Diego González, 13 de septiembre de 2024.

Solución de Monitoreo para cliente “Cargo Express”

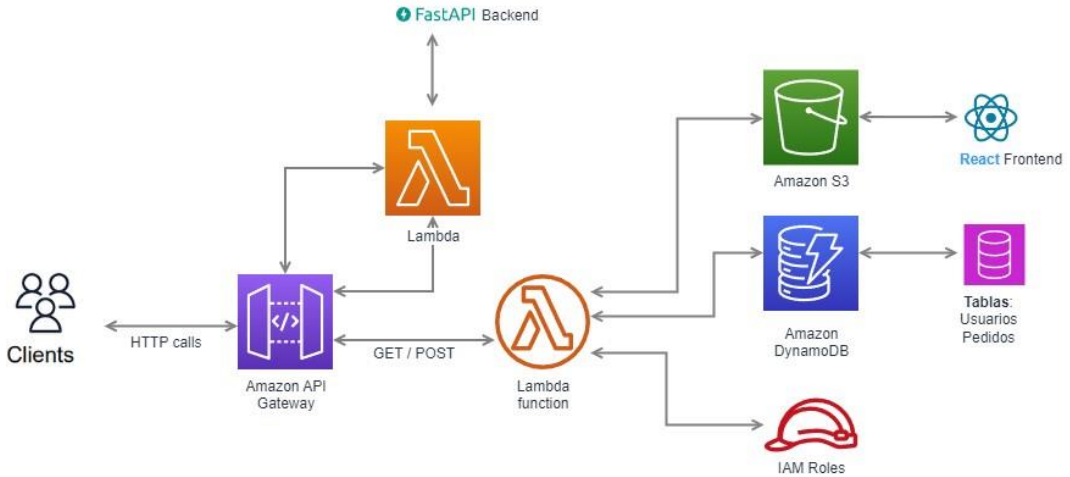
Bajo el contexto del desarrollo y despliegue de soluciones tecnológicas para atender las necesidades de negocio de nuestros clientes, Mazio ha trabajado en conjunto con la empresa de mensajería “Cargo Express”, quienes operan en toda Colombia distribuyendo distintos tipos de productos. En esta ocasión, el objetivo ha sido la implementación de una aplicación de monitoreo para realizar el seguimiento en un tiempo cercano al real de los principales indicadores y métricas, utilizando arquitecturas serverless y la nube de AWS.

El presente documento recopila todos los pasos seguidos para el desarrollo de dicha solución, diagramas de arquitectura, explicación de los códigos, etc, para así poder servir también de manual en caso de que se requiera la replicación de uno o varios componentes de este despliegue en otro ambiente.

1. Descripción de la solución

Cargo Express requiere un sistema eficiente y de rápida respuesta que permita registrar las entregas de productos en tiempo real, con una tolerancia máxima de 10 segundos para reflejar los registros en sus sistemas. Además, se busca una plataforma para monitorear las entregas y métricas clave de manera fácil y accesible para tomadores de decisiones que no son expertos en tecnología.

Con esto en mente, se utilizaron los servicios de la nube de AWS para construir la arquitectura de la solución, los módulos y elementos empleados se resumen en el siguiente diagrama:



El diagrama presentado describe la arquitectura la solución en la nube para un sistema backend basado en AWS Lambda y FastAPI, con un frontend en React. A continuación, se describen cada uno de los componentes y cómo interactúan entre sí:

1. **Clientes**: Los usuarios o clientes hacen solicitudes HTTP (GET/POST) a través de sus dispositivos. Estas solicitudes pueden ser para registrar un pedido, consultar datos o realizar otras operaciones con la aplicación.
2. **Amazon API Gateway**: Es el punto de entrada para las solicitudes de los clientes. Este servicio actúa como un proxy para procesar las solicitudes HTTP, que luego son dirigidas a la función Lambda que ejecuta el backend.
3. **Lambda Function**: Las solicitudes gestionadas por el API Gateway son enviadas a AWS Lambda, que ejecuta el backend desarrollado con FastAPI. Lambda es un servicio serverless que permite ejecutar el código sin necesidad de gestionar servidores. Aquí se maneja la lógica de negocio de la aplicación, como validaciones, autenticación y procesamiento de pedidos.
4. **FastAPI Backend**: La función Lambda ejecuta el backend desarrollado con FastAPI, que es un framework ligero y rápido para desarrollar APIs. Este backend maneja las diferentes operaciones de la aplicación, que constan de realizar el proceso de login mediante credenciales de usuario, registrar pedidos en las bases de datos, visualizar dichos pedidos y obtener métricas de negocio actualizadas.
5. **Amazon DynamoDB**: DynamoDB es la base de datos NoSQL utilizada para almacenar las tablas de usuarios y pedidos. El backend se comunica con DynamoDB para guardar y recuperar los datos necesarios, como información de los usuarios, detalles de los pedidos y estadísticas.
6. **Amazon S3**: Los archivos estáticos, imágenes, u otros tipos de recursos, como los relacionados con el frontend en React, están almacenados en un bucket de Amazon S3. S3 actúa como un repositorio de almacenamiento escalable para la aplicación.
7. **React Frontend**: El frontend de la aplicación está desarrollado con React y alojado en S3 para que los usuarios interactúen a través de la interfaz gráfica. Las solicitudes de los usuarios en la interfaz son gestionadas por el API Gateway.
8. **IAM Roles**: Para garantizar la seguridad, IAM Roles (AWS Identity and Access Management) se utilizan para controlar los permisos de acceso a los diferentes servicios (como Lambda, DynamoDB y S3). Estos roles aseguran que solo las partes autorizadas tengan acceso a los recursos adecuados.

Habiendo explicado los elementos que conforman esta solución, se procede a presentar en detalle los pasos seguidos para la construcción de los códigos, configuraciones en la nube, etc.

2. Desarrollo del Backend.

A continuación, se detallan todos los pasos realizados para la implementación del backend de la aplicación de monitoreo en tiempo real de entregas. Este backend fue desarrollado utilizando FastAPI y desplegado en AWS Lambda, integrándose con otros servicios como API Gateway y DynamoDB. A través de estos servicios, se garantiza una arquitectura serverless que permite gestionar pedidos, autenticación de usuarios y la obtención de métricas sin necesidad de gestionar servidores físicos. Cada componente fue configurado para asegurar alta disponibilidad, escalabilidad automática y seguridad en el acceso mediante tokens JWT.

2.1 Creación del entorno virtual: Iniciamos creando un entorno virtual de Python para aislar las dependencias necesarias para el proyecto utilizando el siguiente comando.

```
python -m venv env
```

Se activa el entorno virtual:

Windows: `.\env\Scripts\activate`
Linux/Mac: `source env/bin/actívate`

2.2 Instalación de Dependencias: Instalamos las dependencias necesarias para el proyecto, principalmente FastAPI, Mangum (para la integración con Lambda), Boto3 (para interactuar con DynamoDB) y otras librerías requeridas presentes en el archivo de texto requirements. En este caso, las versiones y librerías utilizadas fueron:

```
annotated-types-0.7.0
anyio-4.4.0
boto3-1.35.17
botocore-1.35.17
certifi-2024.8.30
cffi-1.17.1
charset-normalizer-3.3.2
click-8.1.7
colorama-0.4.6
cryptography-43.0.1
ecdsa-0.19.0
exceptiongroup-1.2.2
fastapi-0.99.1
h11-0.14.0
idna-3.8
```

```
jmespath-1.0.1

mangum-0.17.0

pyasn1-0.6.1

pycparser-2.22

pydantic-1.10.18

pydantic_core-2.23.3

python-dateutil-2.9.0.

post0 python-jose-3.3.0

python-multipart-0.0.9

requests-2.32.3

rsa-4.9

s3transfer-0.10.2

six-1.16.0

sniffio-1.3.1

starlette-0.27.0

typing_extensions-4.12.2

urllib3-2.2.2

uvicorn-0.30.6
```

Es importante destacar que, para versiones superiores de FastAPI, existen problemas de compatibilidad al momento de hacer el despliegue en AWS Lambda, ya que entra en conflicto con el módulo de pydantic. Dicho esto, se recomienda trabajar con la versión aquí especificada que demostró ser más estable.

Se instalan las dependencias con el comando:

```
pip install -r requirements.txt
```

2.3 Desarrollo del Backend con FastAPI: El backend consta de un código de Python 3.10 (main.py), que utiliza FastAPI como framework, con soporte para AWS Lambda a través de Magnum y DynamoDB como base de datos para almacenar pedidos y usuarios. A continuación, se explica cada parte clave del código:

2.3.1 Importaciones:

```
from decimal import Decimal
import boto3
from fastapi import FastAPI, HTTPException, Depends
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from fastapi.middleware.cors import CORSMiddleware
from jose import JWTError, jwt
from pydantic import BaseModel
from datetime import datetime, timedelta
from mangum import Mangum
from typing import List
from datetime import date
```

El código importa varias librerías esenciales:

- boto3: Biblioteca para interactuar con los servicios de AWS (en este caso DynamoDB)
- fastapi: Framework para construir la API backend.
- jose: Utilizado para crear y verificar tokens JWT (Json Web Tokens).
- pydantic: Para la validación de modelos de datos.
- Mangum: Adaptador para ejecutar FastAPI en AWS Lambda.

2.3.2 Inicialización de FastAPI y Middleware:

```
app = FastAPI()
security = HTTPBearer()
```

Se crea una instancia de la aplicación FastAPI, con soporte para autenticación basada en tokens (HTTPBearer). Se configura CORS para permitir solicitudes desde orígenes específicos como la aplicación web.

```
# Agregar el middleware de CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

2.3.3 Conexión a DynamoDB:

Aquí se configuran las tablas de DynamoDB. Pedidos almacena los pedidos realizados, y Usuarios almacena las credenciales de los usuarios que tienen acceso.

```python
# Configura DynamoDB
dynamodb = boto3.resource('dynamodb')
table_pedidos = dynamodb.Table('Pedidos')
table_users = dynamodb.Table('Usuarios')
```

2.3.4 Seguridad y generación de Tokens JWT: Se definen la clave secreta (a partir de una variable de entorno) y el algoritmo de cifrado utilizados para generar tokens JWT. Estos tokens son válidos por 30 minutos.

```python
# Clave secreta y configuración de token
SECRET_KEY = os.getenv("SECRET_KEY", "default_secret_key")
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30
```

2.3.5 Funciones de autenticación y verificación:

get_user_from_dynamo: Esta función busca y recupera la información del usuario desde DynamoDB en base a su nombre de usuario.

```python
def get_user_from_dynamo(username: str):
    try:
        response = table_users.get_item(Key={'username': username})
        return response.get('Item')
    except Exception as e:
        return None
```

Utiliza el método get_item de boto3 para buscar un usuario por su nombre de usuario (username) en la tabla Usuarios. Si encuentra al usuario, devuelve la información almacenada de este (un diccionario con atributos del usuario, como la contraseña).

verify_password: Esta función compara la contraseña almacenada en la base de datos (DynamoDB) con la contraseña proporcionada por el usuario al intentar iniciar sesión.

```python
# Funciones de autenticación y verificación de token
def verify_password(stored_password: str, provided_password: str):
    return stored_password == provided_password
```

Nota: *En un entorno de producción, generalmente se usa una función de "hash" en lugar de una comparación directa para mayor seguridad. Para el desarrollo de esta prueba técnica, por motivos de tiempo no se alcanzó a implementar el hash por incompatibilidades con la librería bcrypt al momento de desplegar en Lambda.*

create_access_token: Esta función genera un token de acceso (JWT) para el usuario autenticado. Los tokens se usan para mantener la sesión del usuario sin necesidad de enviar las credenciales en cada solicitud.

```python
def create_access_token(data: dict, expires_delta: timedelta | None = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt
```

Parámetros:

data: Un diccionario que contiene la información que se incluirá en el token (por ejemplo, el nombre de usuario o cualquier otra información relevante).
expires_delta: Es el tiempo de expiración opcional del token. Si no se proporciona, el token expira en 15 minutos.

Funcionamiento:

Copia la información recibida en data y añade una clave adicional exp para definir la fecha y hora de expiración del token. Usa la biblioteca jwt (JSON Web Token) para codificar el diccionario to_encode en un token firmado con la clave secreta (SECRET_KEY) y el algoritmo de codificación (HS256). El token generado por esta función se devuelve al cliente como parte del proceso de autenticación. El cliente usa este token para hacer solicitudes autenticadas a la API.

Verify_token: Esta función verifica y decodifica el token JWT enviado por el cliente en cada solicitud. Si el token es válido y no ha expirado, se permite la operación; de lo contrario, se rechaza.

```python
# Función para verificar el token JWT
def verify_token(token: str):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload
    except JWTError:
        raise HTTPException(status_code=403, detail="Token inválido o expirado")
```

Utiliza jwt.decode para decodificar el token con la clave secreta (SECRET_KEY) y el algoritmo de codificación (HS256). Si el token es válido, devuelve el contenido del token (payload), que normalmente contiene la identidad del usuario y la información de expiración. Si el token no es válido o ha expirado, lanza una excepción HTTPException con el estado 403 (prohibido) y el mensaje "Token inválido o expirado".

2.3.6 Endpoints de la API:

Autenticación: /token
Este endpoint recibe un nombre de usuario y contraseña, verifica la autenticación, y si es válida, genera un token JWT que se devuelve al usuario.

```python
# Endpoint para generar token
@app.post("/token", response_model=Token)
async def login_for_access_token(user: User):
    db_user = get_user_from_dynamo(user.username)
    if not db_user or not verify_password(db_user["password"], user.password):
        raise HTTPException(status_code=401, detail="Incorrect username or password")

    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(data={"sub": user.username}, expires_delta=access_token_expires)
    return {"access_token": access_token, "token_type": "bearer"}
```

Registro de pedidos: /registrar_pedido_entregado
Este endpoint permite registrar un pedido en la base de datos DynamoDB, solo si el usuario tiene un token JWT válido.

```python
# Endpoint para registrar pedidos en DynamoDB
@app.post("/registrar_pedido_entregado", status_code=200)
async def registrar_pedido_entregado(pedido: Pedido, credentials: HTTPAuthorizationCredentials = Depends(security)):
    try:
        # Verificar el token
        token = credentials.credentials
        verify_token(token)  # Valida correctamente el token

        # Convertir a Decimal los precios de los productos
        for producto in pedido.productos:
            producto.precio = Decimal(str(producto.precio))

        # Guardar pedido en DynamoDB
        table_pedidos.put_item(Item=pedido.dict())
        return {"message": "Pedido registrado exitosamente", "pedido": pedido.dict()}
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error al registrar pedido: {str(e)}")
```

Obtener pedidos: /pedidos
Este endpoint devuelve todos los pedidos almacenados en la base de datos, solo si el usuario tiene un token JWT válido

```python
# Endpoint para obtener pedidos desde DynamoDB
@app.get("/pedidos", status_code=200)
async def obtener_pedidos(credentials: HTTPAuthorizationCredentials = Depends(security)):
    try:
        token = credentials.credentials
        verify_token(token)  # Valida correctamente el token

        response = table_pedidos.scan()
        data = response.get('Items', [])
        return {"pedidos": data}
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error al obtener pedidos: {str(e)}")
```

Obtener métricas: /metrics

Este endpoint devuelve las métricas calculadas de los pedidos, como el top 3 de productos más vendidos, el precio promedio por pedido y el acumulado en ventas del día.

```python
# Endpoint para obtener métricas
@app.get("/metrics", status_code=200)
async def obtener_metricas(credentials: HTTPAuthorizationCredentials = Depends(security)):
    try:
        # Verificar el token
        token = credentials.credentials
        verify_token(token)  # Valida correctamente el token

        # Consultar la tabla de pedidos
        response = table_pedidos.scan()
        pedidos = response.get('Items', [])

        # Calcular el top 3 productos más vendidos
        productos_vendidos = {}
        total_ventas_dia = 0
        total_pedidos = 0

        for pedido in pedidos:
            total_pedidos += 1
            for producto in pedido["productos"]:
                producto_nombre = producto["producto"]
                producto_precio = float(producto["precio"])

                # Acumular el total de ventas
                total_ventas_dia += producto_precio

                if producto_nombre in productos_vendidos:
                    productos_vendidos[producto_nombre] += 1
                else:
                    productos_vendidos[producto_nombre] = 1

        # Obtener el top 3 productos más vendidos
        top_3_productos = sorted(productos_vendidos.items(), key=lambda x: x[1], reverse=True)[:3]

        # Calcular el precio promedio por pedido
        precio_promedio_por_pedido = total_ventas_dia / total_pedidos if total_pedidos > 0 else 0

        # Devolver las métricas calculadas
        return {
            "top_3_productos": top_3_productos,
            "precio_promedio_por_pedido": precio_promedio_por_pedido,
            "acumulado_ventas_dia": total_ventas_dia
        }

    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error al obtener métricas: {str(e)}")
```

2.3.7 **Integración con Lambda:** Finalmente, se utiliza Mangum para adaptar esta aplicación de FastAPI y permitir que se ejecute en AWS Lambda a través de API Gateway.

```python
# Adaptador para Lambda
handler = Mangum(app)
```

2.4 Pruebas locales del backend: Durante el desarrollo, se ejecutó el servidor FastAPI localmente para probar el funcionamiento de la AP con el siguiente comando:

```
uvicorn main:app –reload
```

2.5 Preparación para el despliegue en AWS Lambda: Al usar AWS Lambda, es necesario empaquetar las dependencias junto con el código en un archivo ZIP. Esto incluyó:

2.5.1 Copiar todas las dependencias y el código a un directorio llamado package.

2.5.2 Utilizar pip install -t para instalar las dependencias en ese directorio.

```
pip install -r requirements.txt -t package/
```

2.5.3 Copiar el archivo main.py dentro de ese directorio package para tener el código y las dependencias en un solo lugar.

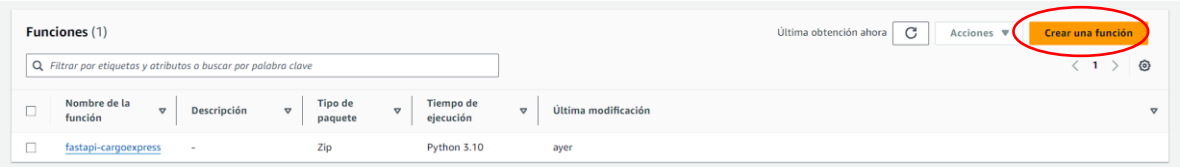
2.5.4 Comprimir el contenido del directorio package en un archivo ZIP:

```
cd package
zip -r ../lambda_function.zip .
```

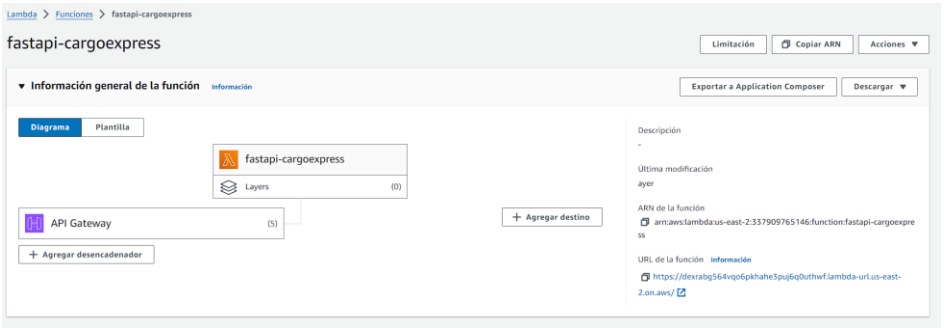
2.6 Despliegue en AWS: A continuación, se mostrarán los pasos realizados para desplegar la solución en la infraestructura de Amazon Web Services (AWS), detallando el uso de AWS Lambda, DynamoDB, API Gateway y la gestión de roles IAM**.**

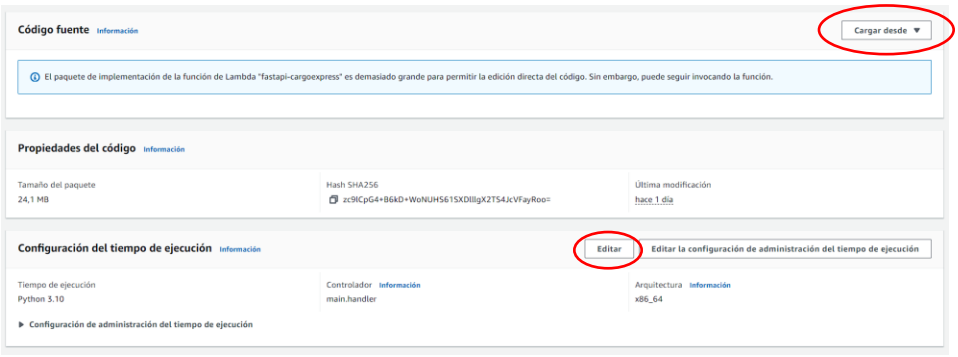
2.6.1 **Despliegue del backend en AWS Lambda:** AWS Lambda fue utilizada para alojar el backend sin necesidad de gestionar servidores. Esta función ejecuta el código del backend bajo demanda, escalando automáticamente según el volumen de tráfico. Desde la pestaña de la función lambda a crear se deben realizar los siguientes pasos:

Crear una función nueva desde la ventana “Funciones” de AWS Lambda dando click en el botón “crear una función”. En la siguiente imagen se puede apreciar la función que ya creamos llamada “fastapi-cargoexpress”:

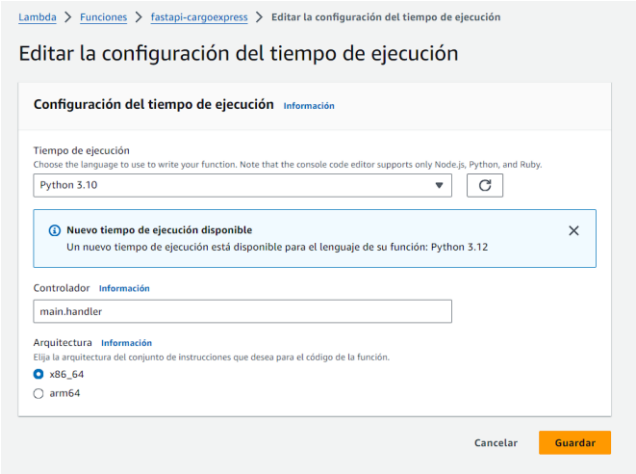


Cargar el archivo .zip que contiene el código fuente y las dependencias (incluyendo librerías como FastAPI, Mangum y boto3). Este se sube a AWS Lambda desde el botón “Cargar desde” mostrado abajo.





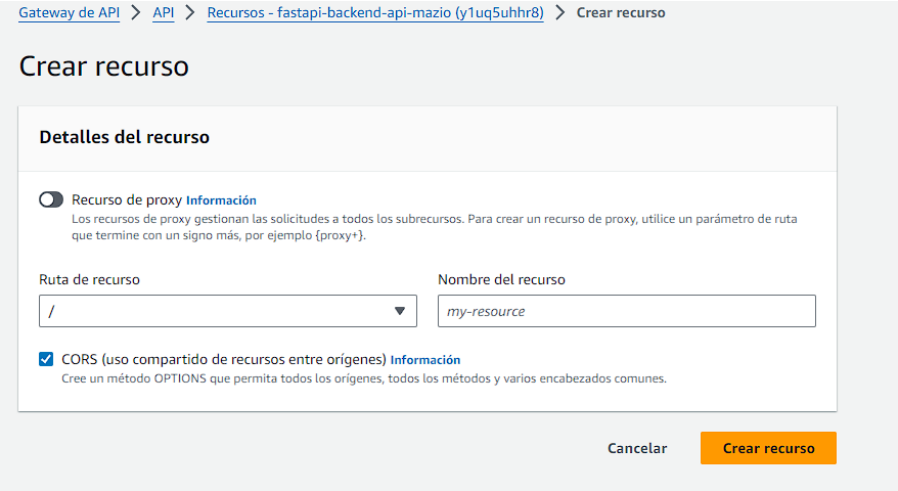
Dando click en el botón “editar”, se abre la siguiente ventana en la cual es importante definir la versión de Python usada para desarrollar el backend, en este caso, Python 3.10, así como una arquitectura x86_64 y también cambiar el nombre del handler que se pone por defecto a “main.handler” para que ejecute nuestro código “main.py”



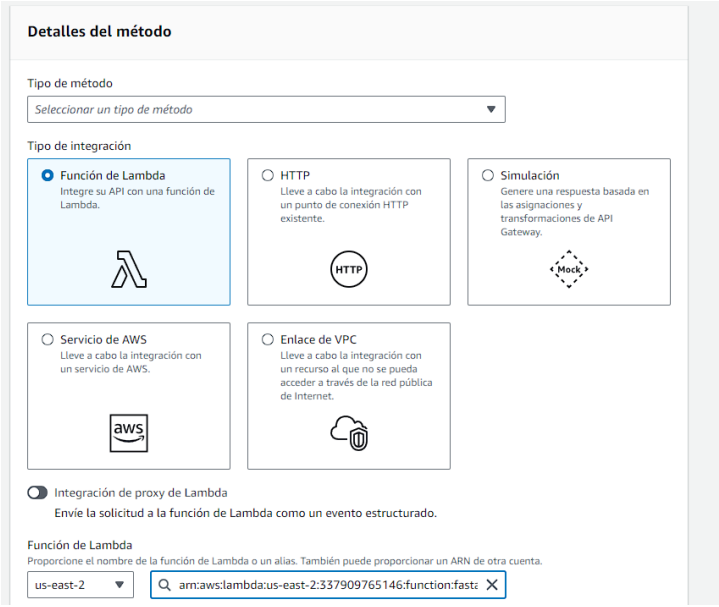
2.6.2 Creación de un API Gateway como intermediario: Se utilizó API Gateway para exponer públicamente las rutas del backend de la aplicación. Esto permite que los usuarios y el frontend puedan interactuar con el backend alojado en Lambda mediante solicitudes HTTP.

Para esto, los pasos a seguir son:

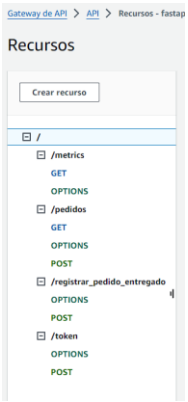
Clickear en el botón “Crear recurso” desde la pestaña y añadir cada uno de los endpoints definidos en nuestro backend (/token, /pedidos, /metrics y /registrar_pedido_entregado). Habilitar el check de CORS para permitir que las solicitudes provenientes del frontend en AWS S3 sean aceptadas más adelante.



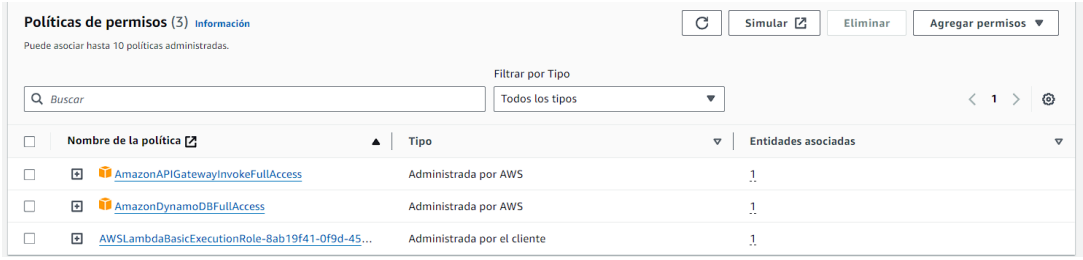
Para cada uno de los recursos creados, es necesario crear los métodos HTTP que usa cada uno, dando click en el botón “Crear método” que lleva a la siguiente pestaña:



Aquí se debe seleccionar el tipo de método (GET, POST, etc) y asociar la ARN de la función lambda que creamos anteriormente, tal como se muestra en la imagen de arriba. Una vez concluido este proceso, la configuración del API Gateway debería quedar así, con los siguientes recursos y métodos:



2.6.3 Asignación de los roles y políticas IAM: Se crearon y configuraron Roles IAM para asegurar que cada servicio tuviera los permisos adecuados sin sobreexponer la seguridad.



Se permitió acceso completo a al servicio de DynamoDB para la base de datos, así como a la función lambda creada y el gateway.

Se adjuntó también la siguiente política en formato JSON:

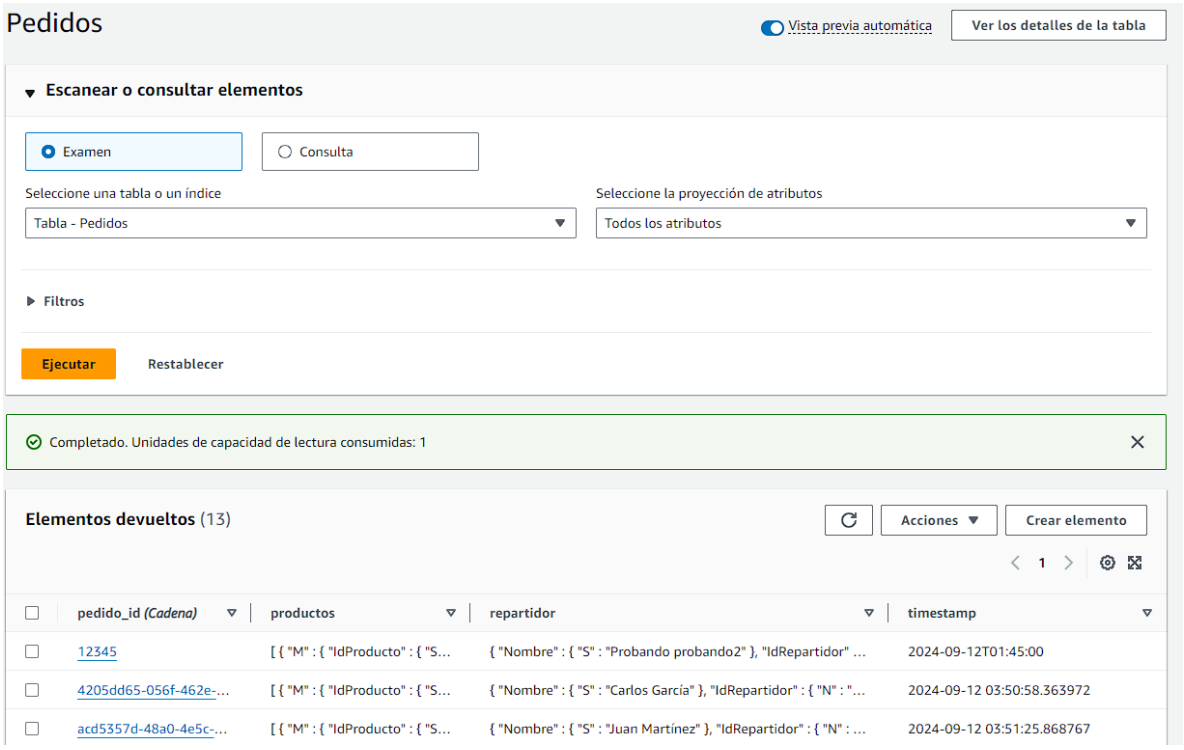
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "lambda:InvokeFunctionUrl",
      "Resource":
"arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
      "Principal": "*"
    }
  ]
}
```

Donde Account_ID es la cuenta de AWS y Function_name la ARN de la función lambda.

2.6.4 Creación de bases de datos NoSQL en DynamoDB: Se crearon dos tablas que son consultadas desde el backend. DynamoDB, al ser un servicio de base de datos NoSQL gestionado por AWS, ofrece escalabilidad y alta disponibilidad sin la necesidad de administrar infraestructuras.

Usuarios: Tabla para almacenar los datos de autenticación de los usuarios. Los campos clave incluyen username y password.
Pedidos: Tabla para almacenar la información de los pedidos registrados. Los campos clave incluyen pedido_id, detalles del repartidor, productos y timestamp.

La función Lambda interactúa con DynamoDB utilizando boto3, el SDK de AWS para Python. Las operaciones principales incluyen la consulta de usuarios y la inserción/lectura de datos sobre los pedidos.



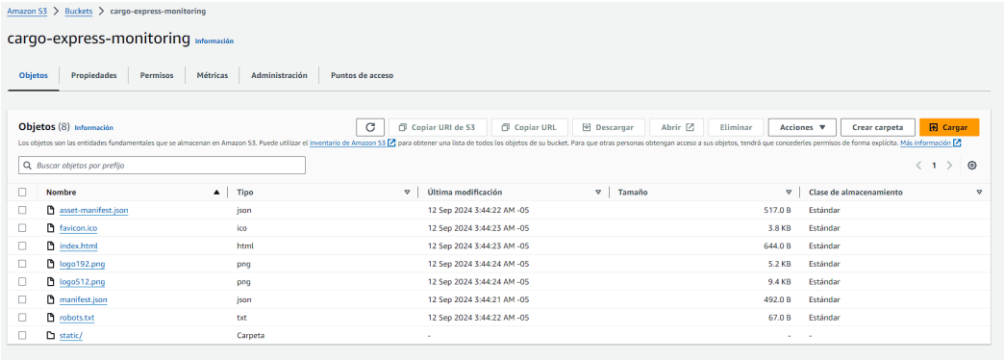
2.6.5 **Despliegue del frontend en S3:** Para la aplicación web, Se desplegó la interfaz de usuario como un sitio web estático en un bucket de Amazon S3. Dicho frontend fue desarrollado en Node.js utilizando React como framework para la creación de la interfaz de usuario. React permite una experiencia de usuario fluida y dinámica al actualizar solo los componentes necesarios sin recargar toda la página.

Se integraron fetch requests para interactuar con el backend a través de API Gateway, permitiendo que los usuarios realicen acciones como el inicio de sesión (autenticación con JWT), la visualización de pedidos registrados y el monitoreo de métricas en tiempo real.

```
const handleLogin = async (e) => {
  e.preventDefault();
  try {
    const response = await fetch('https://dexrabg564vqo6pkhahe3puj6q0uthwf.lambda-url.us-east-2.on.aws/token', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ username, password }),
    });

    const data = await response.json();
    if (response.ok) {
      setAuthToken(data.access_token);
      localStorage.setItem('token', data.access_token); // Guardar el token en localStorage
    } else {
      setError('Credenciales incorrectas');
    }
  } catch (err) {
    setError('Error en el servidor');
  }
};
```

El frontend fue empaquetado mediante Webpack y luego desplegado en un bucket de Amazon S3 como un sitio web estático, asegurando alta disponibilidad y escalabilidad sin necesidad de servidores dedicados. Los archivos cargados fueron:



3. **Pruebas de funcionamiento:** Para verificar el correcto funcionamiento de los endpoints tras haber desplegado el backend en AWS Lambda, se utilizó la consola de prueba para generar eventos JSON y verificar si las respuestas eran las esperadas:

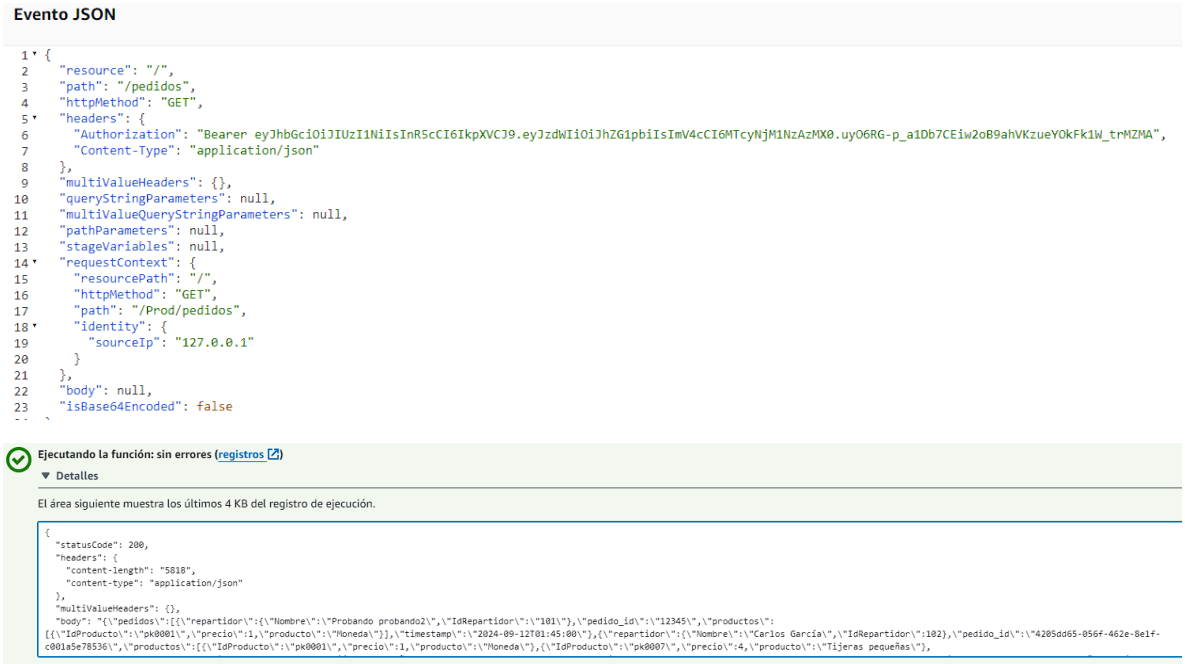
Endpoint /token:

Evento JSON

```
{
  "resource": "/",
  "path": "/token",
  "httpMethod": "POST",
  "headers": {
    "Content-Type": "application/json"
  },
  "multiValueHeaders": {},
  "queryStringParameters": null,
  "multiValueQueryStringParameters": null,
  "pathParameters": null,
  "stageVariables": null,
  "requestContext": {
    "resourcePath": "/",
    "httpMethod": "POST",
    "path": "/Prod/",
    "identity": {
      "sourceIp": "127.0.0.1"
    }
  },
  "body": "{\"username\": \"admin\", \"password\": \"admin\"}",
  "isBase64Encoded": false
```

Se obtiene el token de manera correcta, el cual usaremos en las siguientes pruebas.

Enpoint /pedidos:



Se obtienen los elementos de la tabla “Pedidos” creada en DynamoDB correctamente.

Endpoint /registrar_pedido_entregado:

Ejecutando la función: sin errores (registros)

Detalles

El área siguiente muestra los últimos 4 KB del registro de ejecución.


```
{
  "statusCode": 200,
  "headers": {
    "content-length": "241",
    "content-type": "application/json"
  },
  "multiValueHeaders": {},
  "body": "{\"message\":\"Pedido registrado exitosamente\",\"pedido\":{\"pedido_id\":\"12345\",\"repartidor\":{\"IdRepartidor\":\"101\",\"Nombre\":\"María López\"},\"productos\":
[{\"IdProducto\":\"pk0001\",\"producto\":\"Moneda\",\"precio\":1.0}],\"timestamp\":\"2024-09-12T01:45:00\"}}",
  "isBase64Encoded": false
}
```


Se registra correctamente en la base de datos el pedido enviado en el JSON.

Endpoint /metrics:

Evento JSON


```
{
  "resource": "/metrics",
  "path": "/metrics",
  "httpMethod": "GET",
  "headers": {
    "Authorization": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhZG1pbiIsImV4cCI6MTcyNjM1NzQ3N30.Y7mBAwejbJ1fpysEjgMOw04VBktisOzVCx0a2_uQte0",
    "Content-Type": "application/json"
  },
  "multiValueHeaders": {},
  "queryStringParameters": null,
  "multiValueQueryStringParameters": null,
  "pathParameters": null,
  "stageVariables": null,
  "requestContext": {
    "resourcePath": "/metrics",
    "httpMethod": "GET",
    "path": "/Prod/metrics",
    "identity": {
      "sourceIp": "127.0.0.1"
    }
  },
  "body": null,
  "isBase64Encoded": false
}
```


Ejecutando la función: sin errores (registros)

Detalles

El área siguiente muestra los últimos 4 KB del registro de ejecución.


```
{
  "statusCode": 200,
  "headers": {
    "content-length": "154",
    "content-type": "application/json"
  },
  "multiValueHeaders": {},
  "body": "{\"top_3_productos\":[[\"Moneda\",9],[\"Tarjeta SIM\",7],[\"Estuche para gafas\",7]],\"precio_promedio_por_pedido\":20.676923076923078,\"acumulado_ventas_dia\":268.8}",
  "isBase64Encoded": false
}
```


Se obtiene correctamente el top 3 de los productos más vendidos, el precio promedio por pedido y el acumulado de ventas del día.

Se empleó también el archivo script.py proporcionado para enviar pedidos periódicamente, realizando una modificación al código para que obtuviera primero el token necesario para conectarse a los endpoints:

```python
def obtener_token():
    url = "https://dexrabg564vqo6pkhahe3puj6q0uthwf.lambda-url.us-east-2.on.aws/token"
    payload = {
        "username": [redacted]
        "password": [redacted]
    }
    headers = {"Content-Type": "application/json"}

    response = requests.post(url, json=payload, headers=headers)
    if response.status_code == 200:
        print("Token obtenido correctamente.")
        return response.json()['access_token']
    else:
        print("Error al obtener el token")
        return None
```

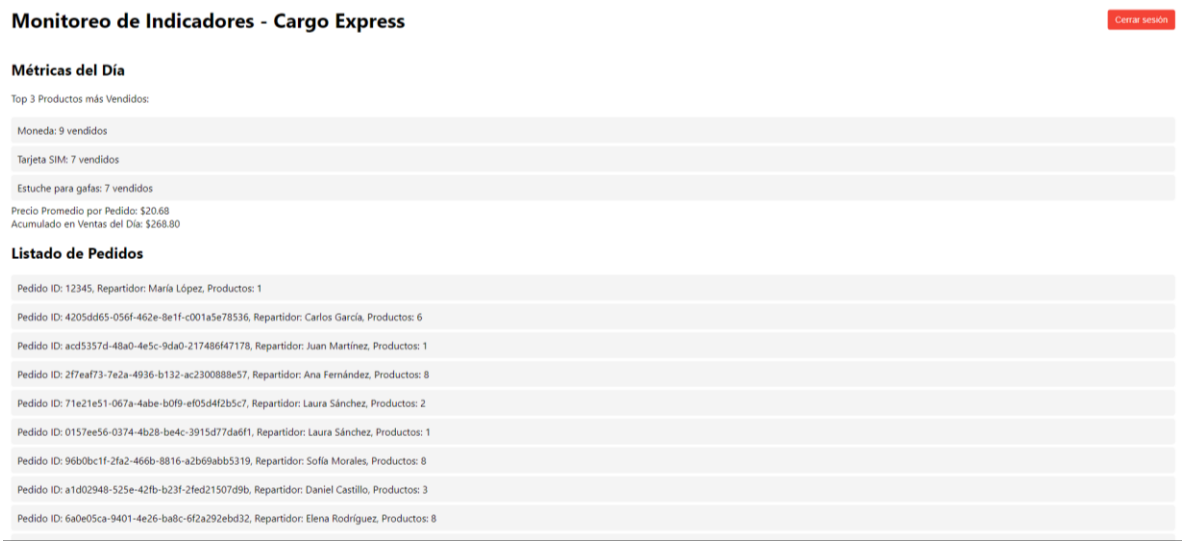
Con la función **obtener_token** , se realiza un login en el backend para que este devuelva el token y de esta manera ya se autoriza la lectura y escritura de registros en las bases de datos de DynamoDB.

```python
# Obtener el token al inicio
token = obtener_token()

if token:
    while True:
        pedido_id = str(uuid.uuid4())
        repartidor = random.choice(repartidores)
        productos_seleccionados = random.choices(productos, k=random.randint(1, 10))

        registrar_pedido_entregado(pedido_id, repartidor, productos_seleccionados, token)
        time.sleep(5)
else:
    print("No se pudo obtener el token. Verifica las credenciales.")
```

Si el token es válido, se procede al envío aleatorio de productos. Accediendo a la URL del frontend pudimos comprobar el correcto funcionamiento de toda la solución desplegada:



4. **Preguntas teóricas:**

La empresa tiene una proyección de expansión en sus operaciones de un 500% en el próximo año y la duplicará en el segundo año. ¿Qué recomendaciones le darías para que pueda garantizar su operación?

- **Uso de servicios serverless:** Lambda se ajusta automáticamente al número de solicitudes. La solución planteada, al ser serverless, permite escalar horizontalmente sin necesidad de administrar servidores. A medida que crezca la demanda, AWS Lambda podrá atender miles de solicitudes simultáneas. Esta capacidad permite que el backend maneje el crecimiento proyectado sin intervención manual, lo que responde bien al aumento en el volumen de pedidos y usuarios.
- **Optimización de API Gateway**: API Gateway también escala automáticamente, pero es recomendable revisar los límites de solicitudes por segundo y el plan de precios para asegurar que pueda manejar el tráfico incrementado. Se puede implementar throttling para evitar que un pico inesperado sobrecargue el sistema.
- **Configurar la escalabilidad de las bases de datos:** Con la correcta configuración de provisión de capacidad en DynamoDB mediante las opciones de On-Demand Capacity, la base de datos puede escalar dinámicamente según la demanda.
- **Monitoreo y alertas:** Implementar herramientas de monitoreo, como AWS CloudWatch, para visualizar y recibir alertas sobre el rendimiento y el uso de los recursos en tiempo real. Esto permitirá detectar cuellos de botella antes de que afecten la operación.
- Es crucial realizar pruebas de carga periódicas para simular los volúmenes de tráfico esperados en los próximos dos años y asegurar que la infraestructura propuesta pueda soportarlo. También se deben realizar pruebas de degradación controlada para ver cómo el sistema se comporta bajo presión.
- Con un mayor volumen de solicitudes y datos, los costos también crecerán. Revisar y optimizar las configuraciones de Lambda, API Gateway, y DynamoDB en función del patrón de uso puede generar ahorros significativos. Se puede optar por una

combinación de tarifas por demanda y provisión reservada, según las previsiones de crecimiento.

¿La solución planteada se encuentra en la capacidad de responder la demanda durante los próximos dos años?

Sí, la solución planteada, basada en servicios serverless como Lambda, API Gateway, y DynamoDB, está diseñada para ser altamente escalable y elástica, lo que significa que puede crecer junto con la demanda sin comprometer la estabilidad. Los servicios de AWS utilizados en la solución pueden manejar incrementos masivos en el tráfico y las operaciones sin la necesidad de reconfigurar infraestructura subyacente.

No obstante, es importante monitorear el crecimiento y realizar ajustes periódicos en la configuración para optimizar tanto el rendimiento como los costos, a medida que la empresa expanda sus operaciones.