrollado y desplegado utilizando exclusivamente la **interfaz gráfica de AWS (AWS Management Console)**. Cada componente se configuró y gestionó directamente desde la plataforma web de AWS, sin recurrir a herramientas locales o simulaciones externas.

Pasos de configuración detallados:

1. **Iniciar sesión en la consola de AWS.**
    - Acceder al portal oficial de AWS y crear una cuenta si no se tenía una previamente.
2. **Crear los recursos necesarios en la consola gráfica:**
    - **Amazon S3:** Crear un bucket único para almacenar los archivos estáticos (HTML, CSS, JavaScript) de la aplicación. Configurar las políticas de acceso público y habilitar el hosting estático.
    - **AWS Lambda:** Crear tres funciones principales (getAvailability, createReservation, deleteReservation) e implementar el código utilizando el editor en línea de Lambda.
    - **Amazon DynamoDB:** Crear una tabla HotelReservations con una clave primaria (reservation_id) y atributos adicionales como fechas y detalles de reservas.
    - **Amazon API Gateway:** Configurar las rutas de la API RESTful (GET /availability, POST /reservation, DELETE /reservation) y vincularlas a las funciones Lambda respectivas.
3. **Configurar permisos y políticas en IAM:**
    - Crear roles específicos para cada servicio, asegurando que solo tengan los permisos mínimos necesarios. Por ejemplo, las funciones Lambda se configuraron con acceso exclusivo a DynamoDB.
4. **Subir los archivos al bucket de S3:**
    - Cargar el frontend de la aplicación al bucket de S3, incluyendo archivos de índice (index.html) y error (error.html), y validar que el hosting estático funcionara correctamente.
5. **Probar la funcionalidad:**
    - Validar directamente desde la consola de AWS las interacciones entre servicios. Además, se usaron herramientas como Postman para probar las rutas de API Gateway.

5. Testing Methodology and Results

Metodología de pruebas:

1. **Pruebas Unitarias:**

    - Se realizaron desde la consola de AWS utilizando eventos de prueba preconfigurados para validar cada función Lambda de manera individual.
2. **Pruebas de Integración:**
    - API Gateway se probó utilizando Postman para verificar que las rutas conectaran correctamente con las funciones Lambda y manejaran las solicitudes según lo esperado.
3. **Pruebas End-to-End:**
    - Se realizó un flujo completo desde el frontend (alojado en S3), enviando solicitudes a través de API Gateway hacia Lambda, y verificando la interacción con DynamoDB.

Resultados:

- **`GET /availability`:** Retornó datos precisos y actualizados sobre la disponibilidad de habitaciones.
- **`POST /reservation`:** Creó reservas de manera exitosa, validando la ausencia de conflictos en las fechas.
- **`DELETE /reservation`:** Eliminó reservas de manera precisa según el ID proporcionado.

Todas las pruebas confirmaron que los componentes se integraron correctamente y que el sistema es funcional.

6. Challenges Faced and Lessons Learned

Trabajar en un proyecto serverless utilizando API Gateway, S3, Lambda y DynamoDB fue una experiencia enriquecedora tanto a nivel técnico como en el desarrollo de habilidades de resolución de problemas. Aquí están las lecciones y desafíos más significativos que encontré durante el proyecto:

Desafíos:

1. **Configuración de CORS y Permisos en IAM:**
    - Configurar correctamente CORS en API Gateway fue un desafío inicial. Sin esta configuración, las solicitudes desde el frontend no se procesaban correctamente. Asimismo, diseñar permisos específicos y seguros en IAM para garantizar el acceso adecuado de cada servicio fue un proceso minucioso y esencial para la seguridad de la aplicación.
2. **Optimización de Respuestas en Lambda:**
    - Al principio, las funciones Lambda presentaban demoras en las respuestas debido a configuraciones iniciales que no estaban

optimizadas. Este problema me obligó a ajustar el código y los parámetros de ejecución para mejorar el rendimiento.

3. **Control de Costos en un Entorno Escalable:**
    - Aunque servicios como Lambda y DynamoDB escalan automáticamente según la demanda, aprendí que esta flexibilidad también puede generar costos inesperados si no se monitorean adecuadamente las métricas de uso y consumo. Este desafío me enseñó la importancia de configurar límites y alertas para evitar sobrecostos.

Lecciones Aprendidas:

1. **Simplificar el Desarrollo con Serverless:**
    - La arquitectura serverless elimina la necesidad de administrar servidores, lo que me permitió centrarme completamente en el diseño y la lógica del negocio. Pude construir una solución rápida y escalable sin preocuparme por el aprovisionamiento o mantenimiento de infraestructura.
2. **Aprovechar el Diseño Modular:**
    - Dividir el proyecto en componentes independientes, como el frontend en S3, la lógica en Lambda y los datos en DynamoDB, mejoró significativamente la organización del proyecto. Esta modularidad facilita la actualización o el reemplazo de partes específicas sin afectar al sistema completo.
3. **La Importancia de la Configuración Correcta:**
    - Aprendí que la configuración correcta de servicios como CORS y los permisos de IAM es fundamental para garantizar la funcionalidad y seguridad del sistema. Este proceso me enseñó a ser minucioso y a realizar pruebas continuas para validar los accesos y las configuraciones.
4. **Manejar la Escalabilidad Automática:**
    - Ver cómo Lambda y DynamoDB se adaptaban automáticamente a las cargas de trabajo fue una experiencia reveladora. Sin embargo, esta escalabilidad automática también destacó la necesidad de monitorear y optimizar el uso para evitar costos elevados.

7. Potential Improvements and Future Enhancements

Mejoras futuras:

1. **Autenticación de usuarios:** Integrar **AWS Cognito** para ofrecer un sistema de inicio de sesión seguro y personalizado para los usuarios.

2. **Notificaciones:** Implementar **Amazon SES** para enviar correos electrónicos automáticos que confirmen reservas o notifiquen cancelaciones.
3. **Optimización de consultas:** Usar índices secundarios globales (GSIs) en **DynamoDB** para mejorar la eficiencia en búsquedas y filtrados complejos.
4. **Frontend dinámico:** Migrar el frontend a un framework como **React** o **Angular** para ofrecer una experiencia de usuario más rica e interactiva.
5. **Análisis y métricas:** Incorporar **AWS CloudWatch** para monitorear métricas clave y analizar el rendimiento de la aplicación.