

# Assignment 6

ICE191 Software Architecture / Cloud Computing

1. Modify your Lambda function to include the field City for each new record in the Students DynamoDB table. **30 points.**

```
# Esto es un copy paste del script de mi tarea 4 pero modificado para funcionar en lambda
try:
    #Si no nos dio una acción que hacer entonces adios
    option = event['httpMethod']
except:
    return get_response(500,"Corre esta API/lambda con un metodo HTTP correcto")

if(option=='POST' or option=='PUT'): # UPSERT ¿Que sinvergüenza no?
    try:
        # Usaremos de la llamada a la lambda, el usuario matricula, nombre completo y URL del sitio personal
        # En un escenario más normal, hubiera hecho una documentación que dice como usar la lambda, pero en
        # este caso mi tarea será lo unico que habrá
        requestBody = json.loads(event["body"])
        fullname = requestBody["full_name"]
        personalWebsite = requestBody["personal_website"]
        city = requestBody["city"]

        # PUT_ITEM es para meter items en tablas, si ya existe uno lo sobrescribirá
        # esto lo identifica con la llave primaria (detalles en el documento)
        dynamoWriter.put_item(
            TableName="Students", # Necesitamos nombre de tabla
            Item={
                # Llave primaria es obligatoria
                "id": {'S': matricula},
                # Esto y el otro dato son los que ya existian en la tabla
                "full_name": {'S': fullname},
                "personal_website": {'S': personalWebsite},
                "city": {'S': city}
            }
        )
        # En mi tarea 2 explico porque los valores son diccionarios, pero en resumen deben de indicar
        # el tipo de valor del valor, en este caso S es STRING, asi esta definida la tabla
    except Exception as e:
        return get_response(200,"Creación/Actualización terminada")
    except Exception as e:
        # Error 500 generico para decirle al usuario que si salió mal
        return get_response(500, f"Error creando/actualizando estudiantes {str(e)}")

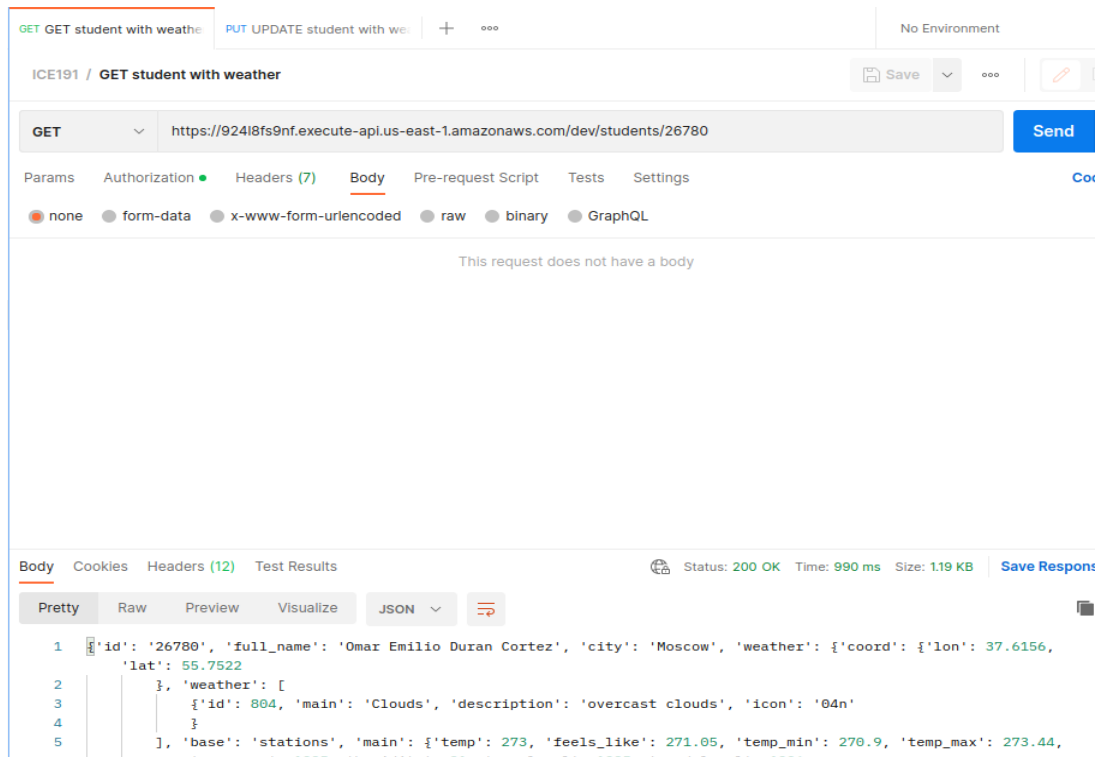
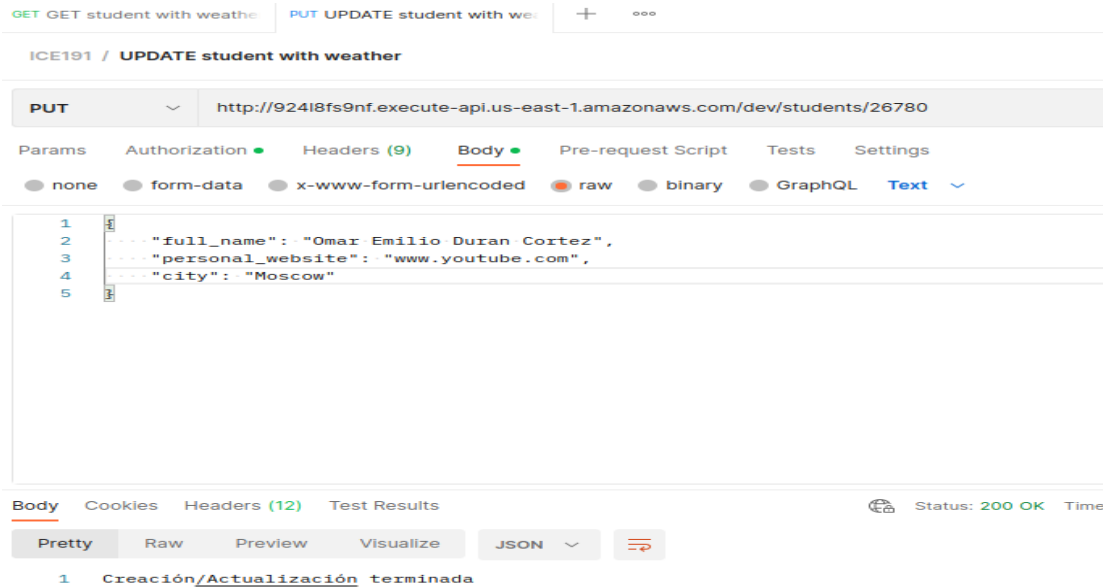
elif(option=="GET"):
```

En la lambda de esta actividad usé para el proceso de escritura mi código de la tarea 5, siendo el UPSERT de mi CRUD. El código sabe si el método fue POST/PUT o GET y dependiendo de eso leerá la entrada del estudiante o escribirá (o actualizará) una nueva. La única modificación notable es que ahora al poner el récord en la tabla también acepta el campo de ciudad, el cual es un string.

Aquí tuve que crear un objeto llamado dynamoWriter, el cual es un cliente de boto3 en vez de un recurso, esto fue para evitar tener que modificar el código ya hecho para lectura, pero para fines del usuario que lo prueba no notará la diferencia.

Otra modificación es que ahora los retornos estan adaptados al código del profesor, pero sigue siendo de tipo PROXY, más detalles de cosas que cambiaron en el ejercicio 2.

A continuación, mostraré imágenes de ejecución, pero cabe destacar que cronológicamente estas tuvieron lugar al acabar el ejercicio 2, por lo que si se desea saber los detalles técnicos de como se hizo la API para esta lambda favor de leer eso primero.



Para poder actualizar mi lambda utilice el código de actualización de código de lambda, más detalles de eso en mi tarea 5. Cabe destacar que se debe enviar la carpeta completa con las dependencias, más detalles de cómo enviar la carpeta completa con dependencias en el ejercicio 2.

2. Modify Read in your Lambda function to return the weather of the city assigned to the Students DynamoDB table record. **30 points.**

```
def get_weather(student, api_key):
    base_url = "http://api.openweathermap.org/data/2.5/weather?q={0}&appid={1}"
    if "city" in student["Item"].keys():
        http = urllib3.PoolManager()
        response = http.request('GET', base_url.format(student["Item"]["city"], api_key))
        return response.data
    else:
        return json.dumps("No city assigned to student")

def get_secret():
    secretsmanager = boto3.client(service_name='secretsmanager')
    secret_name = "weather_api_profe"
    secrets_response = secretsmanager.get_secret_value(SecretId=secret_name)
    return secrets_response['SecretString']
```

En el código del profesor están estas dos funciones, la primera mostrada es la que consigue el clima. Aquí recibe el récord del estudiante, y busca si tiene una Key de clima, si es así hará un request GET a la API del clima con la ciudad en el récord del alumno y la llave de la API.

La llave de la API es obtenida en el bloque try except mencionado en el ejercicio 1, donde se llama a la función `get_secret()`. Aquí se usa el servicio de secretos de AWS para conseguir secretos como contraseñas, llaves, y otros datos que no deberían estar expuestos a cualquiera. De aquí sacaremos el secreto que es la llave de la API para el profesor.

```
def get_response(code, body):
    # El return no estaba completo para ser usado en una API proxy
    # le faltaba headers y "isBase64Encoded", el cual de este ultimo
    # la respuesta es falso
    return {
        "isBase64Encoded": False,
        "statusCode": code,
        "headers": { "headerChido": "ah pero el script estaba bien no B)? : sí"},
        "body": str(body)
    }
```

Al hacer esto el código llamará dentro del bloque try except a `get_response()` que está hecho para retornar una respuesta haya sido buena o mala, si se pudo conseguir el clima y el récord del alumno entonces se llegará aquí con un código 200 y en el cuerpo (body) vendrá el récord del alumno con el clima de su ciudad.

Aquí como se ve el comentario se tuvieron que hacer unas modificaciones menores siendo la falta de dos valores al diccionario de retorno, falta de headers e indicar encodificación de base64.

La otra modificación fue esta:

```
dynamo = boto3.resource("dynamodb")
students_table = dynamo.Table("Students")
# En vez de buscar id directamente
# debemos buscar ID en los path parameters
matricula = event["pathParameters"]["id"]
```

La matrícula viene en los pathParameters, no por sí sola.

En fin, ahora los comandos para crear la lambda con API para este código (explicaciones más detalladas de que hace cada comando en mi tarea 5)

Comando 1:

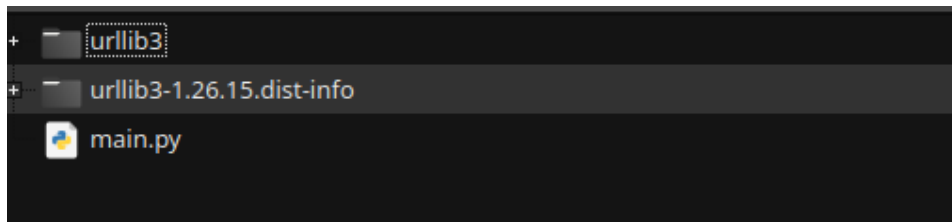
```
aws lambda create-function \
    --function-name weather_lambda_omar \
    --runtime python3.9 \
    --zip-file fileb://weather_lambda_omar.zip \
```

```
--handler weather_lambda_omar.main.lambda_handler \  
--role arn:aws:iam::292274580527:role/lambda_ice191
```

Crear función de lambda con python 3.9. Aquí hay algo distinto, en vez de enviar un zip con un solo archivo, lo enviamos con múltiples cosas dentro, siendo el script, y una librería de solicitudes. Lo otro distinto es el handler, ahora la sintaxis es: NombreDeFolder.NombreDeArchivo.NombreDeFuncion

```
ts % pip3 install --target . urllib3
```

Así se instala la librería dentro de la carpeta en la que estas (ya debes tenerla instalada previamente para poder correr el comando así).



Así se debería ver el folder que enviarás a la lambda. La forma de comprimirlo es esta:

```
zip -r9 weather_lambda_omar.zip weather_lambda_omar
```

Debes usar el parámetro 'r' en zip para que sea recursivo y comprima todo lo de adentro del folder.

Comando 2:

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

Crear la API REST en API Gateway

Comando 3:

```
aws apigateway get-resources --rest-api-id 924l8fs9nf
```

Conseguir los recursos para conseguir la ID del recurso padre '/'

Comando 4:

```
aws apigateway create-resource --rest-api-id 924l8fs9nf --parent-id ydttk04a4 --path-part students
```

Crear un recurso en nuestra API REST el cual será hijo del recurso padre (obtenido en el 3er comando) y la ruta se llamará "students"

Comando 5:

```
aws apigateway create-resource --rest-api-id 924l8fs9nf --parent-id czh3wy --path-part {id}
```

Crear un recurso en nuestra API REST el cual será hijo del recurso "students" (obtenido su ID en el output el 4to comando) y la ruta llamará "{id}". Bueno no, no se llama así, sino que es para indicar que ahí irá el parámetro de "id" con lo que ponga el usuario que haga el request.

Comando 6:

```
aws apigateway put-method --rest-api-id 924l8fs9nf --resource-id kyojoy --http-method GET --authorization-type NONE
```

Ponerle un método GET al recurso "{id}" (su ID se consigue en el output del comando 5) y sin autorización (excepto en el ejercicio 3)

Este recurso estaría en la ruta /students/{id}.

Este comando debe repetirse también para los métodos POST y PUT, por lo que en "http-method" debe cambiar el valor por cada uno de esos

Comando 7:

```
aws apigateway put-integration \  
  --rest-api-id 924l8fs9nf \  
  --resource-id kyojoy \  
  --http-method GET \
```

```
--integration-http-method POST \  
--type AWS_PROXY \  
--uri "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:292274580527:function:weather_lambda_omar/invocations"
```

Crear integración de tipo PROXY (explicación de API proxy y otros parámetros en mi tarea 5) en el recurso "{id}" de tipo GET hacia la lambda del clima mía.

Al igual que en el comando 6, en "http-method" se debe cambiar el valor por cada método HTTP que queremos (en este caso son PUT, POST, y GET) por lo que hay que correr esto dos veces más

Comando 8:

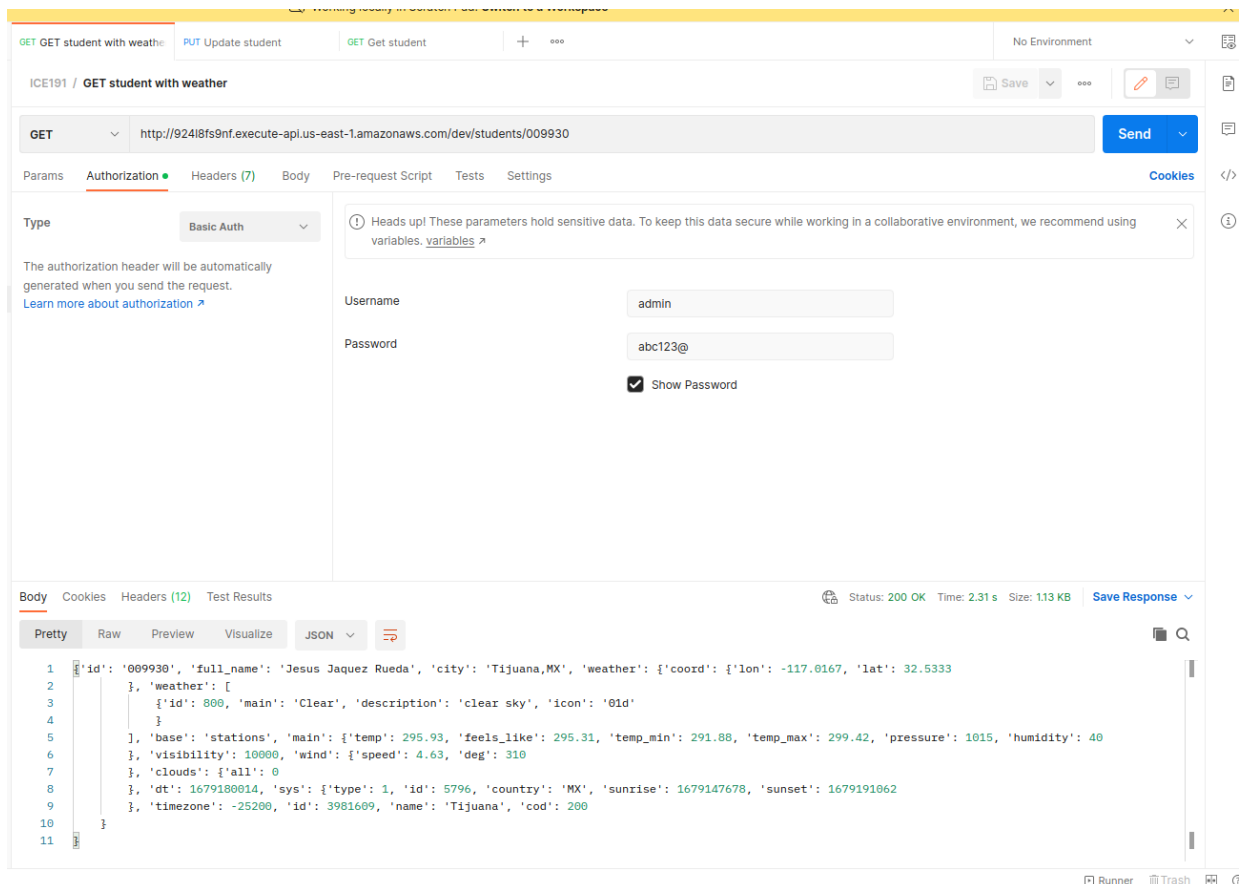
```
aws lambda add-permission --function-name "weather_lambda_omar" \  
--statement-id autorizadorDeLaLambdaDeOmar \  
--action --lambda:InvokeFunction \  
--principal apigateway.amazonaws.com
```

Decirle a nuestra lambda de clima que será invocada por el servicio de API Gateway. Recordatorio que el "statement" es como una descripción/comentario.

Comando 9:

```
aws apigateway create-deployment \  
--rest-api-id 924l8fs9nf \  
--stage-name dev \  
--description "deploy de API de clima de OMAR"
```

Deployear nuestra API REST al nombre de stage de preferencia (a mí me gusta ponerle dev en este caso), y darle de paso una descripción.



Éxito, ahora la API que llama la lambda del clima regresa los detalles del alumno indicado en la solicitud con sus datos y el clima de su ciudad. Eso de autorización lo explicaré en el ejercicio 3.

3. Add authorization to your API Gateway API. Only valid user is admin and password abc123!@#. **30 points.**



Para la resolución de este ejercicio utilicé lo que es BASIC AUTH. Según Cloud Mailin (s.f) es uno de los métodos más básicos y antiguos de autenticación que existen para HTTP. Solo pide usuario y contraseña, algo inseguro si se usa HTTP porque se pueden robar las credenciales si se leen los paquetes. Pero en teoría cumple con lo solicitado, así que así se procederá.

Me basé en el tutorial del mismo autor para hacer este procedimiento, pero hice los equivalentes en CLI.

Comando 1:

```
aws apigateway create-authorizer \  
  --rest-api-id 924l8fs9nf \  
  --name "autorizadorClimaticoDeOmar" \  
  --type REQUEST \  
  --identity-source 'method.request.header.Authorization' \  
  --authorizer-uri "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:292274580527:function:weather_lambda_omar/invocations"
```

El primer paso es crear un autorizador para nuestra API, este requiere la ID de la API, un nombre, la URI de nuestra lambda. Y otras cosas.

La primera de estas otras cosas es el tipo, según documentación del CLI se puede usar request o token, el ultimo es para tokens de autorización, mientras que REQUEST es para cuando llegan request parameters, ese último sonaba más alineado a lo que se hace en esta lambda, por eso lo escogí.

La segunda es el Identity source, el cual es un indicador a la API para decirle "quiero que en los headers del request pongas un header que se llame Authorization y pongas ahí los datos de autenticación dados en el request".

Esto retornará un "recibo" de tipo JSON y ahí vendrá la ID del autorizador. Ahora debemos modificar la API para que utilice el autorizador en sus métodos, o en este caso el único que hay.

Comando 2:

```
aws apigateway update-method \  
  --rest-api-id 924l8fs9nf \  
  --resource-id kyojoy \  
  --http-method GET \  
  --patch-operations      op="replace",path="/authorizationType",value="CUSTOM"  
  op="replace",path="/authorizerId",value="7b8skq"
```

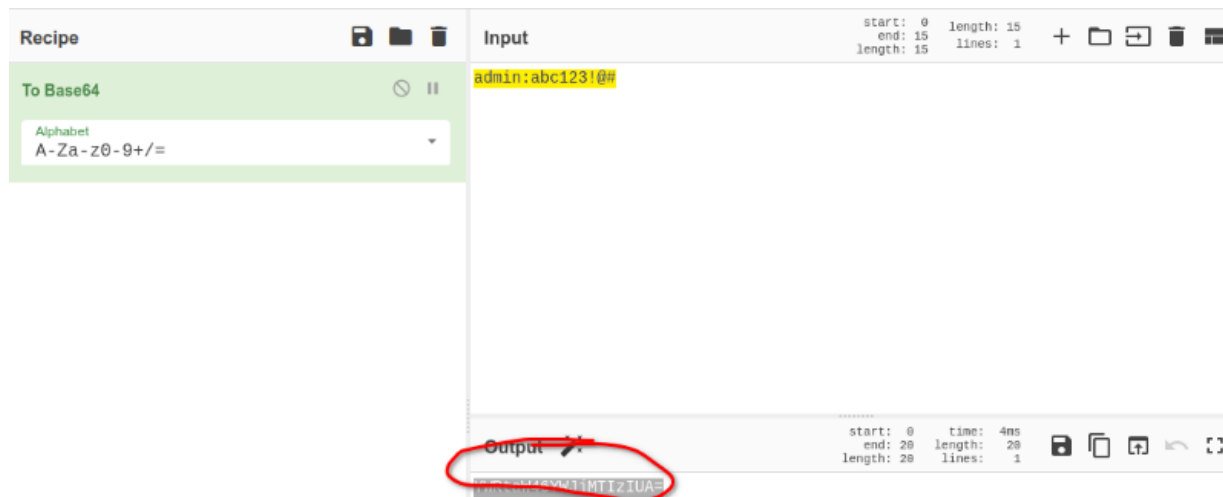
El segundo paso es actualizar los métodos para que use el autorizador. Necesita la ID de la API, ID del recurso ({Id}), el método HTTP, y las operaciones a hacer dado que este comando hace una solicitud de tipo PUT (ediciones parciales).

Nota, al igual que otros comandos con "http-method" se debe repetir este comando por cada método HTTP en nuestra API, con su valor respectivo obviamente (GET, POST, y, PUT).

Esas operaciones que se ven al final del comando deben venir en el mismo parámetro, primero va la operación que en este caso es remplazar un valor, luego el camino, y luego el valor.

Para la primera operación remplazaremos el valor de "authorizationType" a CUSTOM para que use un autorizador customizado, y en la segunda remplazaremos el valor de "authorizerId" para que use nuestro autorizador (hay que darle la ID del autorizador). No se puede cambiar el tipo a CUSTOM sin especificar una ID del autorizador, por eso se deben enviar ambas instrucciones en el mismo comando.

En patch operations, al menos en CLI, para mandar operaciones adicionales, debes escribir usando la misma sintaxis que la primera operación, pero al acabar la primera no debes poner coma, sino iniciar la segunda directamente (en la misma línea claro).



Eso no es el fin de la historia, ahora falta el código de autorización. En la función principal hice una llamada a una función que revisa el header con las credenciales, si son correctas el programa sigue, si no regresa un 401 (no autorizado) con un texto diciendo que no está autorizado.

Ese texto raro es base64, una forma de codificar datos. Ahí la contraseña viene en sintaxis: usuario:contraseña. A la izquierda use un programa en línea para hacer la conversión.

¿Pero de donde salen las credenciales? Pues como deben ser secretas porque es información importante (*sensitive information*) hice un secreto con el comando de crear secretos del servicio de secretos de AWS

Comando 3:

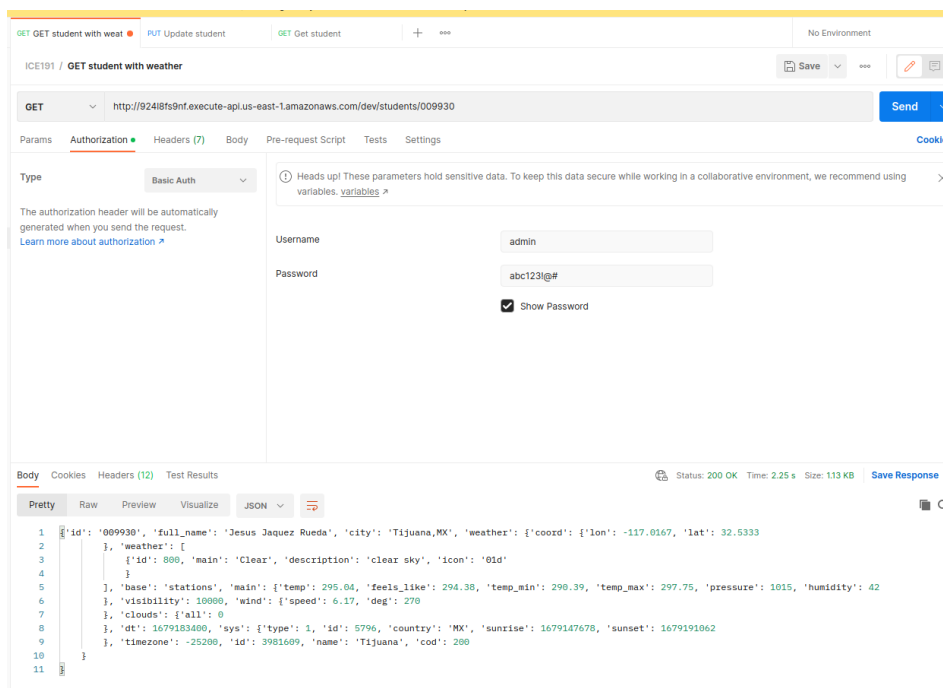
```
aws secretsmanager create-secret --name api_creds_weather_omar --secret-string {StringBase64ConSintaxisMencionada}
```

```
import boto3
import urllib3
from botocore.exceptions import ClientError

def getAPICreds():
    secretsmanager = boto3.client(service_name='secretsmanager')
    #Nombre de secreto
    secret_name = "api_creds_weather_omar"
    #Conseguir el secreto con ese nombre
    secrets_response = secretsmanager.get_secret_value(SecretId=secret_name)
    #Las credenciales estan aqui, en el SecretString
    return secrets_response['SecretString']

def isAuthorized(authHeader):
    #Este metodo es para verificar que el header de autenticación existe
    # Y, que viene con el usuario y contraseña correctos
    superSecretCred = getAPICreds()
    if(superSecretCred in authHeader):
        #Usuario y contraseña en BASIC auth se encodifican en base 64
        # en esta sintaxis: user:password
        # pero en base64, así que reviso que en el header de autorizacion
        # este presente el substring con las credenciales de este ejercicio
        return True
    else:
        return False
```

Así es el fragmento de código encargado de verificar las credenciales, isAuthorized revisa si en el header de autorización está el substring con las credenciales. Y eso lo sabe leyendo del secreto de las credenciales de mi API del clima usando la función encargada de ello, que es getAPICreds.



The screenshot shows a REST client interface with the following details:

- Request:** GET http://9248f8f9nf.execute-api.us-east-1.amazonaws.com/dev/students/009930
- Authorization:** Basic Auth with Username: admin, Password: abc123!@#
- Response:** Status: 200 OK, Time: 2.25 s, Size: 113 KB
- Response Body (JSON):**

```
{
  "id": "009930", "full_name": "Jesus Jaquez Rueda", "city": "Tijuana,MX", "weather": {
    "coord": {
      "lon": -117.0167, "lat": 32.5333
    },
    "weather": [
      {
        "id": 800, "main": "Clear", "description": "clear sky", "icon": "01d"
      }
    ],
    "base": "stations", "main": {
      "temp": 295.04, "feels_like": 294.38, "temp_min": 290.39, "temp_max": 297.75, "pressure": 1015, "humidity": 42
    },
    "visibility": 10000, "wind": {
      "speed": 6.17, "deg": 278
    },
    "clouds": {
      "all": 0
    },
    "dt": 1679183490, "sys": {
      "type": 1, "id": 5796, "country": "MX", "sunrise": 1679147678, "sunset": 1679191862
    },
    "timezone": -25208, "id": 3981609, "name": "Tijuana", "cod": 280
  }
}
```

Autorizado

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

GET student with weatPUT Update studentGET Get student+...

No Environment

ICE191 / GET student with weather

SaveSend

GEThttp://92418fs9nf.execute-api.us-east-1.amazonaws.com/dev/students/009930

ParamsAuthorizationHeaders (7)BodyPre-request ScriptTestsSettings

TypeBasic Auth

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. variables

The authorization header will be automatically generated when you send the request. Learn more about authorization

Usernameadmin

Passwordabc123

Show Password

BodyCookiesHeaders (12)Test Results

Status: 403 ForbiddenTime: 96 msSize: 590 BSave Response

PrettyRawPreviewVisualizeJSON

1No estas autorizado para usar esta API

No Autorizado

Éxito total 😎

4. Read the [Test Driven Development is the best thing that has happened to software design](#) article and write a summary and opinions about it. **10 points.**

Lo siguiente es un copy-paste de mi respuesta de la tarea 5

TDD es una técnica para hacer desarrollo de software. Está basada en la técnica de primero hacer las pruebas del código, y luego hacer el código de la implementación en sí. Es una técnica muy sencilla una vez que se le agarra la onda.

Para la lectura, la prueba *Driven Development* es “lo mejor” que pudo haber pasado en el mundo del desarrollo de software. Consiste en un ciclo de Desarrollo que siempre inicia en hacer las pruebas de software **sin haber escrito una línea de código previamente**. Primero se escribe el código de la prueba, luego se escribe el código que la hace pasar, y luego se hace un arreglo del código para que quede ordenado y mayormente funcional.

Las pruebas por defecto siempre fallarán al inicio, pero eso es bueno, porque al hacer el código que las cumpla, habrás satisfecho todos los requisitos de algún componente de tu aplicación. Esto es lo contrario a hacer las pruebas después, porque si haces eso entonces nunca sabrás cuantas pruebas hacer, en TDD siempre satisface las necesidades de pruebas.

Gracias a esto podemos facilitar el proceso de cómo debemos implementar funcionalidad, al hacer pruebas desde las más chicas hasta las más grandes podemos entender las granularidades del problema, en el caso de las más pequeñas podemos lidiar con pruebas en fechas específicas, donde podemos hacer algo que se le llama como *mock*.

El *mock* es una “simulación” en donde en el código de tu prueba declaras que cuando llames a x función, regrese tal valor. Esto puede ser por varias razones, uno es que siempre quieres probar el mismo comportamiento, otro es que necesitas simular el comportamiento de una API, entre otros similares, nos evita el proceso de andar lidiando con crear escenarios en la vida real y dejárselos al código (de pruebas) al que los haga.

El TDD también nos ayuda a darnos cuenta de que hacemos algo mal, si nos tardamos demasiado en hacer las preparaciones de las pruebas significa que lo que

estamos probando es demasiado grande, y debemos achicarlo en términos de dependencias y responsabilidades que tiene.

Lo último importante que nos dice la lectura es que no hay que irnos con él por si acaso, solo ir por lo necesario, si los requisitos funcionales no lo piden no hay porque implementar pruebas, ya si fuera no funcional es otra historia.

Opino en base a experiencia de casi un año que esto es correcto, el TDD e ha ayudado en mi desarrollo como profesional en el campo de software, desde que inicié mis prácticas profesionales (ya las acabé pero sigo trabajando en el mismo lugar) se me ha puesto esta costumbre de hacer pruebas primero, y nos ha hecho encontrar problemas en nuestro desarrollo que no hubieran pasado si hubiéramos hecho el código antes, es un buen procedimiento que permite que se cumpla lo que se pide en un proyecto sea desde pruebas unitarias, hasta de integración. Es laborioso, eso sí, pero mejor eso a que se rompa algo en producción.

Materiales utilizados:

- [https://www.cloudmailin.com/blog/basic\\_auth\\_with\\_aws\\_lambda](https://www.cloudmailin.com/blog/basic_auth_with_aws_lambda)
  - Guía para usar BASIC AUTH en una API de API Gateway con lambda
- <https://awscli.amazonaws.com/v2/documentation/api/latest/reference/apigateway/index.html#cli-aws-apigateway>
  - Documentación del CLI de API Gateway
- <https://gchq.github.io>
  - Utilidad para hacer conversiones a BASE64 y viceversa

Nota: Si le llegaron alarmas a su correo, sí, puse la contraseña de mi dashboard en mi GitHub por accidente, pero me di cuenta y la cambié, perdón 🙄.