

Assignment 5

ICE191 Software Architecture / Cloud Computing

1. Write a Lambda function to CRUD over the Students DynamoDB table. **30 points.**

La lambda se encuentra en mi repo de Github y en mi sitio web, pero en si misma es casi igual a mi script de mi tarea 3.

El primer paso para hacer una lambda es decidir qué hará la lambda, pero eso ya lo sabemos, el segundo es decidir el ambiente donde correrá, AWS tiene múltiples ambientes de acuerdo, pero yo usaré python3.9 porque me gusta python y en eso hice el script de mi tarea 3.

En fin, el tercer paso es crearla y este es el comando para crear la lambda (Nota, este proceso es asíncrono y no siempre estará lista la lambda al instante de correrlo):

```
aws lambda create-function --function-name DuranOmar-LabdaFunc --runtime  
python3.9 --zip-file fileb://DuranOmarLamb  
da.zip --handler Tarea5.lambda_handler --role  
arn:aws:iam::292274580527:role/lambda_ice191
```

Explicaré los parámetros de acuerdo a la documentación oficial del CLI (s.f):

- Function-name: Nombre de la función, no la del archivo, sino en si el nombre de la lambda es libre a escoger
- Runtime: Ambiente en que correrá, está python, NodeJs, Java, .NET, entre otros
- Zip file: Aquí se envía el archivo de la lambda (código) y dependencias adicionales comprimidos en un zip. En mi caso solo está el archivo de código porque boto3 viene integrado en el servicio usando el runtime de python.
 - Nota: dice fileb porque los archivos zip son binarios, a diferencia de los de texto (tecnicamente tambien lo son, pero aquí se refiere a data binaria que no es solo texto ASCII)

- Handler: La función handler es la primera que correrá la lambda al ser invocada, aquí se debe especificar donde está con la sintaxis *NombreDeArchivoDeCodigo.NombreDeFuncionHandler*
 - Siempre se requiere si se envía el código (y sus dependencias) en ZIP
- Role: El rol de IAM de ejecución de la lambda

Ahora mostraré los cambios en mi código (resto del código en mi Github y sitio web)

```
# AWS lambdas ya viene con boto3 instalado si usas python 3.8 en adelante

import boto3
import botocore

# Lambda handler es la funcion principal que corre mi lambda

def lambda_handler(event, context):
    # Utilizaremos (al igual que en la tarea e) el cliente de DYNAMO DB, este objeto será nuestro medio de comunicaci
    dynamodb = boto3.client('dynamodb')
    # Esto es un copy paste del script de mi tarea 3 pero modificado para funcionar en lambda
    try:
        #Si no nos dió una acción que hacer entonces adios
        option = event["action"]
    except:
        return ("Corre esta lambda denuevo y da un valor C, R, U, o D en la llave 'action' del json de entrada")
    # Haremos un switch case de 3 opciones (más default) una para creación/actualización,
    # otro para borrado, y otro para lecturas
```

Primero que nada, remplace todos los prints por returns, si tuviera logs configurados los hubiera implementado, pero no lo consideré necesario **en esta ocasión** porque la lambda es lo suficientemente simple y esto es un ejercicio académico.

Segundo, event es lo que se recibe cuando la lambda se invoca, más detalles luego, en este caso la acción en vez de ser input del usuario como en mi tarea 3 ahora es parte del json de entrada.

Tercero, el resto del código es exactamente igual menos las diferencias que habia mencionado ¿pero de donde llega el json event? Pues de este comando:

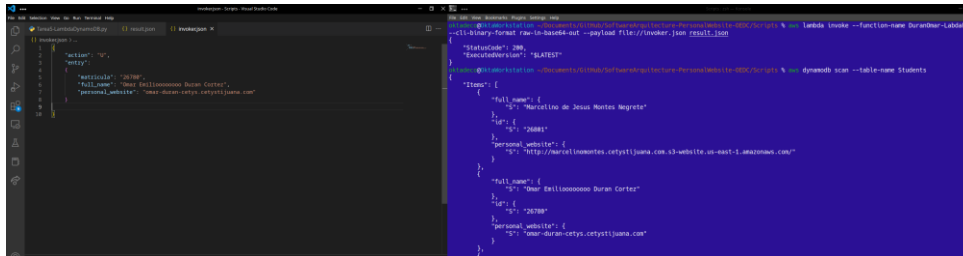
```
aws lambda invoke --function-name DuranOmar-LabdaFunc --cli-binary-format raw-in-
base64-out --payload file://invoke
r.json result.json
```

Explicaré los parámetros según la documentación del CLI:

- Function name: Nombre de la función (debe coincidir con el valor que usaste al crearla)
- Cli Binary Format: Formato del payload: Por defecto AWS según la documentación oficial del CLI para lambda, se usa en este comando base 64 en

vez de las letras normales que usamos los días, debemos escribir este valor para que lo lea nuestro payload de forma correcta

- Payload: Payload para la lambda, en este caso el valor de entrada que era "event" en el código mostrado allá arriba. En este caso usé un archivo a parte
- Result.json: Por defecto AWS quiere que le digas a donde mandar los resultados, desconozco sus motivos y la documentación del CLI no dice porque, solo debemos poner el nombre del archivo del output así nomás.



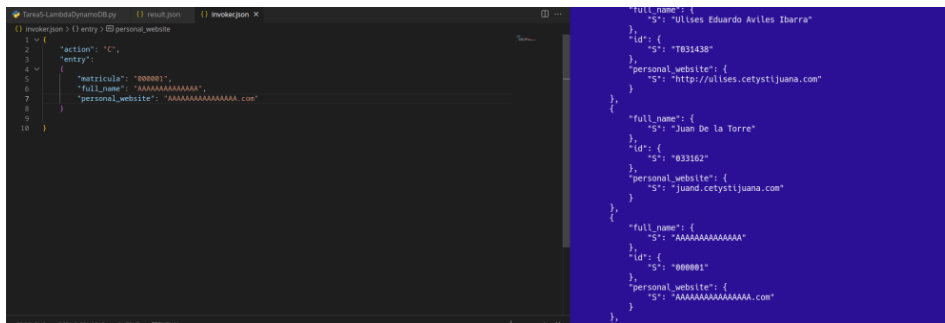
```
aws lambda update-function-code --function-name duranher-LambdaFunction --cli-binary-format raw-in-base64-out --payload file://invoker.json result.json
```

```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "application/json"
  },
  "body": "[]"
}
```

```
aws lambda invoke --function-name duranher-LambdaFunction --payload file://invoker.json --cli-binary-format raw-in-base64-out --result-name Students
```

```
{
  "full_name": {
    "id": {
      "id": "833162"
    },
    "personal_website": {
      "url": "http://marcelinoesquivel.cetystijuana.com.43-website-on-west-1.amazonaws.com"
    }
  },
  "full_name": {
    "id": {
      "id": "833162"
    },
    "personal_website": {
      "url": "http://marcelinoesquivel.cetystijuana.com.43-website-on-west-1.amazonaws.com"
    }
  }
}
```

Corriendo Update, cambié mi segundo nombre a "Emilioooooooooo"



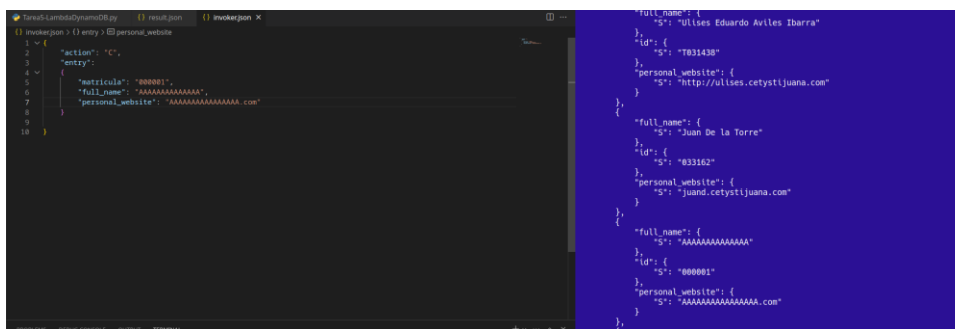
```
aws lambda update-function-code --function-name duranher-LambdaFunction --cli-binary-format raw-in-base64-out --payload file://invoker.json result.json
```

```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "application/json"
  },
  "body": "[]"
}
```

```
aws lambda invoke --function-name duranher-LambdaFunction --payload file://invoker.json --cli-binary-format raw-in-base64-out --result-name Students
```

```
{
  "full_name": {
    "id": {
      "id": "833162"
    },
    "personal_website": {
      "url": "http://marcelinoesquivel.cetystijuana.com.43-website-on-west-1.amazonaws.com"
    }
  },
  "full_name": {
    "id": {
      "id": "833162"
    },
    "personal_website": {
      "url": "http://marcelinoesquivel.cetystijuana.com.43-website-on-west-1.amazonaws.com"
    }
  }
}
```

Corriendo Create



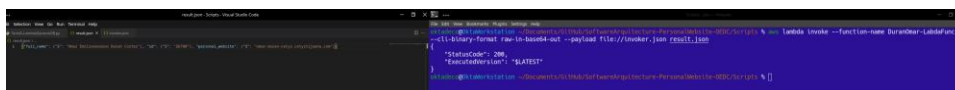
```
aws lambda update-function-code --function-name duranher-LambdaFunction --cli-binary-format raw-in-base64-out --payload file://invoker.json result.json
```

```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "application/json"
  },
  "body": "[]"
}
```

```
aws lambda invoke --function-name duranher-LambdaFunction --payload file://invoker.json --cli-binary-format raw-in-base64-out --result-name Students
```

```
{
  "full_name": {
    "id": {
      "id": "833162"
    },
    "personal_website": {
      "url": "http://marcelinoesquivel.cetystijuana.com.43-website-on-west-1.amazonaws.com"
    }
  },
  "full_name": {
    "id": {
      "id": "833162"
    },
    "personal_website": {
      "url": "http://marcelinoesquivel.cetystijuana.com.43-website-on-west-1.amazonaws.com"
    }
  }
}
```

Corriendo Delete, el nombre "AAAAAA..." Ha desaparecido



```
aws lambda update-function-code --function-name duranher-LambdaFunction --cli-binary-format raw-in-base64-out --payload file://invoker.json result.json
```

```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "application/json"
  },
  "body": "[]"
}
```

```
aws lambda invoke --function-name duranher-LambdaFunction --payload file://invoker.json --cli-binary-format raw-in-base64-out --result-name Students
```

```
{
  "full_name": {
    "id": {
      "id": "833162"
    },
    "personal_website": {
      "url": "http://marcelinoesquivel.cetystijuana.com.43-website-on-west-1.amazonaws.com"
    }
  },
  "full_name": {
    "id": {
      "id": "833162"
    },
    "personal_website": {
      "url": "http://marcelinoesquivel.cetystijuana.com.43-website-on-west-1.amazonaws.com"
    }
  }
}
```

Corriendo Read, ahí estoy yo con mi matrícula.

Mi JSON de entrada aquí tiene dos valores, "action" y "entry", el primero es la letra del CRUS que se hará, Create, Update, Delete, Read. El segundo es la entrada a la DB donde hay tres valores, no siempre se usan todos, es dependiendo de la acción en el CRUD, pero son el nombre completo, id del alumno, y sitio personal, esta entrada es un json dentro del json completo.

2. Write an API Gateway API to CRUD over your Lambda function. **30 points.**

Este fue un proceso altamente complejo, pero iniciemos desde el inicio.

Cabe destacar que me basé en la guía de Sreeprakash Neelakantan (2018) y la guía oficial de la documentación de AWS para crear un gateway API con el CLI

Primer comando:

```
aws apigateway create-rest-api --name 'DuranOmarAPI' --description 'Ahora vas a funcionar?'
```

Este comando es para crear la API, es la raíz de toda mi respuesta, se necesita un nombre y de regreso nos da un "recibo" con una ID de la API

Segundo comando:

```
aws apigateway get-resources --rest-api-id 48iv33fetc
```

Este comando es para obtener la ID del recurso de la raíz, la necesitaremos para crear una ruta en donde poner las solicitudes. Necesita el ID de nuestra API, y nos regresará la ID del recurso padre de la API, o sea -> '/', la ruta raíz.

Tercer comando:

```
aws apigateway create-resource \  
  --rest-api-id 48iv33fetc \  
  --parent-id f40dzb38zf \  
  --path-part {proxy+}
```

Este comando es para crear un recurso en nuestra API, es necesario porque ahí se alojará el endpoint, necesita ID de la API, ID del recurso padre, y el lugar en donde poner el endpoint, el valor de ahí es para crear una API proxy. Una API proxy hará que la lambda según documentación oficial de AWS (sf) tenga acceso al cuerpo, headers y tipo de la solicitud, necesario para cumplir el ejercicio 3 y gran parte de este ejercicio también.

Cuarto comando:

```
aws apigateway put-method \  
  --rest-api-id 48iv33fetc \  
  --resource-id m1t4no \  
  --http-method PUT \  
  --authorization-type "NONE" \  
  --no-api-key-required
```

Este comando le pone el método al endpoint. Necesita ID de la API, ID del recurso (el recurso proxy del comando 3, el método de HTTP, y otros parametros de autorización. Estos ultimos dos son necesarios para hacer que la API sea accesible para **todo** el público, no se necesitan llaves ni headers, así nomás cualquiera puede acceder.

Quinto comando:

```
aws apigateway put-method-response \  
  --rest-api-id 48iv33fetc \  
  --resource-id m1t4no \  
  --http-method PUT \  
  --status-code 200
```

Este comando es para las respuestas del método, esto es una respuesta básica y en realidad la lambda se encarga del trabajo pesado. Los parametros son los mismos que en el previo, menos el código de estado, 200 significa OK.

Sexto comando:

```
aws apigateway put-integration \  
  --rest-api-id 48iv33fetc \  
  --resource-id m1t4no \  
  --http-method PUT \  
  --type AWS_PROXY \  
  --integration-http-method POST \  
  --uri "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:292274580527:function:DuranOmar-LabdaFunc/invocations"
```

Este comando es el más importante, es el que comunica a la API con la lambda, los parametros nuevos aquí son el tipo, el cual en este caso es AWS_PROXY porque quiero que la lambda sepa los parámetros que le llegan, de no ser así solo hubiera usado AWS como valor, en método de integración siempre debe de ser POST para estos escenarios de acuerdo con la guía oficial de AWS (s.f).

En URI (identificador del recurso) va la siguiente sintaxis:

```
arn:aws:nombreDeServicioQueLlamaLaLambda:region:lambda: path/2015-03-31/functions/arn:aws:lambda:region:IDdeUsuario:function:NombreDeLambda/invocations
```

Septimo comando:

```
aws apigateway put-integration-response \  
  --rest-api-id 48iv33fetc \  
  --resource-id m1t4no --http-method PUT \  
  --status-code 200 \  
  --content-handling CONVERT_TO_TEXT
```

Este comando es para procesar las respuestas de la lambda, aquí de parametros nuevos está el Content Handling, que define que hacer con la respuesta, ya sea convertir a texto, o binario, pero queremos texto.

Octavo comando:

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

Le debemos pedir permiso a nuestra lambda para que la usé el gateway API, se hace con esto, necesita el nombre de la función, un Statement que no es nada más que un comentario/nombre de la solicitud de permiso, el tipo de acción el cual es invocar la función, y un Principal que es la forma de decir que servicio de AWS hará una acción sobre la lambda.

Noveno comando:

```
aws apigateway create-deployment --rest-api-id 48iv33fetc --stage-name dev --  
description 'DuranOmar-API-DEV'
```

Esto deployea nuestra API, necesita ID de la API y un nombre para el stage, en verdad puede ser lo que sea, pero le puse dev porque sonaba apropiado. También se puede poner descripción.

Comando 10)

```
aws apigateway test-invoke-method --rest-api-id 48iv33fetc --resource-id m1t4no --http-  
method GET
```

Por defecto no nos dice AWS que sale mal si algo sale mal cuando lo llámanos por fuera, pero si lo hacemos por dentro este comando puede ayudar. Lo que hace es invocar la lambda e imprimir absolutamente todos los detalles de la solicitud incluyendo errores en la lambda. Este comando es "opcional" técnicamente dado que si los primeros nueve los haces bien no necesitas este (pero yo si lo tuve que usar D:)

¿Y el código?

Tuve que hacer cambios en el código para que utilice ahora valores ya sean de parámetros del query o del cuerpo.

```
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"])
        matricula = requestBody["matricula"]
        fullname = requestBody["full_name"]
        personlWebsite = requestBody["personal_website"]

        # 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)
        dynamodb.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': personlWebsite}
            }
        )
        # 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, así esta definida la tabla
        baseResponse["body"] = "Proceso de creación terminado"
    except:
        return baseResponse
```

Ahora el CREATE/UPDATE se ve así, lo distinto es que ahora los valores para la entrada a la tabla la saca del cuerpo (body), y para eso tengo que hacer un json.loads(), sino se rompe, lo otro distinto es que ahora se regresa un tal baseResponse y la respuesta en vez de ser un print o un simple return como en el ejercicio 1 de esta tarea ahora es un valor en una llave de ese objeto.

¿Qué es base response?

Es un objeto JSON que cree yo, pero ese objeto es necesario para las APIs proxy para que funcionen de forma correcta (es un diccionario en términos de python)

```
def lambda_handler(event, context):
    baseResponse = {
        "isBase64Encoded": False,
        "statusCode": 200,
        "headers": { "uuuh": "ya vamonos no?!" },
        "body": "..."
    }
    # Utilizaremos (al igual que en la tarea e) el cliente de DYNAMO DB, este objeto será nuestro medio de comunicación con dynamo
    dynamodb = boto3.client('dynamodb')
    # Esto es un copy paste del script de mi tarea 3 pero modificado para funcionar en lambda
    try:
        # Si no nos dió una acción que hacer entonces adios
        option = event['httpMethod']
    except:
        baseResponse["statusCode"] = 500
        baseResponse["body"] = "Corre esta lambda denuuevo y da un METODO correcto de corrida"
        return baseResponse
    # Haremos un switch case de 3 opciones (más default) una para creación/actualización,
    # otro para borrado, y otro para lecturas

    #C -> Create
    #R -> Delete
    #U -> Update
    #D -> Delete
```

Cada respuesta en una API de tipo PROXY requiere un json con esos cuatro valores, uno es para decir si está encodificado en base64 (lo cual no es cierto en este caso), el

codigo de respuesta, headers, y un cuerpo. Cada función de mi lambda hace modificaciones dependiendo de la situación, pero todas están basadas en el script original de la tarea 3 aun.

```
elif(option=='DELETE'):
    try:
        # Solo necesitamos la llave primaria para borrar
        matricula = event["queryStringParameters"]["matricula"]

        # Para borrar usamos DELETE_ITEM
        dynamodb.delete_item(
            TableName="Students",
            Key={
                "id": {'S': matricula}
            },
            ConditionExpression="attribute_exists(id)"
            # Creamos una condición de expresión, "attribute_exists(N)"
            # es para definir si la operación fue un éxito si se cumplió esa función
            # más detalles en documento
        )
        baseResponse["body"] = "Proceso de borrado terminado"
        return baseResponse

    except botocore.exceptions.ClientError as e:
        # En caso de que no se cumpla la condición (no existe ese item)
        # se retornará un 404 como error
        if e.response["Error"]["Code"] == 'ConditionalCheckFailedException':
            baseResponse["statusCode"] = 404
            baseResponse["body"] = "No existe tal record en la tabla"
            return baseResponse
        except:
            # Error 500 generico si hubo otro tipo de excepción
            baseResponse["statusCode"] = 500
            baseResponse["body"] = "Error inesperado borrando record de la tabla Students, diste matricula?"
            return baseResponse
```

Así se ve mi DELETE ahora

```
elif(option=='GET'):
    try:
        # Solicitar llave primaria (en este caso matricula) para leer el record en la tabla
        matricula = event["queryStringParameters"]["matricula"]

        # Guardar respuesta en variable porque queremos leer información
        # Esto se hace con GET_ITEM
        response = dynamodb.get_item(
            TableName="Students",
            Key={
                "id": {'S': matricula}
            }
        )
        # Verificar si existe la llave Item en la respuesta, este contiene el record en la base de datos
        if ("Item" in response):
            # Imprimirlo tal como está
            baseResponse["body"] = json.dumps(response["Item"])
            return baseResponse
        else:
            # Retornar error 404 si no existe ese record
            baseResponse["statusCode"] = 404
            baseResponse["body"] = "No hay record para ese alumno"
            return baseResponse

    except:
        baseResponse["statusCode"] = 500
        baseResponse["body"] = "Error inesperado intentando leer record de la tabla Students, ¿diste matricula?"
        return baseResponse
```

Así se ve mi GET ahora, hasta incluye el código 404 que pide el ejercicio 3 de esta tarea, en mi condicional del método si no se encuentra una ID en la tabla de estudiantes modificará el valor del código de estado de la respuesta, causando que se regrese un 404 (que significa que no se pudo encontrar lo solicitado).

Todo gracias a la API proxy

Oye, ¿pero esos comandos solo servían para PUT?

Así es, se debe repetir parte del proceso por cada método, siendo los comandos 4, 5, 6, y, 7 para crear los métodos e integraciones para cada método HTTP del CRUD. Y, también se debe repetir el comando 9 para redeployear la API.

¿Cómo actualizo mi lambda para que funcione todo este rollo?

Es necesario actualizar el código de la lambda, porque si no se hace se usará el código viejo del ejercicio 1, el cual no fue hecho pensando en que se usaría un gateway API, pero este código nuevo sí (código completo en mi sitio y Github)

1. Vuelve a crear un ZIP con el código fuente de tu lambda
2. Corre este comando para actualizar el código:
 - 2.1. `aws lambda update-function-code --function-name Dura
nOmar-LabdaFunc --zip-file fileb://DuranOmarLambda.zip`
 - 2.2. A diferencia del comando de creación, el de actualización solo necesita nombre de la función y el archivo zip con código y dependencias
3. Implement 404 HTTP response when an invalid id is passed to the Read in your API Gateway API/Lambda. **30 points.**

Respuesta en pregunta #2

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

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

lidar 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.

Recursos utilizados:

- <https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html>
 - Runtimes de AWS lambda
- <https://docs.aws.amazon.com/lambda/latest/dg/python-package.html>
 - Ejemplo de proceso de creación de lambda con python
- <https://docs.aws.amazon.com/lambda/latest/dg/python-handler.html>
 - Documentación del handler de python de AWS

- <https://awscli.amazonaws.com/v2/documentation/api/latest/reference/lambda/index.html>
 - Documentación del CLI de lambda de AWS
- <https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-create-api-as-simple-proxy-for-lambda.html>
 - Tutorial oficial de AWS para crear un gateway API de tipo proxy
- <https://medium.com/@schogini/aws-apigateway-and-lambda-a-get-mehod-cli-demo-8a05e82df275>
 - Guía de Sreeprakash Neelakantan (2018) para hacer un gateway API y implementarlo en lambda
- <https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-lambda-proxy-integrations.html#set-up-lambda-proxy-integration-using-cli>
 - Guía oficial de AWS para crear un gateway API tipo proxy con CLI
- <https://awscli.amazonaws.com/v2/documentation/api/latest/reference/apigateway/index.html>
 - Documentación oficial del CLI de AWS de API Gateway