Assigment 5

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

A CRUD API stands for **C**reate, **R**ead, **U**pdate, and **D**elete, which is used on persistent storage systems that help keep their data even after there is a shutdown.

The **create** is to add a new database record, the **read** is to retrieve the existing records, the **update** will change the existing record and the **delete** removes records from the database.

This form of API (unlike the REST API that uses HTTP protocol) uses whatever protocol the database has enabled, an example of what could use a CRUD API is a travel agency website:

- The application **creates** a reservation record.
- **Reads** available hotel and room information.
- **Updates** the list of available rooms when a reservation is confirmed.
- If the user **deletes** the entire reservation record it cancels the request.

To have the CRUD implementation we need to create a **Lambda Function**, this is used to run code without you having to provision or manage the servers, this function can be triggered in response to API events or other events in an AWS account.

DynamoDB methods

Steps:

First, we need to have DynamoDB Table already created, in this case, will be using the STUDENTS table.

We need to have an **execution role** that will give our function the right permission to access the AWS resources, my execution role is `arn:aws:iam::292274580527:role/lambda_ice191`.

Now we need to create the Python functions that will be used by CRUD, we will need to import `Boto3` like the last time to use it with the DynamoDB and the official page of AWS recommends using an Error exception, for that will use `ClientError`. We need to call our DynamoDB and the table Students which will be used.

```python
import json
import boto3
from botocore.exceptions import ClientError

dynamoDb = boto3.resource("dynamodb")
table = dynamoDb.Table("Students")
```

After that, we can start creating the CRUD functions, we'll start with the **UPDATE** function, which will use **update_item** to edit the data of an existing record inside the table, we will need:

```python
#Update Item from table
def update_item(key: dict, expression: str, expression_attr: dict):
    table.update_item(Key=key,
                      UpdateExpression=expression,
                      ExpressionAttributeValues=expression_attr
                      )

    response = table.get_item(Key=key)
    return response
```

- A `key` (dictionary) that has the key that will be searched for, in other words, the key of the item that you want to update.
- `UpdateExpression` (string) to indicate what you want to modify (indicating the attributes that you want to modify and the values that you want to assign to them).
- `ExpressionAttributeValues` (dictionary <key: value>) is used in UpdateExpression to indicate the key being used and the value as the new value the record's attribute will take.

Finally, we return an answer with `get_item()` to return the attributes for the item with the given primary key.

Now we do the **INSERT** item into the table:

```python
#Insert Item to table
def insert_item(item: dict):
    table.put_item(Item=item)
    key = item.get("id")
    key = {"id": key}
    response = table.get_item(Key=key)
    return response
```

It will get an item which will be a dictionary with all the attributes and their values to be inserted., once that is received it can use `put_item()` that creates a new item, or replaces an old item with a new item, then we just return an answer with the change made with `get_item()` .

Next, we process with the **GET** function to read something from the table:

```python
#Get Item from table
def get_item(key: dict = None):
    if not key:
        response = table.scan()
        data = response['Items']

        while 'LastEvaluatedKey' in response:
            response = table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])
            data.extend(response['Items'])
        return data

    else:
        response = table.get_item(Key=key)
        return response
```

Here we can make use of the previously explained `put_item()` method, which takes a key dictionary, and if the key is not found it should go through the whole table and respond with a `scan()` method to return one or more items and item attributes by accessing every item in a table.

Then we use the **DELETE** function:

```python
#Delete Item from table
def delete_item(key: dict):
    response = table.delete_item(Key=key)
    return response
```

Here we specify a key and then returned an answer that will contain the table with the found element by their key argument, the `delete_item()` method deletes a single item in a table by its primary key.

Now it is time for our handler to work, this will tell our code what function we want to perform, depending on the CRUD option that we choose:

```python
# Handler function
def handler(event, context):

    operation = event.get("operation")

    if operation == "CREATE":
        try:
            item = event.get("item")
            if "id" and "full_name" and "personal_website" in item:
                dynamo_resp = insert_item(item=item)
                return {"body": json.dumps(dynamo_resp)}
            else:
                return {"body": json.dumps({"message ": "invalid item format"})}
        except ClientError:
            return {"message": json.dumps({"CREATE not successful"})}

    elif operation == "READ":
        try:
            key = event.get("key")
            dynamo_resp = get_item(key=key)
            if "Item" not in dynamo_resp:
                dynamo_resp = {"isBase64Encoded": False,
                               "statusCode": 404,
                               "headers": {"Content-Type": "application/json"},
                               "body": "Item not found"
                               }
                return {"body": json.dumps(dynamo_resp)}
            return json.dumps(dynamo_resp)
        except ClientError:
            return {"body": json.dumps({"READ not successful"})}
```

```python
    elif operation == "UPDATE":
        try:
            key = event.get("key")
            expression = event.get("expression")
            expression_attr = event.get("expression_attr")
            dynamo_resp = update_item(key=key, expression=expression, expression_attr=expression_attr)
            return {"body": json.dumps(dynamo_resp)}
        except ClientError:
            return {"body": json.dumps({"UPDATE not successful"})}
    else:
        return {"body": json.dumps({"Not a CRUD operation"})}
```

Here we use the `routeKey` to know what type of operation will be done (the route key for the route. is for HTTP APIs, the route key can be either `$default` or a combination of an HTTP method and resource path, for example, `GET /pets`). Now we should indicate if we want to CREATE, READ, UPDATE, or DELETE an item from the database. They all use `json.dumps()` to convert a python object into a serialized JSON object.

If we use the **put item (create)** function we should “input” all the parameters that the table uses otherwise it will say that the entered format is invalid, or if the function has an error it throws the `ClientError exception` (This is a general exception when an error response is provided by an AWS service to your Boto3 client’s request).

If we choose the **get item (not found)** where we use the HTPP 404 response which will be explained in part 3 of this assignment. Or if we choose the normally **get item** it will read the table to find the key argument it was given**.**

The **update time** will simply search the given parameters and update them if they are different from what they already are.

At last, the **deleted item** will use the given id to call the delete function and erase that item from the database table.

If the option entered was not part of any of the CRUD functions then it will return a message explaining that it was an invalid option.

Make the .py a zip

```
zip CRUD_TableGaby.zip CRUD_table_gaby.py
```

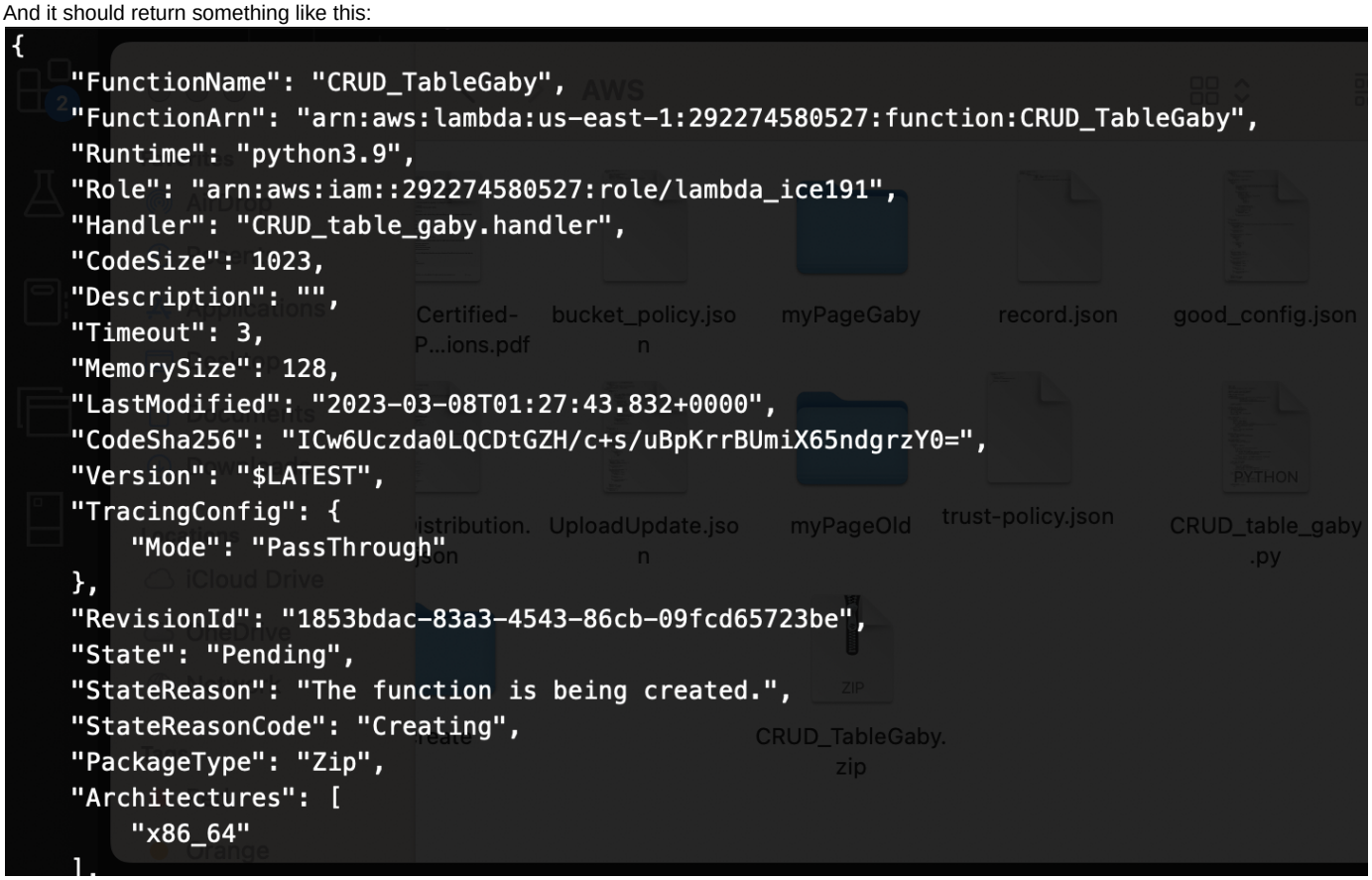
Now we make the file a zip and deploy it using the following command:

```
aws lambda create-function --function-name CRUD_TableGaby \
--zip-file fileb://CRUD_TableGaby.zip \
--handler CRUD_table_gaby.handler \
--runtime python3.9 \
--role arn:aws:iam::292274580527:role/lambda_ice191
```

`aws lambda update-function-code --function-name CRUD_TableGaby --zip-file fileb:///Users/gs0522/Documents/CETYS/AWS/lambda_function/lambda_function.zip`

Here we will create the Lamba function that will take the name of **CRUD_TableGaby,** then we specify from what file will be used, the `handler` the name of the code file, what version we use for python, and the execution role.

And it should return something like this:



To confirm the creation was successful we can use the next code:

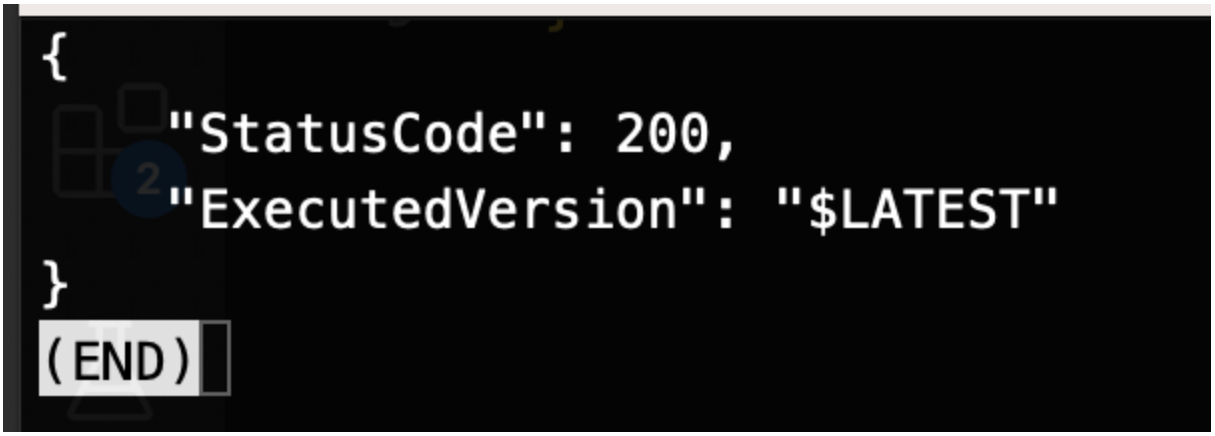
```
aws lambda get-function --function-name CRUD_TableGaby
```

Here we can see the name is correct, this is JSON file, and it has a link location, this link can be copied and pasted into the address bar of the browser to download the zip file of the function code

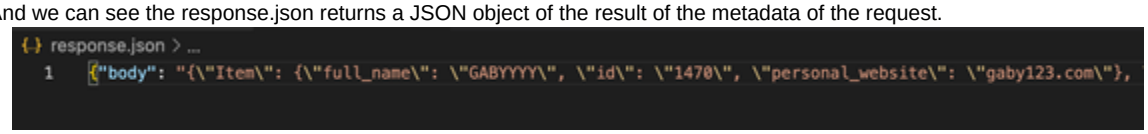
Now we can test our script, we need to have a JSON file with the parameters we want to insert, first the **CREATE**:

```
 aws lambda invoke --function-name CRUD_TableGaby --cli-binary-format
raw-in-base64-out --payload fileb:///Users/gs0522/Documents/CETYS/AWS
/create/\create_example.json response.json
```

```
aws lambda invoke --function-name CRUD_TableGaby --cli-binary-format raw-in-base64-out --payload
fileb:///Users/gs0522/Documents/CETYS/AWS/CRUD_functions/\ create_example.json response.json
```



Here we can see the message 200 which means that it was correctly made, The ExecutedVersion indicates the version of the function that executed. And we can see the response.json returns a JSON object of the result of the metadata of the request.



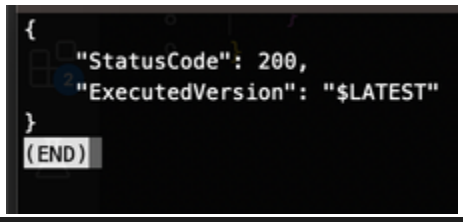
To see the changes we use:

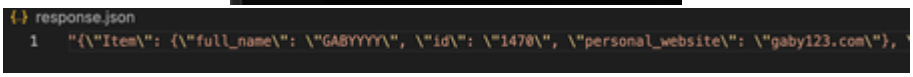
```
aws dynamodb scan --table-name Students
```

Here we can see that the parameters were successfully created in the Students table.

We can continue with the **GET/READ** file:

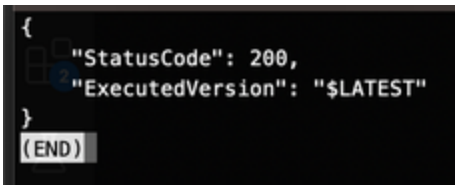

```
aws lambda invoke --function-name CRUD_TableGaby --cli-binary-format
raw-in-base64-out --payload fileb:///Users/gs0522/Documents/CETYS/AWS
/CRUD_functions/get.json response.json
```



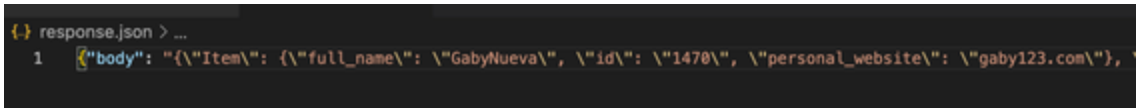


For the **UPDATE** we search with the given ID:


```
aws lambda invoke --function-name CRUD_TableGaby --cli-binary-format
raw-in-base64-out --payload fileb:///Users/gs0522/Documents/CETYS/AWS
/CRUD_functions/update.json response.json
```

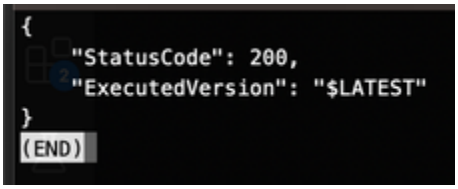


We can see the table changes:



For the **DELETE:**


```
aws lambda invoke --function-name CRUD_TableGaby --cli-binary-format
raw-in-base64-out --payload fileb:///Users/gs0522/Documents/CETYS/AWS
/CRUD_functions/delete.json response.json
```




```
aws dynamodb scan --table-name Students
```

If we search the entry with the previous command, it won't be there.

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

An **API Gateway** is used to create, publish, maintain, monitor, and secure REST, HTTP, and WebSocket APIs at any scale, this will allow us to use GET, POST, PUT, PATCH, and DELETE HTTP methods. In this case, it will use our HTTP protocol to communicate with the created CRUD functions to the Lambda function uploaded.

We need to run the following command:

To create API gateway we run the following command, where we establish a name for the created rest API.

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

Here we get a response:

- **id:** which is a unique identifier
- **name:** the name we chose on the command for the API.
- **createdDate:** the date-time this command was made.
- **apiKeySource:** tell where the API key will be used.

- **endpointConfiguration:** the type of connection we want to use (EDGE = we are using an edge location and not a private one).
- **disableExecuteApiEndpoint**: true/false to see if a user can call the default endpoint.

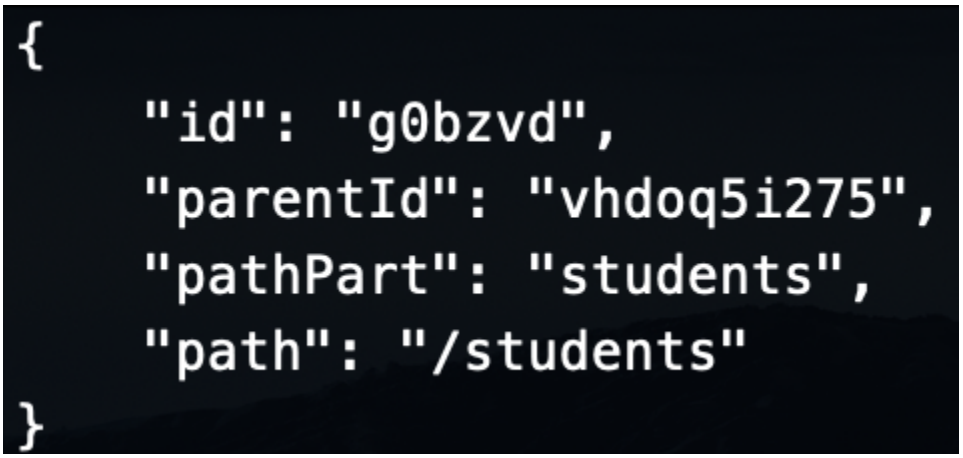
Now we can see if ID of our API endpoint is pointing to any HTTP response, for that we need to use the following command with the ID generated previously:

```
aws apigateway get-resources --rest-api-id [id]
```

```
aws apigateway get-resources --rest-api-id 6tjhrcdzll
```

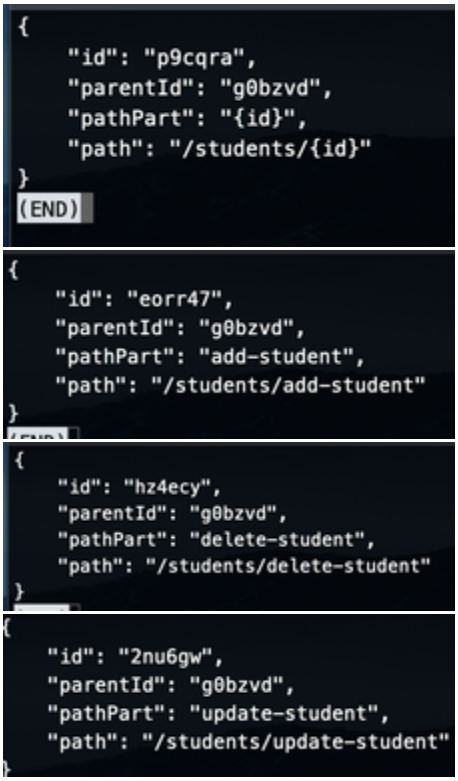
We need to create a new resource that points to our root at the students' table:

```
aws apigateway create-resource --rest-api-id 6tjhrcdzll --region us-
east-1 --parent-id vhdoq5i275 --path-part students
```



Now the path is: /students, now we just have to make the path of each of the HTTP response point to them:

```
aws apigateway create-resource --rest-api-id 6tjhrcdzll --region us-
east-1 --parent-id g0bzvd --path-part "{id}"
aws apigateway create-resource --rest-api-id 6tjhrcdzll --region us-
east-1 --parent-id g0bzvd --path-part add-student
aws apigateway create-resource --rest-api-id 6tjhrcdzll --region us-
east-1 --parent-id g0bzvd --path-part delete-student
aws apigateway create-resource --rest-api-id 6tjhrcdzll --region us-
east-1 --parent-id g0bzvd --path-part update-student
```



- A path to **get** a student by searching their id.
- A path to **create** a new user under the name.
- A path to **delete** a student.
- A path to **update** a student.

Now we need to specify which HTTP method will be using depending on what resource we are at either **POST, GET, PATCH, DELETE.**

```
aws apigateway put-method --rest-api-id 6tjhrcdzll --resource-id eorr47
--http-method GET --authorization-typ "NONE" --region us-east-1 --
request-parameters method.request.path.id=true
```

Implement 404 HTTP response when an invalid id is passed to the Read in your APIGateway API/Lambda. 30 points.

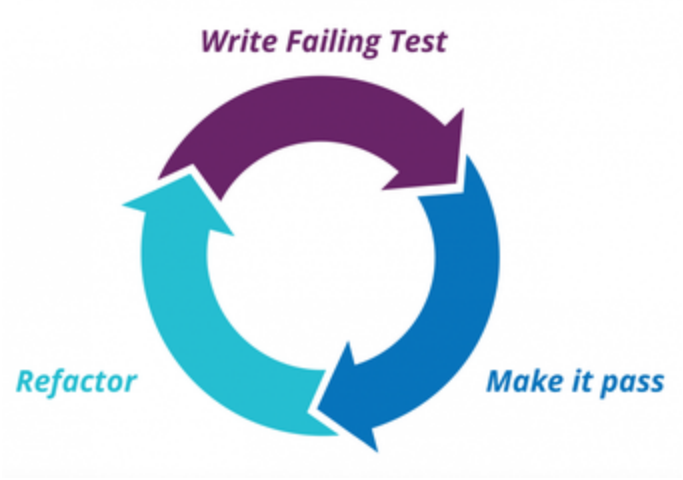
We can add to our function a HTTP **404 not found** response that whenever we try to **READ/GET** a value ID from our DynamoDb that is not existing it throws us that error. In HTTP the first digit indicates a client error, and then the following two digits indicate the specific error encountered.

We can add to our code:

```
elif operation == "READ":
    try:
        key = event.get("key")
        dynamo_resp = get_item(key=key)
        if "Item" not in dynamo_resp:
            dynamo_resp = {"isBase64Encoded": False,
                           "statusCode": 404,
                           "headers": {"Content-Type": "application/json"},
                           "body": "Item not found"
                           }
            return {"body": json.dumps(dynamo_resp)}
        return json.dumps(dynamo_resp)
    except ClientError:
        return {"body": json.dumps({"READ not successful"})}
```

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.

Test Driven Development (TDD) is an iterative approach to software design, when we write software using this approach the final product is unknown; but it provides the ability to have fast feedback about software design



There are usually two ways a project is made first we write the code and then the tests or vice versa:

- **When tests are driven by code:**

Here we first write the code and then we test it, the “hard time” of this method is deciding when to stop writing tests or deciding how much we want it covered.

- **When code is driven by tests:**

Here we define “what we expect from the code”, this makes tests verify the behavior and not the implementation details, also you will always have up-to-date documentation on how it should behave.

When we have tests that use external libraries, variable test data (e.g. LocalDateTime.now() changing every day), a Bloaded setup (too many dependencies), or Mocking Hell (a small change causes the butterfly effect for both the production code and the corresponding tests), we use TDD to help requirements meet implementation.

I personally have never thought of first thinking and writing the tests and then the coding part, I think it does have some benefits such as having the code make specifically what it needs from the requirements and you do not have to “wonder” how many tests are too much.

I think this is something someone has to get used to, otherwise, it may cause some conflict/confusion at first.

This is great since you first write the test that you know the code MUST fulfill, that way you dont overdo; in other words, you build your code to pass the tests, with all the requieremnts already specified.