Assignment 5

1. **Write a Lamba function to CRUD over the Students DynamoDB table.**

Before creating the lambda functions, lets define a few concepts:

Lambda function

Is a service that allows you to run code without provisioning or managing servers. It runs it on a high-availability infrastructure and performs all of the administration of the resources, such as: server is included, operating system maintenance, capacity provisioning, automatic scaling and logging.

Quite popular, since it allows you to focus only on your software of your product and the business aspect of it, instead of managing access controls, operating system, etc.

- There is no need to worry on how you should launch AWS resources or how to manage them. You just need to supply the code in one of the languages that Lambda supports and run it.
    - You organice your code into a lambda function/functions. Lambda runs the code only when needed and scales automatically. Even though there are a considerably large amount of requests per day, you only pay for the compute time consumed, meaning there is no charge when code is not running.

What is CRUD?

CRUD stands for Create, Read, Update and Delete, which are the four basic operations that a model should be able to perform. Used for creating and managing persistent data element, mainly in relational and NoSQL databases.

Software applications that apply CRUD have 3 parts:

1. *An API or server:* contains the code and the functions of each method.

2. *A database:* stores data and help the user retrieve information.
3. *User interface (UI):* helps users interact with the app.

Each operation represent by the letters in CRUD have a corresponding HTTP request method, as the following table shows:

CRUD Operation	HTTP request method
Create	POST
Read	GET
Update	PUT or PATCH
Delete	DELETE

Lets exemplify in a visual way how this request are in a REST environment

Create

We use the POST method to create a new resource of the specified type:

```
POST http://www.mystore.com/items/
Body -
{
  "item":{
    "name": "Tomato sauce",
    "price": 56
}
```

Read

We use the GET method to retrieve information, when an ID is specified:

```
GET http://www.mystore.com/items/1223
Body -
{
  "item":{
    "id": 1223,
    "name": "Tomato sauce",
    "price": 56
  }
}
```

Update

We can use PUT or PATCH, in this case we are using PUT to update information:

```
PUT http://www.mystore.com/items/1223
Body -
{
  "item": {
    "name": "Tomato sauce",
    "price": 61
  }
}
```

Delete

For the delete operation we use DELETE method to remove a resource:

```
DELETE http://www.mystore.com/items/1223
```

Having define this concepts, we can now write the lambda function. We must create the lambda function handler on a python file. When the function is running, lambda executes handler function, which is the method where all events/methods will be processed.

First, we import the needed libraries to access aws services and we initialize the dynamoDB instance. We are also importing an exception provided by aws service, which is *ClientError.* This handles an exception when an error response happens and will be used when an operation wan't done successfully.

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

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

We the create the separate functions for each CRUD operation. To **add/create** an item within a table, we have the following function:

```python
def createItem(item: dict):
    table.put_item(Item=item)
    key = item.get("id")
    key = {"id": key}
    res = table.get_item(Key=key)
    return res
```

- First a new item must be created to return it when we use the *get_item* method. This will helps us confirm that the recent resource was created successfully.
- We use the *put_item* method to add the new item, it accepts “Item” as an attribute, which is what our function create*Item* requires a dictionary (for item) that contains the corresponding values “id”, “full_name” and “personal_website” for the item to be inserted.

To **update** an already existing item, we use the following method:

```python
def updateItem(key: dict, expression: str, expression_attr: dict):
    table.update_item(Key=key,
                      UpdateExpression=expression,
                      ExpressionAttributeValues=expression_attr
                      )

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

- This function takes as arguments the key, that contains the primary key of the item that wants to be updated. It also has the “UpdateExpression” argument that indicates the action that wants to be done and finally the “ExpressionAttributeNames” argument, which is a dictionary that has the expression that wants to be executed (variable name), as well as the new value that the item is going to change to.
- We then return the item that has just been updated with the *get_item* method, by indicating the key of the item.

To **read/get** an item from a table, we have the following method:

```python
def getItem(key: dict = None):
    if not key:
        res = table.scan()
        data = res['Items']

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

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

- This function takes the key argument, which the dictionary that contains the primary key of the item that wants to be read. We indicated as an optional argument, since we could read a specified item or the whole table.
- If the table has an amount of record within the limit and the key was specified, we use the *get_item* method to obtain the wanted item.

To do read the whole table, we use the *scan()* method and as response we get a dictionary containing all items in the table.

- **Important:** We must use pagination because of the 1MB limit of data. We use a loop to iterate according to the “LastEvaluatedKey”. It indicates when the next operation is starting, which for the next call the value is passes as the ExclusiveStartKey parameter.

To **delete** an item from a table, we have the following method:

```python
def deleteItem(key: dict):
    res = table.delete_item(Key=key)
    return res
```

DeleteItem function takes as argument the dictionary with the primary key of the item that wants to be removed from a table, “key” argument. To do so, we use the *delete_item* method and finally return the deleted item, to confirm that is was donde successfully.

Now that we have all 4 functions corresponding to the CRUD operations, we can proceed to write the lambda handler function:

```
def handler_function(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 = createItem(item=item)
                return {"body": json.dumps(dynamo_resp)}
            else:
                return {"body": json.dumps({"message ": "invalid item format"})}
        except ClientError:
            return {"message": json.dumps({"Operation CREATE unsuccessful"})}

    elif operation == "READ":
        try:
            key = event.get("key")
            dynamo_resp = getItem(key=key)
            return {"body": json.dumps(dynamo_resp)}
        except ClientError:
            return {"body": json.dumps({"Operation READ unsuccessful"})}

    elif operation == "DELETE":
        try:
            key = event.get("key")
            dynamo_resp = deleteItem(key=key)
            return {"body": json.dumps(dynamo_resp)}
        except ClientError:
            return {"body": json.dumps({"Operation DELETE unsuccessful"})}

    elif operation == "UPDATE":
        try:
            key = event.get("key")
            expression = event.get("expression")
            expression_attr = event.get("expression_attr")
            dynamo_resp = updateItem(key=key, expression=expression, expression_attr=expression_attr)
            return {"body": json.dumps(dynamo_resp)}
        except ClientError:
            return {"body": json.dumps({"Operation UPDATE unsuccessful"})}
    else:
        return {"body": json.dumps({"Not CRUD operation"})}
```

In the *handler_function* we first obtain the operation that wants to be done on a table by indicating the “method”, we obtain it with the event argument and the *.get* method. We use if-elif-else conditional to determine which operation has to be executed.

- *CREATE:* Function gets the item from the event object and evaluates the item format, meaning it should contain all 3 values: id, full_name and personal_website.

If it does, we call our previously created function for the CREATE operation, which is *createItem().* If not it shows a message indicating that it is an invalid format of the item. We use the exception , to indicate when the request failed and the operation was unsuccessful.

  - We use *json.dump()* method to convert the python objects to json strings, so that is an acceptable language/format for the lambda functions.

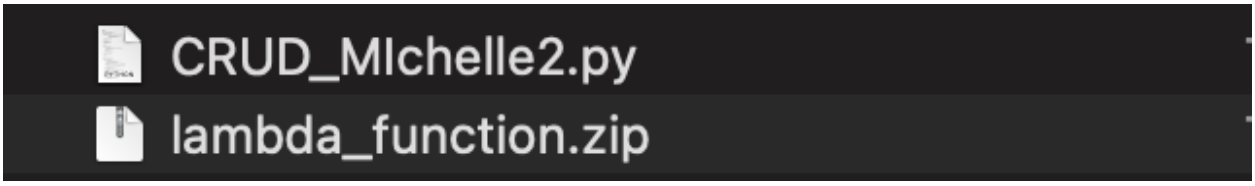
- *READ & DELETE:* Both operations logic is identical, we just use the corresponding function for each action. They get the key from event object as argument and call the *getItem()* function and the *deleteItem()* function accordingly. At the end, we use the same exception as before, to indicate an error has occurred.
- *UPDATE:* Function get 3 values: the key, expression and exp_attribute from the event object and we call the *updateItem* function. We return the item that wants to be updated and we also use the exception as before.

If the indicated operation is not one of the 4 options (CREATE, READ, UPDATE, DELETE) or is not in the payload, we show a message indicating the user that the given operation is not valid.

The command use to create and deploy the lambda function is as the following examples:

```
aws lambda create-function --function-name my-function \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::123456789012:role/lambda-ex
```

- Here we notice that the required file type is .zip, so we must compress our previous python file.



- We can also notice that we must specify a role, which gives our function permission to access AWS resources. The given role is: *arn:aws:iam::292274580527:role/lambda_ice191*

We run the following command:

```
CV9FCYQ4XQ:~ mxm0822$ aws lambda create-function --function-name CRUD_Michelle
 --zip-file fileb:///Users/mxm0822/Documents/School/6tosemestre/CloudComputing/
CRUD_Func_Michelle.zip  --handler CRUD_Michelle.handler  --runtime python3.9  -
-role arn:aws:iam::292274580527:role/lambda_ice191
```

- *function-name:* name of the file that contains our lambda function
- zip-file: file path from the zip generated before from our file with all the functions
- handler: name of our previous python file, instead the extension .handler is used to indicate the method.
- runtime: in this case we are using python 3.9.
- role: given by the teacher, allows us to use AWS resources.

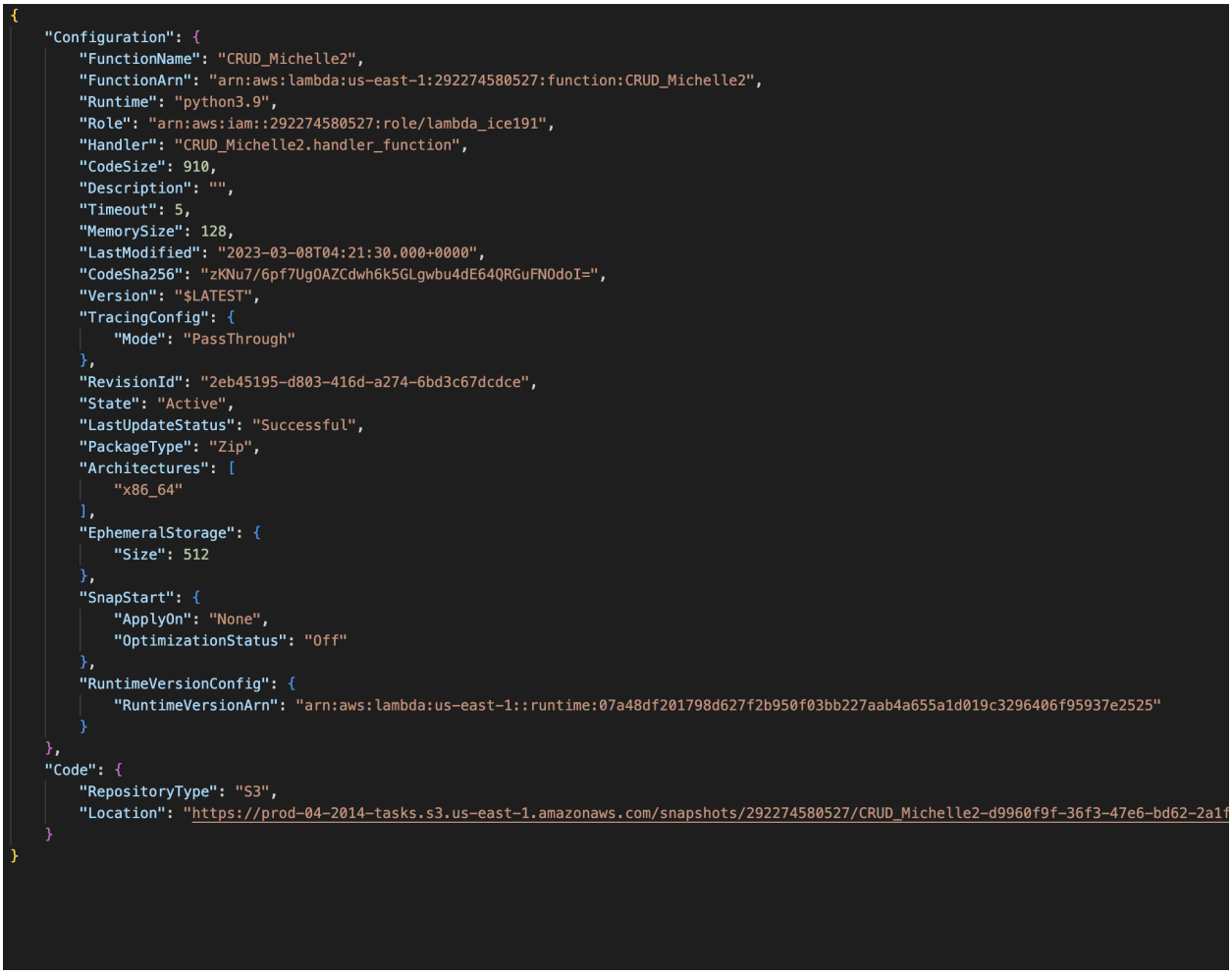
It returns the following json file:


```
{
    "FunctionName": "CRUD_Michelle2",
    "FunctionArn": "arn:aws:lambda:us-east-1:292274580527:function:CRUD_Michelle2",
    "Runtime": "python3.9",
    "Role": "arn:aws:iam::292274580527:role/lambda_ice191",
    "Handler": "CRUD_Michelle2.handler_function",
    "CodeSize": 998,
    "Description": "",
    "Timeout": 3,
    "MemorySize": 128,
    "LastModified": "2023-03-08T01:38:36.000+0000",
    "CodeSha256": "QAeJ3MRymyTPDLgLHJ3wIUhk5ZRqoNpntgh5MX+jQ54=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "fa5fb293-0fc1-4906-aef6-f288b980601d",
    "State": "Active",
    "LastUpdateStatus": "Successful",
    "PackageType": "Zip",
    "Architectures": [
        "x86_64"
    ],
    "EphemeralStorage": {
        "Size": 512
    },
    "SnapStart": {
        "ApplyOn": "None",
        "OptimizationStatus": "Off"
    },
    "RuntimeVersionConfig": {
        "RuntimeVersionArn": "arn:aws:lambda:us-east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525"
    }
}
```


We can use the following command to verify that our function was created successfully:

```
CV9FCYQ4XQ:lambda_function mxm0822$ aws lambda get-function --function-name CRUD
_Michelle2
```

We must obtain an output as the image below shows, which contains the information related to our lambda function. Such information like the role, state of our function and an important one location, which is an url where our zip file can be downloaded from.



To test the **CREATE** functionality we use the following command to invoke our function:

```
CV9FCYQ4XQ:lambda_function mxm0822$ aws lambda invoke --function-name CRUD_Miche
lle2 --cli-binary-format raw-in-base64-out --payload file:///Users/mxm0822/Docum
ents/School/6tosemestre/CloudComputing/test_create.json response.json
```

We use the command *invoke* to call our lambda function. In *function-name* we indicate the name of our file where our function is. The *cli-binary-format* is used to indicate that we are going to pass a raw json string as payload.

- *Payload*: json file that contains all our operations, items and values that will be passed to our lambda function.

At the end we indicate the file where aws cli will write the response from our function, which we named as “response.json”.

The json file used has the content as follows:

```
{
    "operation": "CREATE",
    "item":
    {
        "id": "1313",
        "full_name": "Mimi",
        "personal_website": "pagina_test.com"
    }
}
```

We get the following response. We get the HTTP 200 as StatusCode which indicates us that our request was donde successfully. And the ExecutedVersion indicates us the version of our function that was executed.

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

If we enter our response.json file, we can see that a query of our new created item is shown and more additional data regarding that item.


```
{"body": "{\"Item\": {\"full_name\": \"Mimi\", \"id\": \"1313\", \"personal_website\":
\"pagina_test.com\"}, \"ResponseMetadata\": {\"RequestId\":
\"TF1C61ULHQMCOT05FK9E302G1NVV4KQNSO5AEMVJF66Q9ASUAAJG\", \"HTTPStatusCode\": 200, \"HTTPHeaders\":
{\"server\": \"Server\", \"date\": \"Wed, 08 Mar 2023 02:26:32 GMT\", \"content-type\":
\"application/x-amz-json-1.0\", \"content-length\": \"96\", \"connection\": \"keep-alive\", \"x-amzn-
requestid\": \"TF1C61ULHQMCOT05FK9E302G1NVV4KQNSO5AEMVJF66Q9ASUAAJG\", \"x-amz-crc32\":
\"1047543347\"}, \"RetryAttempts\": 0}}"}
```


The json file used for the read operation was:

```
{
    "operation": "READ",
    "key": {"id": "1313"}
}
```

To test the **READ** functionality we use the following command to invoke our function:

```
[CV9FCYQ4XQ:lambda_function mxm0822$ aws lambda invoke --function-name CRUD_Miche]
lle2 --cli-binary-format raw-in-base64-out --payload file:///Users/mxm0822/Docum
ents/School/6tosemestre/CloudComputing/test_read.json response.json
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

We get the following response. We also get the HTTP 200 as StatusCode which indicates us that our request was donde successfully.

```
{"body": "{\"Item\": {\"full_name\": \"Mimi\", \"id\": \"1313\", \"personal_website\":
\"pagina_test.com\"}, \"ResponseMetadata\": {\"RequestId\":
\"EOFAL5LM9INEBRR27HCETEDJF7VV4KQNSO5AEMVJF66Q9ASUAAJG\", \"HTTPStatusCode\": 200, \"HTTPHeaders\":
{\"server\": \"Server\", \"date\": \"Wed, 08 Mar 2023 04:21:39 GMT\", \"content-type\":
\"application/x-amz-json-1.0\", \"content-length\": \"96\", \"connection\": \"keep-alive\", \"x-amzn-
requestid\": \"EOFAL5LM9INEBRR27HCETEDJF7VV4KQNSO5AEMVJF66Q9ASUAAJG\", \"x-amz-crc32\":
\"1047543347\"}, \"RetryAttempts\": 0}}"}
```

When we enter our response.json file, we can see that the query of the item of the indicated “id” is shown and more additional data regarding that item.

The json file used for the update operation was:

```
{
    "operation": "UPDATE",
    "key": {"id": "1313"},
    "expression": "SET full_name = :val1",
    "expression_attr": {":val1": "Mimi Muñiz"}
}
```

To test the **UPDATE** functionality we use the following command to invoke our function:

```
[CV9FCYQ4XQ:lambda_function mxm0822$ aws lambda invoke --function-name CRUD_Miche]
lle2 --cli-binary-format raw-in-base64-out --payload file:///Users/mxm0822/Docum
ents/School/6tosemestre/CloudComputing/test_update.json response2.json
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

We get the following response. We also get the HTTP 200 as StatusCode which indicates us that our request was donde successfully.


```
{"body": "{\"Item\": {\"full_name\": \"Mimi Mu\\u00f1iz\", \"id\": \"1313\", \"personal_website\":
\"pagina_test.com\"}, \"ResponseMetadata\": {\"RequestId\":
\"IJPSIKIO50O0SIUSTIM7OKCA7NVV4KQNSO5AEMVJF66Q9ASUAAJG\", \"HTTPStatusCode\": 200, \"HTTPHeaders\":
{\"server\": \"Server\", \"date\": \"Wed, 08 Mar 2023 04:32:42 GMT\", \"content-type\":
\"application/x-amz-json-1.0\", \"content-length\": \"103\", \"connection\": \"keep-alive\", \"x-
amzn-requestid\": \"IJPSIKIO50O0SIUSTIM7OKCA7NVV4KQNSO5AEMVJF66Q9ASUAAJG\", \"x-amz-crc32\":
\"1821176515\"}, \"RetryAttempts\": 0}}"}
```


When we enter our response2.json file, we can see that the query of the item that was updated is shown with more additional data regarding that item. Important to notice that the change was created in the *full_name* value, which is now Mimi Muñiz.

The json file used for the delete operation was:

```
{
    "operation": "DELETE",
    "key": {"id": "1313"}
}
```

To test the **DELETE** functionality we use the following command to invoke our function:

```
[CV9FCYQ4XQ:lambda_function mxm0822$ aws lambda invoke --function-name CRUD_Miche]
lle2 --cli-binary-format raw-in-base64-out --payload file:///Users/mxm0822/Docum
ents/School/6tosemestre/CloudComputing/test_delete.json response3.json
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

We also get the HTTP 200 as StatusCode which indicates us that our request was donde successfully.


```
{"body": "{\"ResponseMetadata\": {\"RequestId\":
\"L63M5MQULQJ2HR686FA1S69777VV4KQNSO5AEMVJF66Q9ASUAAJG\", \"HTTPStatusCode\": 200, \"HTTPHeaders\":
{\"server\": \"Server\", \"date\": \"Wed, 08 Mar 2023 04:35:29 GMT\", \"content-type\":
\"application/x-amz-json-1.0\", \"content-length\": \"2\", \"connection\": \"keep-alive\", \"x-amzn-
requestid\": \"L63M5MQULQJ2HR686FA1S69777VV4KQNSO5AEMVJF66Q9ASUAAJG\", \"x-amz-crc32\":
\"2745614147\"}, \"RetryAttempts\": 0}}"}
```


When we enter our response3.json file, we can see that the query of the item that was deleted from the table is shown.

- If we ran *aws dynamodb scan —table-name Students* command, we can notice that the object was deleted and is no longer in the table.

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

API Gateway

Amazon API Gateway is helpful for developers to deliver robust, secure and scalable web and/or mobile applications backends. Allows their applications to connect on a secure way to APIs that run on AWS Lambda, Amazon EC2, etc.

- It sits between a client and a collection of backend services.
- It allows us to create RESTful APIs, such as HTTTP APIs, that crated the real-time communication between two sides.

  HTTP API

  Also known as web API, is a protocol that describes how clients can access resources and indicates what methods with an architecture. Such resources can be: HTML elements, metadata or images.

  - Viewed as a translation guide from one technology to another.
  - HTTP API uses Hypertext Transfer Protocol as the communication protocol. And they expose endpoints as API gateways for some HTTP requests to have access to a server.
  - Comparing to REST APIs, HTTP APIs have better performance and lower costs for serverless APIs.

The command used to create our HTTP API is as follows:

```
aws apigatewayv2 create-api \
--name michelle-http-api \
--protocol-type HTTP \
--target arn:aws:lambda:us-east-1:292274580527:function:CRUD_Michelle2
```

We use the *create-api* to indicate the action we want to do, then we indicate the name that we want our API to be called. We specify the protocol type, which in this case we want HTTP and finally the target, which is lambda function arn. (We can obtain this path from our lambda function configuration json file)

After running this command, we obtain the following response:

With this, we have crated our HTTP API with the integration of our previously created lambda function. Having created our API, we can now proceed to created the different routes.

- With the routes we are going to make allow us to make requests to backend resources. We must have an HTTP method and a resource path.

Since we are integrating out HTTP API and our lambda function, we must pass the integration's id to the routes. Doing this, API can access the lambda function through all routes. If we look into our response generated when we created the API, we can see that it shows our "ApiId", in this case is: *z0k2utbh77*.

We run the following command:

```
[CV9FCYQ4XQ:lambda_function mxm0822$ aws apigatewayv2 get-integrations --api-id z]
 0k2utbh77
```

Our output is:

```
{
    "Items": [
        {
            "ApiGatewayManaged": true,
            "ConnectionType": "INTERNET",
            "IntegrationId": "uwx8sd9",
            "IntegrationMethod": "POST",
            "IntegrationType": "AWS_PROXY",
            "IntegrationUri": "arn:aws:lambda:us-east-1:292274580527:function:CR
UD_Michelle2",
            "PayloadFormatVersion": "2.0",
            "TimeoutInMillis": 30000
        }
    ]
}
```

It is important to notice in the output that a *IntegrationId* was generated (*uwx8sd9*), which will be fundamental to indicate when creating the routes.

Now, to create the route for our READ operation, in HTTP it will be GET. We must indicate our Apis id, the route key where we can access it. Command is as follows:

```
[CV9FCYQ4XQ:lambda_function mxm0822$ aws apigatewayv2 create-route --api-id z0k2u]
tbh77 --route-key 'GET /items' --target integrations/uwx8sd9
{
    "ApiKeyRequired": false,
    "AuthorizationType": "NONE",
    "RouteId": "tjlq4r7",
    "RouteKey": "GET /items",
    "Target": "integrations/uwx8sd9"
}
```

- In the “AuthorizationType” we have NONE, which indicates that is open access.

We can also make a READ operation when we specify the id of the item we want to obtain, if this is the case, command is as follows:

```
[CV9FCYQ4XQ:lambda_function mxm0822$ aws apigatewayv2 create-route --api-id z0k2u
tbh77 --route-key 'GET /items/{id}' --target integrations/uwx8sd9
{
    "ApiKeyRequired": false,
    "AuthorizationType": "NONE",
    "RouteId": "etqn9tu",
    "RouteKey": "GET /items/{id}",
    "Target": "integrations/uwx8sd9"
}
```

To create the route for our CREATE/UPDATE operation, in HTTP it will be PUT. We must indicate our Apis id, the route key where we can access it. We use PUT for both since it either creates a new item or it can also update an existing item with new defined values if the item is found. Command is as follows:

```
[CV9FCYQ4XQ:lambda_function mxm0822$ aws apigatewayv2 create-route --api-id z0k2u
tbh77 --route-key 'PUT /items' --target integrations/uwx8sd9
{
    "ApiKeyRequired": false,
    "AuthorizationType": "NONE",
    "RouteId": "ojwie6u",
    "RouteKey": "PUT /items",
    "Target": "integrations/uwx8sd9"
}
```

For the route of the DELETE operation, in HTTP is the same. We can do it by specifying the id of the item we want to delete. As follows:

```
[CV9FCYQ4XQ:lambda_function mxm0822$ aws apigatewayv2 create-route --api-id z0k2u
tbh77 --route-key 'DELETE /items/{id}' --target integrations/uwx8sd9
{
    "ApiKeyRequired": false,
    "AuthorizationType": "NONE",
    "RouteId": "bwqhqw1",
    "RouteKey": "DELETE /items/{id}",
    "Target": "integrations/uwx8sd9"
}
```

Having all routes created, we must give our API Gateway access to invoke our lambda function. We do this by running the following command:

```
[CV9FCYQ4XQ:lambda_function mxm0822$ aws lambda add-permission --statement-id api
-invoke-lambda --action lambda:InvokeFunction --function-name arn:aws:lambda:us-
east-1:292274580527:function:CRUD_Michelle2 --principal apigateway.amazonaws.com
 --source-arn "arn:aws:execute-api:us-east-1:292274580527:z0k2utbh77/*"
>
```

We obtain de following output:

```
{
    "Statement": "{\"Sid\":\"api-invoke-lambda\",\"Effect\":\"Allow\",\"Principa
l\":{\"Service\":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunctio
n\",\"Resource\":\"arn:aws:lambda:us-east-1:292274580527:function:CRUD_Michelle2
\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:execute-api:us-east-1
:292274580527:z0k2utbh77/*\"}}}"
}
```

Having done this, it is important to update our handler function to be able to call and execute according to the resource paths that where created before when routes where created.

Our modified *handler_function* is now:

```python
def handler_function(event, context):
    method = event.get("routeKey")

    if method == "PUT /items":
        try:
            body = event.get("body")
            body = json.loads(body)
            try:
                item = {"id": body["id"], "full_name": body["full_name"], "personal_website": body["personal_website"]}
                dynamo_resp = insertItem(item=item)
                return json.dumps(dynamo_resp)
            except KeyError:
                return json.dumps("Invalid item format")
        except ClientError:
            return json.dumps("Operation CREATE unsuccessful")

    elif method == "GET /items/{id}":
        try:
            id = event["pathParameters"]["id"]
            key = {"id": id}
            dynamo_resp = getItem(key=key)

            if "Item" not in dynamo_resp:
                dynamo_resp = {"isBase64Encoded": False,
                               "statusCode": 404,
                               "headers": {"Content-Type": "application/json"},
                               "body": "Item with such id not found"
                               }
                return json.dumps(dynamo_resp)

            return json.dumps(dynamo_resp)

        except ClientError:
            return json.dumps("Operation READE unsuccessful")

    elif method == "GET /items":
        try:
            dynamo_resp = getItem()
            return json.dumps(dynamo_resp)

        except ClientError:
            return json.dumps("Operation READE unsuccessfull")

    elif method == "DELETE /items/{id}":
        try:
            id = event["pathParameters"]["id"]
            key = {"id": id}
            dynamo_resp = deleteItem(key=key)
            return json.dumps(dynamo_resp)
        except ClientError:
            return json.dumps("Operation DELETE unsuccessful")

    else:
        return json.dumps("Not a CRUD operation")
```

- **IMPORTANT:** Here it is important to notice, that I already integrated the 404 response. This to not make many updates to my lambda function.

To update our lambda function, we must delete the previous zip file and generated again. And to update de file we must run the following command:

```
[CV9FCYQ4XQ:lambda_function mxm0822$ aws lambda update-function-code --function-n]
ame CRUD_Michelle2 --zip-file fileb:///Users/mxm0822/Documents/School/6tosemestr
e/CloudComputing/lambda_function/lambda_function.zip
```

We can now use CURL to test that our API is working correctly with the changes made in our function.

- **Curl:** it stands for client URL. Is a command line tool that developers use to move data from a server, into and from it. It supports different protocols such as HTTP and HTTPS, and runs almost on every platform.
    - Ideal for testing communication form any device from a local server to most edge devices.

To test **PUT** method we use the following command:

```
CV9FCYQ4XQ:lambda_function mxm0822$ curl -X "PUT" -H "Content-Type: application/
json"  -d  '{"id": "1313", "full_name": "Mimi Muñiz ", "personal_website": "mimi
test.com"}' https://z0k2utbh77.execute-api.us-east-1.amazonaws.com/items
```

In the above command we indicate that we will be using the curl method and in ContentType we indicate that it is an application. Then we introduce and indicate the data we want our item to have and finally we indicate the URL of our API. Our returned response was:


```
{"Item": {"full_name": "Mimi Mu\u00f1iz", "id": "1313", "personal_website": "mim
itest.com"}, "ResponseMetadata": {"RequestId": "2V5PM041IFHS1NQHSHKIK7GJFVVV4KQN
SO5AEMVJF66Q9ASUAAJG", "HTTPStatusCode": 200, "HTTPHeaders": {"server": "Server"
, "date": "Wed, 08 Mar 2023 07:11:44 GMT", "content-type": "application/x-amz-js
on-1.0", "content-length": "100", "connection": "keep-alive", "x-amzn-requestid"
: "2V5PM041IFHS1NQHSHKIK7GJFVVV4KQNSO5AEMVJF66Q9ASUAAJG", "x-amz-crc32": "348473
```


To verify it was created successfully, we can run the command *aws dynamodb scan — table-name Students.* As we can see, it is now shown in the table:

```
    {
        "full_name": {
            "S": "Mimi Muñiz"
        },
        "id": {
            "S": "1313"
        },
        "personal_website": {
            "S": "mimitest.com"
```


To test the **READ** method, we can use a shorter version of the command, where we use our API URL and at the end, specify de id of the item i want to obtain. Like so:

```
[CV9FCYQ4XQ:CloudComputing mxm0822$ curl https://z0k2utbh77.execute-api.us-east-1]
 .amazonaws.com/items/1313
```

And we obtain as response the body and more detailed information of that item:


```
{"Item": {"full_name": "Mimi Mu\u00f1iz", "id": "1313", "personal_website": "mim
itest.com"}, "ResponseMetadata": {"RequestId": "NS17AB1G2MS1PO2EG742RTEM63VV4KQN
SO5AEMVJF66Q9ASUAAJG", "HTTPStatusCode": 200, "HTTPHeaders": {"server": "Server"
, "date": "Wed, 08 Mar 2023 07:19:45 GMT", "content-type": "application/x-amz-js
on-1.0", "content-length": "100", "connection": "keep-alive", "x-amzn-requestid"
: "NS17AB1G2MS1PO2EG742RTEM63VV4KQNSO5AEMVJF66Q9ASUAAJG", "x-amz-crc32": "348473
```

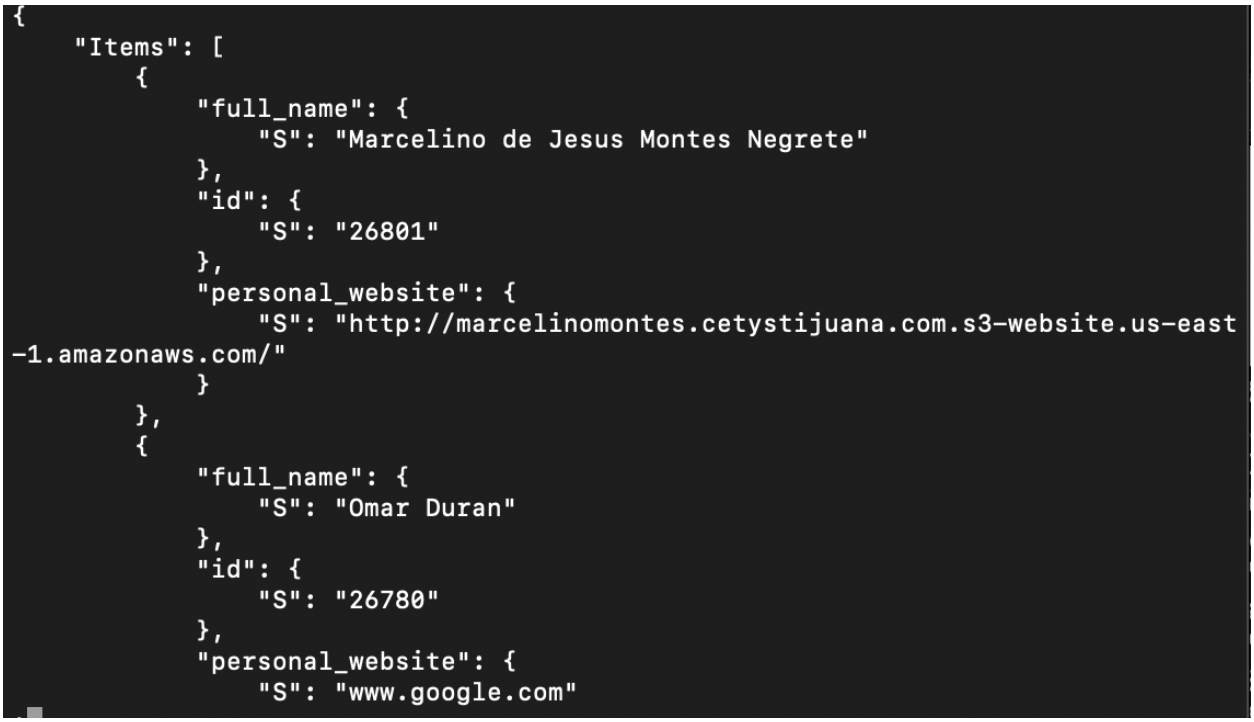

Now to test the **DELETE** method, we can also use the short version of our command, but at the beginning we must specify that we want to execute the DELETE method and the end indicating the id of the item we want to eliminate. Like so:

```
[CV9FCYQ4XQ:CloudComputing mxm0822$ curl -X "DELETE" https://z0k2utbh77.execute-a]
pi.us-east-1.amazonaws.com/items/1313
```

We obtain the following response with the metadata of the eliminated item:


```
{"ResponseMetadata": {"RequestId": "09HQH1N79944TTQEIV15632R6VVV4KQNSO5AEMVJF66Q
9ASUAAJG", "HTTPStatusCode": 200, "HTTPHeaders": {"server": "Server", "date": "W
ed, 08 Mar 2023 07:24:30 GMT", "content-type": "application/x-amz-json-1.0", "co
ntent-length": "2", "connection": "keep-alive", "x-amzn-requestid": "09HQH1N7994
4TTQEIV15632R6VVV4KQNSO5AEMVJF66Q9ASUAAJG", "x-amz-crc32": "2745614147"}, "Retry
```

If we scan the Students table we can see that our item is no longer there:



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

AWS PROXY

It is the wat we integrate our API Gateway with our lambda function. It allows the communication with the created routes. The requests made to our API Gateway URL are passes directly to our lambda function and the response is sent from lambda.

To implement a 4040 HTTP response we must add new integrations to our API, as well as modify and implement this response in put GET method which is in our python file.

We do this by:

```
{
    "Items": [
        {
            "ApiGatewayManaged": true,
            "ConnectionType": "INTERNET",
            "IntegrationId": "uwx8sd9",
            "IntegrationMethod": "POST",
            "IntegrationType": "AWS_PROXY",
            "IntegrationUri": "arn:aws:lambda:us-east-1:292274580527:function:CR
UD_Michelle2",
            "PayloadFormatVersion": "2.0",
            "TimeoutInMillis": 30000
        }
    ]
}
```

It is important to notice that our PayloadFormatVersion is 2.0, which we will have to indicate when updating our code.

Our modified code:

```python
elif method == "GET /items/{id}":
    try:
        id = event["pathParameters"]["id"]
        key = {"id": id}
        dynamo_resp = getItem(key=key)

        if "Item" not in dynamo_resp:
            dynamo_resp = {"isBase64Encoded": False,
                           "statusCode": 404,
                           "headers": {"Content-Type": "application/json"},
                           "body": "Item with such id not found"
                           }
            return json.dumps(dynamo_resp)

        return json.dumps(dynamo_resp)

    except ClientError:
        return json.dumps("Operation READE unsuccessful")
```

In our dynamo response, we just added the statusCode, which in this case is a 404 and the *body* where we show a message indicating that the id given was not found in the table.

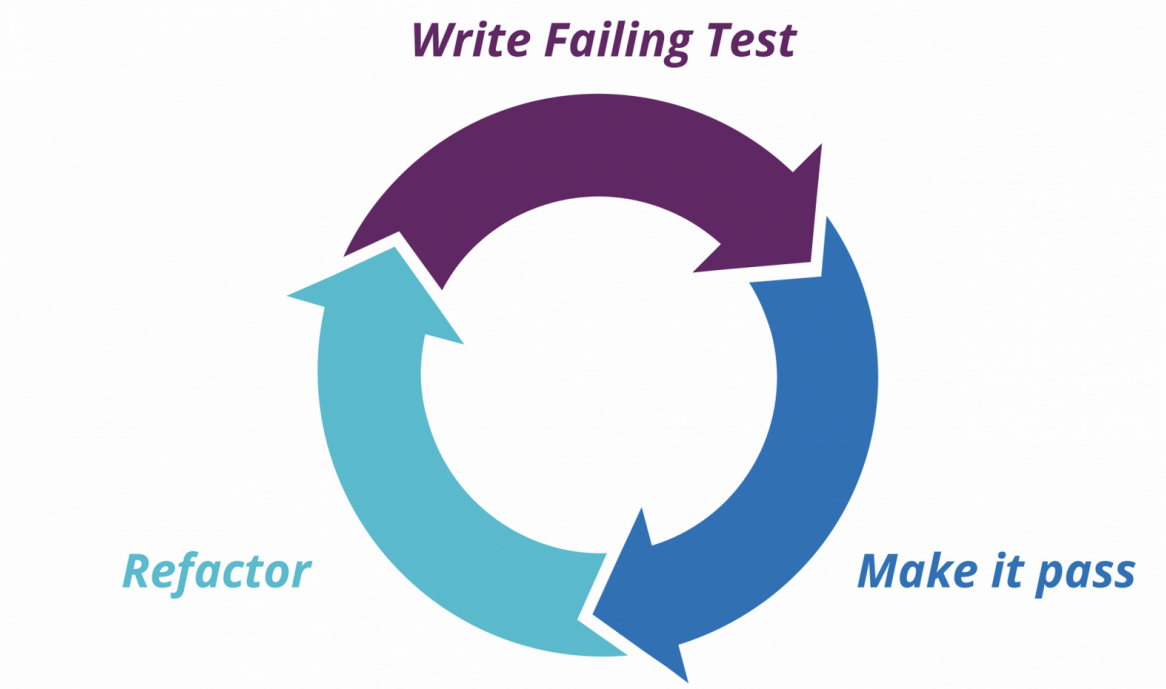
4. **Read the Test Driven Development is the best thing that has happened to software design article and write a summary and opinions about it.**

Summary

This article dives into the importance of applying TDD (Test Driven Development) and some key elements that are beneficial for software design.

TDD Cycles

The iterative approach that TTD applies, supports software to evolve into a working and more elegant solution. This process is different from others, since it doesn't take as much time like normal evolutionary processes.



- *Purple phase:* a failing test is written.
- *Dar blue phase:* minimum code to pass the test created prior
- *Blue phase:* the implementation of the test is cleaned

Difference between test driven by code and code driven by tests

1. When tests are driven by code: Tests are made after code is written. Could be tested method-by-method, success scenarios and failure scenarios. Since it passes

all the tests made and code is running already in production, you could say its functionality is confirmed. The author says that this approach has a lack of implementation and design.

The principles questions that the test-first approach answers are:

- When should I stop adding new tests?
- Am I sure the implementation is finished?

2. When code is driven by tests: There are multiple benefits for this approach. Firstly, tests are verifying behaviors and not implementation details. Second, code is written always with up-to-date documentation. If there is any doubt about a module, a unit or a component, looking through tests a help clarify any questions. Thi*rd, writing tests before actual implementation forces us to ask the question: What do I expect from the code?*
    - It allows you to determine when to stop creating tests.
    - You introduce a new requirement by adding the new red test. Evolution of the implementation is more natural.

TDD on code impossible to test

- When code is written first, there is no determine way to write a proper unit test. For example, when objects are calling external services.
- In reality there is no safe way to verify functionality.
- When code uses dates and we are using static methods such as LocalDateTime.now() complicated the creation of a test and may failed.
    - Test are nor repeatable.
- When tests are too long in the preparation phase and complicated as well.
    - Can either mean there are too many dependencies or single responsibility principle is broken.

Opinion

I definitely think TDD has enormous benefits for software design, especially because it helps requirements meet the wanted and correct implementation. Even though some people don’t really view tests as important, they really are. Tests are the ones that will give fast and more in debt feedback, even before the firs production code line is written.

From personal experience in my work. TDD is a great approach for code that wasn’t thought at the beginning of writing to be able to do unit tests. Not doing this really complicates things when wanting to create unit tests and have the higher code coverage possible. Also, it can really take a long time trying to understand the code and determine what is the approach that we want to take for the testing.