Colegio de Ciencias e Ingeniería

Ingeniería en Ciencias de la Computación

NRC: 1439

Proyecto Final

Integrantes:

Gabriel Oña - 320597

Elian García - 323141

16 de diciembre del 2024

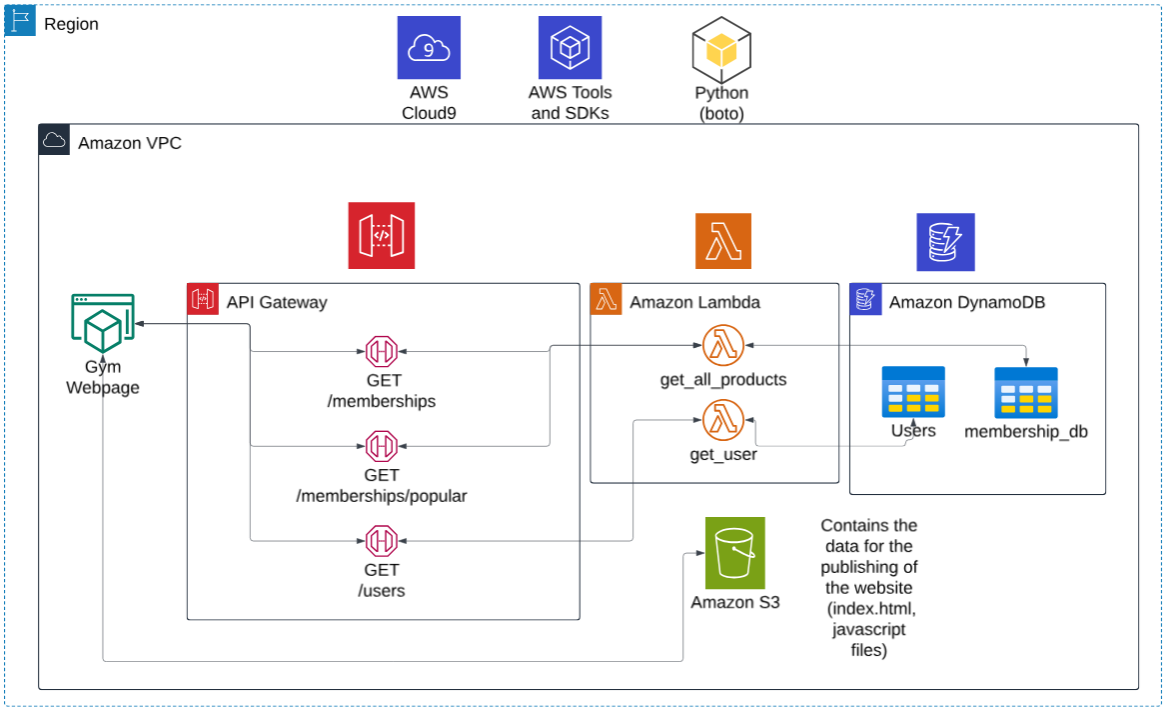
1. Introducción

En el proyecto final, se desarrolló una página web alojada en la nube de AWS, diseñada específicamente para un gimnasio. Esta página presenta un diseño visual atractivo que incluye imágenes generadas con DALL-E. El usuario puede visualizar un listado de membresías, generado dinámicamente mediante etiquetas. Este listado se obtiene a través de una solicitud GET a una REST API, que consulta una base de datos para devolver las membresías disponibles en el gimnasio.

Además, se implementó una funcionalidad para alternar entre vistas al hacer clic en el botón "POPULARES". En esta sección, únicamente se muestran las membresías marcadas como populares, según los datos obtenidos de la misma REST API. El sistema también incluye una funcionalidad de LOGIN. Al ingresar el nombre de usuario y la contraseña, se envía una solicitud a la REST API de usuarios. Una vez autenticado, el sistema devuelve las membresías a las que el usuario en cuestión está suscrito, proporcionando una experiencia personalizada y dinámica.

El propósito de este proyecto es proporcionar una automatización sencilla y ágil para desplegar la página web de un gimnasio utilizando los servicios de AWS con solo unos pocos clics. A través del uso de un archivo setup.sh, se logra implementar el entorno diseñado en la nube mediante AWS CLI y AWS SDK.

2. Arquitectura de la Página Web



3. AWS SDK Implementación Detallada

Como se explicó anteriormente, para automatizar el despliegue de la aplicación, sin la necesidad de usar la consola de AWS, se optó por crear un archivo BASH que ejecutará los comandos necesarios para este propósito. A continuación, se detalla cada uno del código implementado.

```
#!/bin/bash
#the two lines below are new for Amazon Linux2
sudo yum -y remove python37
sudo amazon-linux-extras install -y python3.8
sudo update-alternatives --set python /usr/bin/python3.8
sudo ln -sf /usr/bin/python3.8 /usr/bin/python3
sudo ln -sf /usr/bin/python3.8 /usr/bin/python
sudo ln -sf /usr/bin/pip3.8 /usr/bin/pip
sudo ln -sf /usr/bin/pip3.8 /usr/bin/pip3
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
rm awscliv2.zip
pip install boto3
```

Figura 1. Instalación previa

En este apartado se está realizando la actualización de Python, la instalación del AWS CLI para poder utilizar comandos de AWS en el terminal y se instala el SDK de AWS (BOTO3) con pip de Python.

```
echo Please enter a valid IP address:
read ip_address
echo IP address:$ip_address
echo Please wait...
#sudo pip install --upgrade awscli
bucket="cloud-computing-gym-project-bucket-webpage"
AWS_ACCOUNT_ID=$(aws sts get-caller-identity --query "Account" --output text)

FILE_PATH="/home/ec2-user/environment/gym-webpage/resources/public_policy.json"
FILE_PATH_2="/home/ec2-user/environment/gym-webpage/resources/permissions.py"
FILE_PATH_3="/home/ec2-user/environment/gym-webpage/resources/website/config.js"
FILE_PATH_4="/home/ec2-user/environment/gym-webpage/python_3/update_config.py"

sed -i "s/<FMI_1>/$bucket/g" $FILE_PATH
sed -i "s/<FMI_1>/$bucket/g" $FILE_PATH_4
sed -i "s/<FMI_2>/$ip_address/g" $FILE_PATH
sed -i "s/<FMI>/$bucket/g" $FILE_PATH_2
```

Figura 2. Variables y modificaciones

En este apartado, se pide al usuario su dirección IP pública para restringir el acceso a la página web de solo los paquetes de red que provengan de esta dirección IP. Agregamos variables del sistema el nombre del bucket de s3 que hosteará la página. Es importante recalcar que este nombre tiene que ser único en el mundo ya que no puede existir dos buckets con mismo nombre en la infraestructura de AWS. Finalmente, se ubica algunos archivos para remplazar con el comando “sed” las variables anteriormente creadas. El archivo public_policy.json se encarga de

definir las restricciones IP que tendrá el bucket s3. El archivo permissions utiliza esta política y la otorga al bucket. El archivo config.js contiene la URL del API Gateway a la cual se va a apuntar en todos los request que se haga, y finalmente el archivo update_config carga en el bucket precreado el archivo config.js.

```
#DISABLE PAGER
aws configure set cli_pager ""

#Create bucket s3 for webpage
aws s3 mb s3://$bucket
#Create apigateway
python ~/environment/gym-webpage/python_3/create_products_api.py
python ~/environment/gym-webpage/python_3/create_popular_api.py
python ~/environment/gym-webpage/python_3/create_user_get_api.py

apigateway=`aws apigateway get-rest-apis | grep id | cut -f2- -d: | tr -d ',' | xargs`
aws apigateway create-deployment \
    --rest-api-id $apigateway \
    --stage-name prod
invoke_url="https://$apigateway.execute-api.us-east-1.amazonaws.com/prod"

sed -i "s@<FMI_1>@$invoke_url@g" $FILE_PATH_3
python ~/environment/gym-webpage/python_3/update_config.py
```

Figura 3. Api Gateway CLI

En este apartado se desactiva pager en el terminal para no inhabilitar la ejecución del código. Se crea el bucket s3 y se crea la API. En nuestra aplicación, utilizamos el nombre MembershipsAPI con 3 métodos.

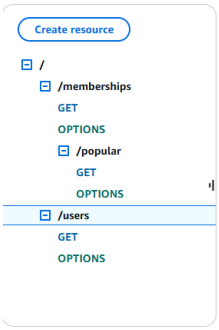


Figura 4. API methods

Posteriormente, se define el id de la API creada y se crea el stage de producción. Finalmente, se modifica el archivo config.js con el nuevo URL para el API de producción y se actualiza el bucket.

```
#Create dynamoDB
python ~/environment/gym-webpage/python_3/create_table.py
python ~/environment/gym-webpage/python_3/add_gsi.py
cd ~/environment/gym-webpage/python_3
zip get_all_products_code.zip get_all_products_code.py
zip get_all_users_code.zip get_all_users_code.py

aws s3 cp ~/environment/gym-webpage/resources/website s3://$bucket/ --recursive --cache-control "max-age=0"
aws s3 cp ~/environment/gym-webpage/python_3/get_all_products_code.zip  s3://$bucket
aws s3 cp ~/environment/gym-webpage/python_3/get_all_users_code.zip  s3://$bucket

python ~/environment/gym-webpage/resources/permissions.py
python ~/environment/gym-webpage/resources/seed.py
```

Figura 5. Creación Tablas DynamoDB

```
import boto3

def create_table():

    DDB = boto3.resource('dynamodb', region_name='us-east-1')

    params = {
        'TableName': 'memberships_db',
        'KeySchema': [
            {'AttributeName': 'membership_name', 'KeyType': 'HASH'}
        ]
        'AttributeDefinitions': [
            {'AttributeName': 'membership_name', 'AttributeType': 'S'}
        ]
        'ProvisionedThroughput': {
            'ReadCapacityUnits': 1,
            'WriteCapacityUnits': 1
        }
    }
    table = DDB.create_table(**params)
    table.wait_until_exists()
    print ("Done table memberships")

    params = {
    'TableName': 'users',
    'KeySchema': [
        {'AttributeName': 'user_name', 'KeyType': 'HASH'}  # Define HASH key
    ],
    'AttributeDefinitions': [
        {'AttributeName': 'user_name', 'AttributeType': 'S'}  # Match HASH key type
    ],
    'ProvisionedThroughput': {
        'ReadCapacityUnits': 1,
        'WriteCapacityUnits': 1
    }

    }
    table = DDB.create_table(**params)
```

Figura 6. AWS SDK DynamoDB

Para este apartado se crea las tablas de membresías y usuarios en dynamoDB mediante el AWS SDK. Como se puede apreciar, el SDK es muy parecido al CLI solo que se lo ejecuta como código de Python. Las ventajas de usar SDK es el mantener aislado y modular códigos para la creación del entorno, mientras el CLI tiene que estar todo en el SETUP.sh

Name	Status	Partition key	Sort key	Indexes
memberships_db	Active	membership_name (S)	-	1
users	Active	user_name (S)	-	0

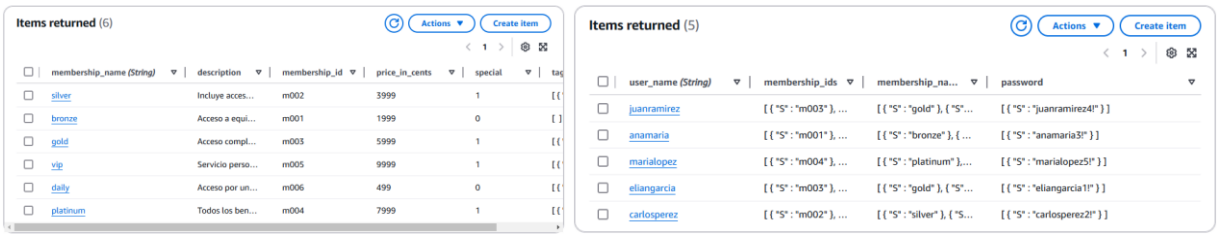


Figura 7. Tablas DynamoDB

Adicionalmente, para la tabla memberships se lo indexa con add_GSI.py el cual va a tomar la clase ‘special’ como otro index. Finalmente, se comprime los códigos de Python que se encargan en el bucket de S3 de pedir a las dos bases de datos información y transformarlos en JSON respose adecuadamente.

```
FILE_PATH_5="/home/ec2-user/environment/gym-webpage/python_3/get_all_products_wrapper.py"
FILE_PATH_6="/home/ec2-user/environment/gym-webpage/python_3/get_all_users_wrapper.py"

sed -i "s@<FMI_ROLE>@$ROLE_ARN@g" $FILE_PATH_5
sed -i "s@<FMI_ROLE>@$ROLE_ARN@g" $FILE_PATH_6
sed -i "s@<FMI_BUCKET>@$bucket@g" $FILE_PATH_5
sed -i "s@<FMI_BUCKET>@$bucket@g" $FILE_PATH_6

python ~/environment/gym-webpage/python_3/get_all_products_wrapper.py
python ~/environment/gym-webpage/python_3/get_all_users_wrapper.py

MEMBERSHIPS_RESOURCE_ID=$(aws apigateway get-resources --rest-api-id $apigateway --query "items[?path=='/memberships'].id" --output text)
POPULAR_RESOURCE_ID=$(aws apigateway get-resources --rest-api-id $apigateway --query "items[?path=='/memberships/popular'].id" --output text)
USERS_RESOURCE_ID=$(aws apigateway get-resources --rest-api-id $apigateway --query "items[?path=='/users'].id" --output text)
LAMBDA_ARN=$(aws lambda get-function --function-name get_all_products --query 'Configuration.FunctionArn' --output text)
LAMBDA_USERS_ARN=$(aws lambda get-function --function-name get_user --query 'Configuration.FunctionArn' --output text)
```

Figura 8. Wrappers Lambda

En este apartado se ejecuta los wrappers de las funciones lambda cargadas en s3 para que AWS genere las funciones al usar su AWS SDK para crear Lambda. Finalmente, se generan nuevas variables como los resource id de cada método del API, así como los ARN de las dos funciones lambda.

```
aws lambda add-permission \
    --region us-east-1 \
    --function-name get_all_products \
    --principal apigateway.amazonaws.com \
    --statement-id apigateway-get-memberships \
    --action "lambda:InvokeFunction" \
    --source-arn arn:aws:execute-api:us-east-1:$AWS_ACCOUNT_ID:$apigateway/*/GET/memberships

aws lambda add-permission \
    --region us-east-1 \
    --function-name get_all_products \
    --principal apigateway.amazonaws.com \
    --statement-id apigateway-get-popular \
    --action "lambda:InvokeFunction" \
    --source-arn arn:aws:execute-api:us-east-1:$AWS_ACCOUNT_ID:$apigateway/*/GET/memberships/popular

aws lambda add-permission \
    --region us-east-1 \
    --function-name get_user \
    --principal apigateway.amazonaws.com \
    --statement-id apigateway-get-user \
    --action "lambda:InvokeFunction" \
    --source-arn arn:aws:execute-api:us-east-1:$AWS_ACCOUNT_ID:$apigateway/*/GET/users
```

Figura 9. Lambda IAM permission

En este apartado se está otorgando los privilegios a las funciones lambda para que se vinculen con el respectivo método de API Gateway.

```
aws apigateway put-integration \
        --region us-east-1 \
        --rest-api-id $apigateway \
        --resource-id $MEMBERSHIPS_RESOURCE_ID \
        --http-method GET \
        --type AWS \
        --integration-http-method POST \
        --uri arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/$LAMBDA_ARN/invocations \
        --passthrough-behavior WHEN_NO_TEMPLATES \
        --timeout-in-millis 29000


aws apigateway put-integration-response \
        --region us-east-1 \
        --rest-api-id $apigateway \
        --resource-id $MEMBERSHIPS_RESOURCE_ID \
        --http-method GET \
        --status-code 200 \
        --selection-pattern "" --response-templates '{"application/json": ""}'
```

```
aws apigateway put-integration-response \
        --region us-east-1 \
        --rest-api-id $apigateway \
        --resource-id $POPULAR_RESOURCE_ID \
        --http-method GET \
        --status-code 200 \
        --selection-pattern "" \
        --response-templates '{"application/json": "{\"membership_item_arr\": [#foreach($item in $input.path(\"$.membership_item_arr\")) #if($item.

aws apigateway put-integration \
        --region us-east-1 \
        --rest-api-id $apigateway \
        --resource-id $USERS_RESOURCE_ID \
        --http-method GET \
        --type AWS \
        --integration-http-method POST \
        --uri arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/$LAMBDA_USERS_ARN/invocations \
        --passthrough-behavior WHEN_NO_TEMPLATES \
        --timeout-in-millis 29000 \
        --request-templates '{"application/json": "{\"username\": \"$input.params(\"username\")\", \"password\": \"$input.params(\"password\")\"}"}
```

```
aws apigateway put-integration \
        --region us-east-1 \
        --rest-api-id $apigateway \
        --resource-id $USERS_RESOURCE_ID \
        --http-method GET \
        --type AWS \
        --integration-http-method POST \
        --uri arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/$LAMBDA_USERS_ARN/invocations \
        --passthrough-behavior WHEN_NO_TEMPLATES \
        --timeout-in-millis 29000 \
        --request-templates '{"application/json": "{\"username\": \"$input.params(\"username\")\", \"password\": \"$input.params(\"password\")\"}"}'

aws apigateway put-integration-response \
        --region us-east-1 \
        --rest-api-id $apigateway \
        --resource-id $USERS_RESOURCE_ID \
        --http-method GET \
        --status-code 200 \
        --selection-pattern "" --response-templates '{"application/json": ""}'

aws apigateway create-deployment \
    --rest-api-id $apigateway \
    --stage-name prod
```

Figura 8. API Integrations

En este apartado se indica la integración del método /membership para el pedido a la función lambda, en el cual se identifica esta función con su uri respectivo. Además, el tipo AWS permite vincular a AWS API Gateway con AWS Lambda. Finalmente, se añade también una integración a la respuesta con un mapping de lo que devuelva la función lambda. Este mapping es de tipo json y toma lo el return de lambda y lo configura para enviarlo como JSON devuelta al usuario en el Edge.

4. Simulación e Instrucciones para Setup

Para simularlo se utiliza el laboratorio 7 de AWS Cloud Developing Academy ya que tiene todos los permisos IAM para desplegar el entorno que construimos. Cuando inicia el laboratorio, se tiene que borrar todas las instancias precreadas en S3, API Gateway y DynamoDB. Una vez hecho esto, se abre Cloud9 y se carga el proyecto gym-webpage.

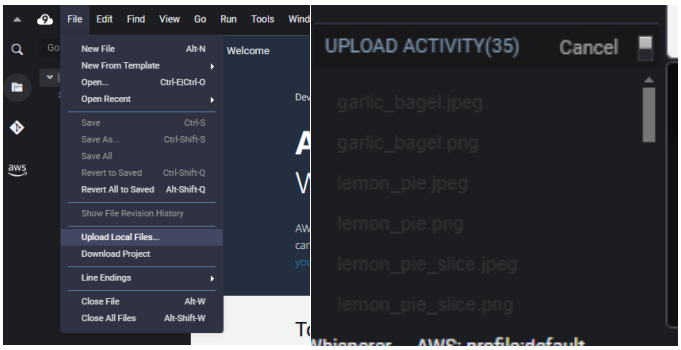


Figura 9. Upload a Cloud9

Posteriormente, se da privilegios al archivo setup.sh y se lo ejecuta. Este archivo pedirá la ip pública de nuestro dispositivo. Finalmente, todo el sistema se desplegará.

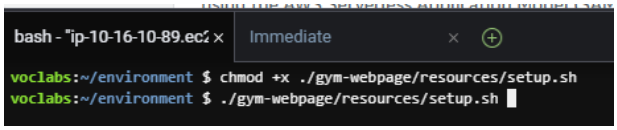


Figura 10. Comandos de ejecución

Es importante notar que hubo dificultades al automatizar CORS, por lo que esta opción se la realiza manualmente. En la consola de AWS se ingresa a API Gateway y se selecciona cada método. Se aplasta “Enable CORS” y se selecciona 4xx, 5xx, y GET.

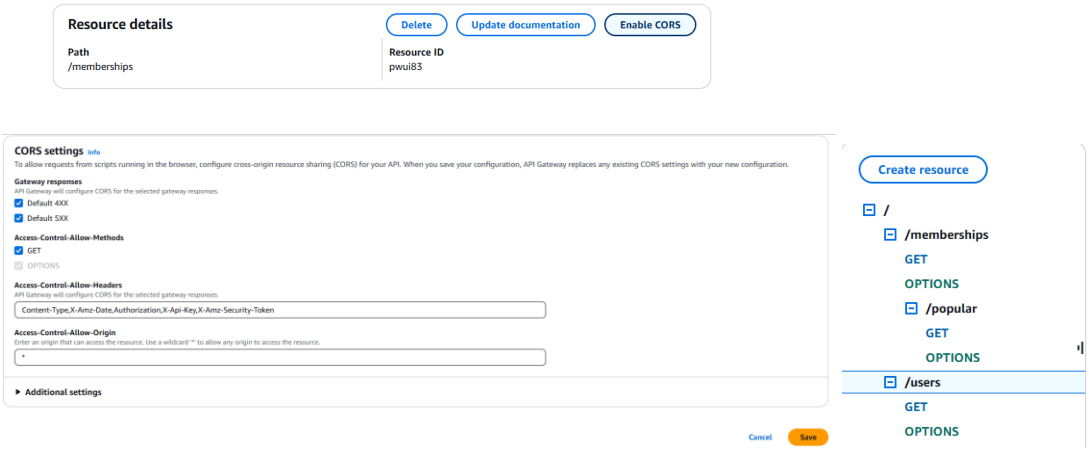
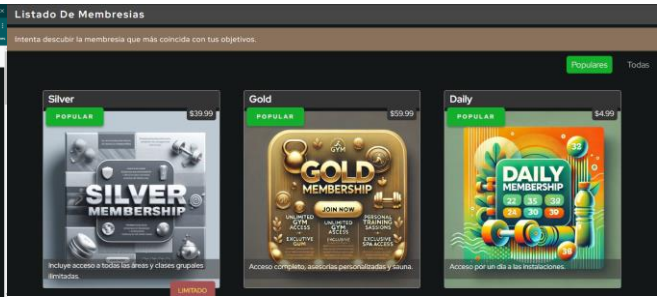


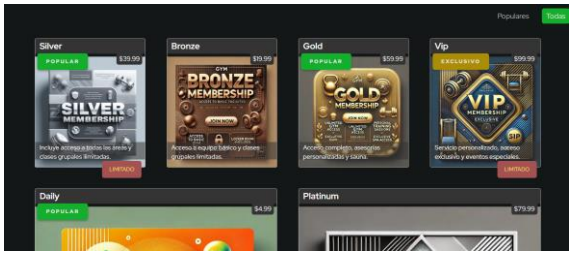
Figura 11. Enable CORS

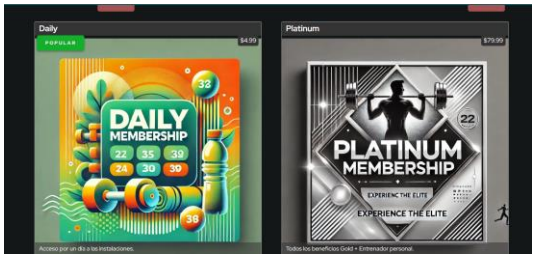
Finalmente, se selecciona deploy API y se escoge /prod.

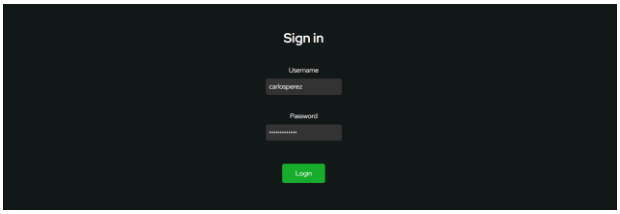
5. Testing Methodology and Results

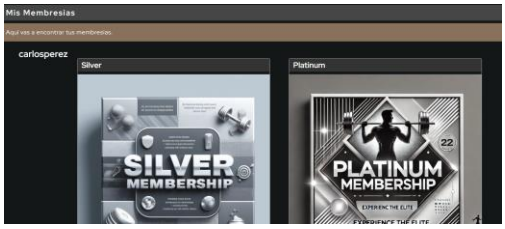
A continuación, se presenta los resultados al abrir index.html en el bucket de s3.

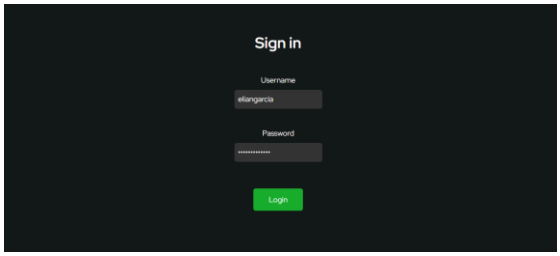


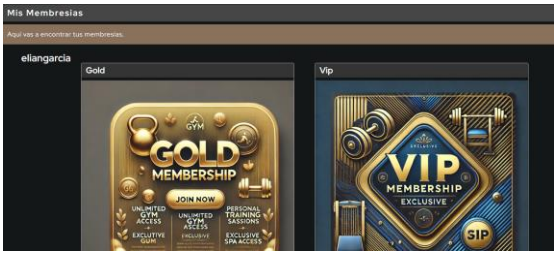






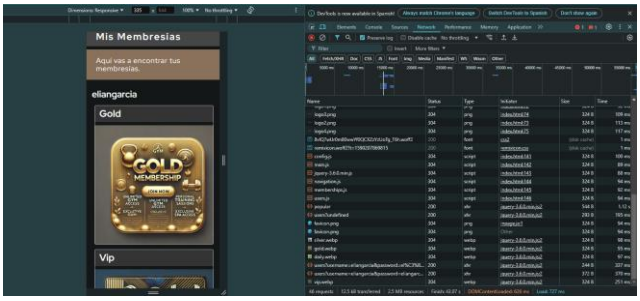




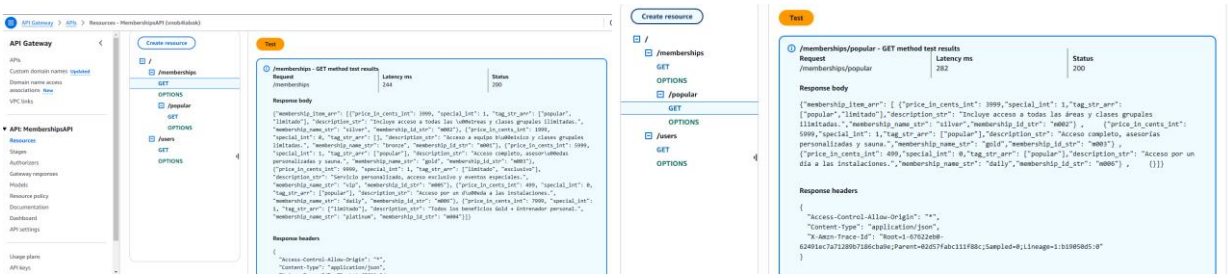


Prueba de elementos:

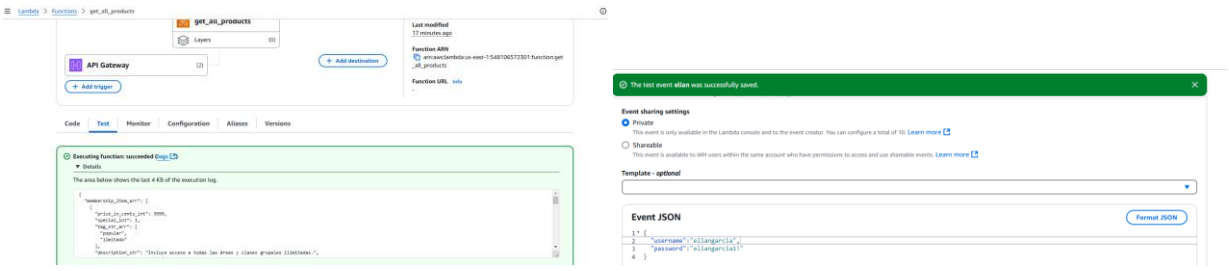
- Request en el EDGE.

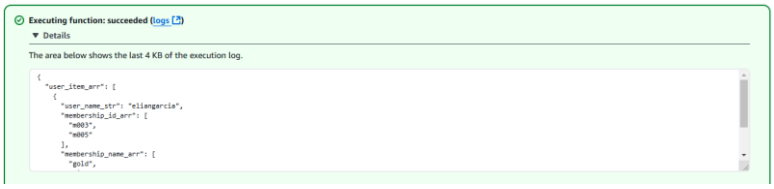


- API gateway test



- Lambda Functions test





6. **Challenges faced and lessons learned**

Durante este proyecto fue complejo manejar AWS CLI de manera local y luego en AWS. Por ello se aprendió que Cloud9 es el entorno perfecto para generar el setup.sh con los comandos del CLI mientras se visualiza los elementos creandoce automaticamente. Para una segunda prueba se tiene que eliminar todos los elementos creados, lo cual es un proceso tedioso.

El manejo de los request con API Gateway tambien fue complejo durante la integración de codigos JS con eventos en AWS. En este proceso se aprendió la importancia de utilizar la

ventana de TEST de respuestas MOCK. Uno de los principales problemas adicionalmente fue habilitar CORS ya que hay muy poca información de como realizar esto en CLI.

Finalmente, otra dificultad radicó en la integración de Mapping de API Gateway con lambda. En este apartado, se aprendio a modificar tanto los templates de integration en respuesta como en pedido para que la función lambda y los códigos de javascript reciban el correcto formato.

En todo este proceso, lo más imporante fue leer la documentación de CLI y SDK que provee AWS para saber los métodos de cada comando.

7. Potential Improvements and future enhacements

Se recomienda para trabajo a futuro el gestionar la autentificación de usuarios con el servicio Cognito de AWS para explotar sus capacidades, e inclusive mejorar la seguridad con respecto a la implementación que se realizo para el proyecto actual. Además, se debería gestionar multiples ventanas entre el login y la página principal para que se genere una plantilla para cada usuario luego de realizar su inicio de sesión.

Adicionalmente, se puede explorar nuevas integraciones de AWS Lambda con multiple permisos de escritura y lectura en las bases de datos. De esta forma, a futuro se podrá integrar nuevos endpoints al API Gateway con diversas posibilidades.

Repositorio GitHub: https://github.com/Gabeinstein/GO-cloud_computing_fall24/tree/main/AWS%20Serverless%20GYM%20Webpage%20Project