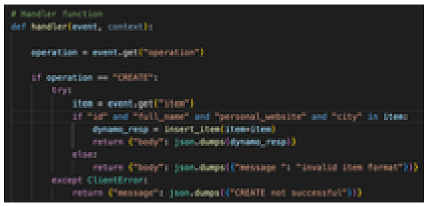
Assigment 6

Modify your Lambda function to include the field City for each new record in theStudents DynamoDB table.

In order to do this, we need a python script that supports the city field about to be created/added, for that, we update our already created lambda function from assignment 5 so it adds this new parameter.



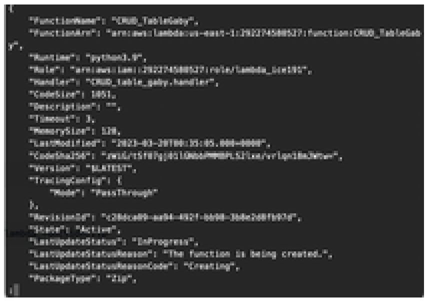
```
# Handler function
def handler(event, context):

    operation = event.get("operation")

    if operation == "CREATE":
        try:
            item = event.get("item")
            if "id" and "full_name" and "personal_website" and "city" 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"})}
```

Make the .py a zip

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

We update the lambda

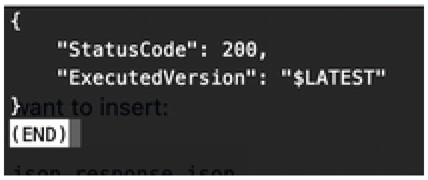
```
aws lambda update-function-code --function-name CRUD_TableGaby --zip-
file fileb:///Users/gs0522/Documents/CETYS/AWS/lambda_function
/lambda_function.zip
```



```
{
    "FunctionName": "CRUD_TableGaby",
    "FunctionArn": "arn:aws:lambda:us-east-1:292274580527:function:CRUD_TableGab
y",
    "Runtime": "python3.9",
    "Role": "arn:aws:iam::292274580527:role/lambda_ice191",
    "Handler": "CRUD_table_gaby.handler",
    "CodeSize": 1051,
    "Description": "",
    "Timeout": 3,
    "MemorySize": 128,
    "LastModified": "2023-03-20T00:35:05.000+0000",
    "CodeSha256": "zW1G/tSf07gj01lGNbbPMMBPLS2lxe/vrlqnl8m3Wtw=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "c28dca09-aa94-492f-bb98-3b8e2d8fb97d",
    "State": "Active",
    "LastUpdateStatus": "InProgress",
    "LastUpdateStatusReason": "The function is being created.",
    "LastUpdateStatusReasonCode": "Creating",
    "PackageType": "Zip",
:
```

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

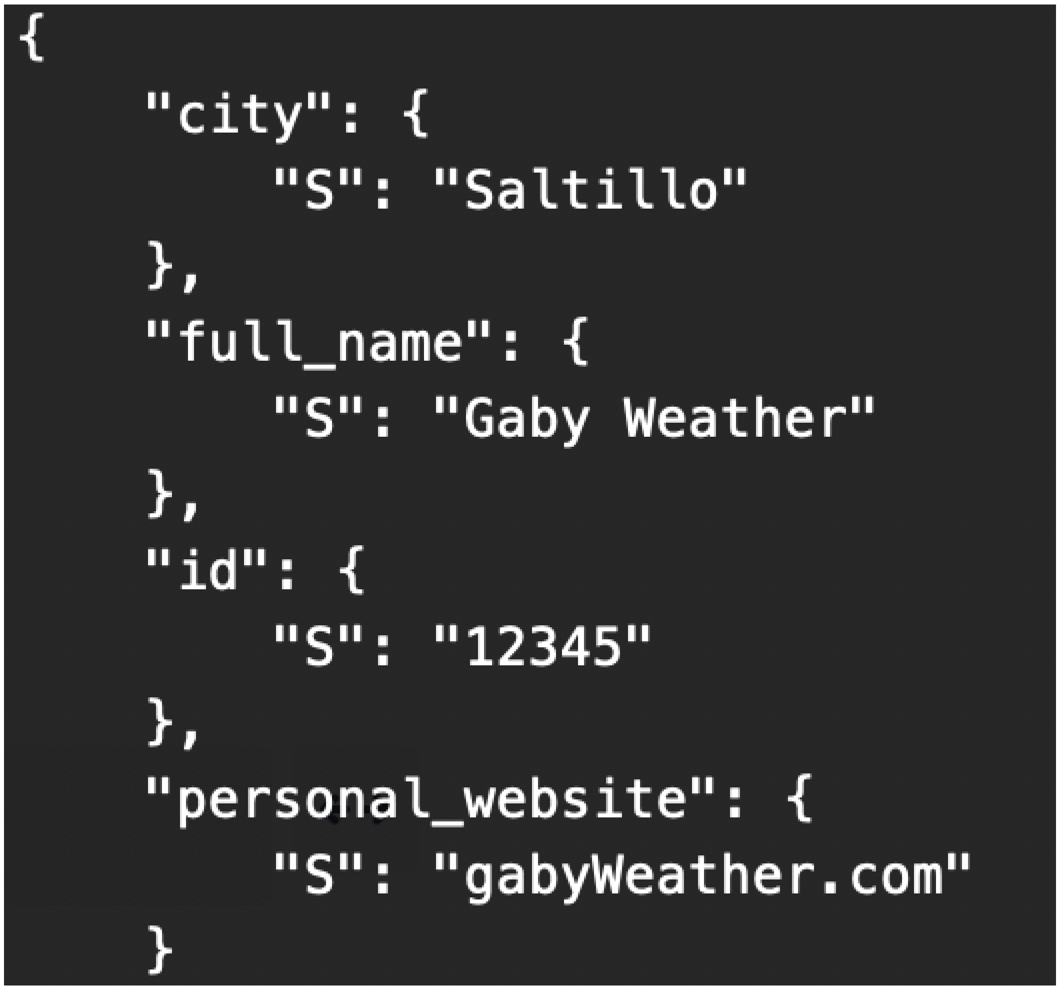
```
aws lambda invoke --function-name CRUD_TableGaby --cli-binary-format
raw-in-base64-out --payload fileb:///Users/gs0522/Documents/CETYS/AWS
/CRUD_functions/\create_example.json response.json
```



```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
(END)
```

To verify that the input was correctly made:

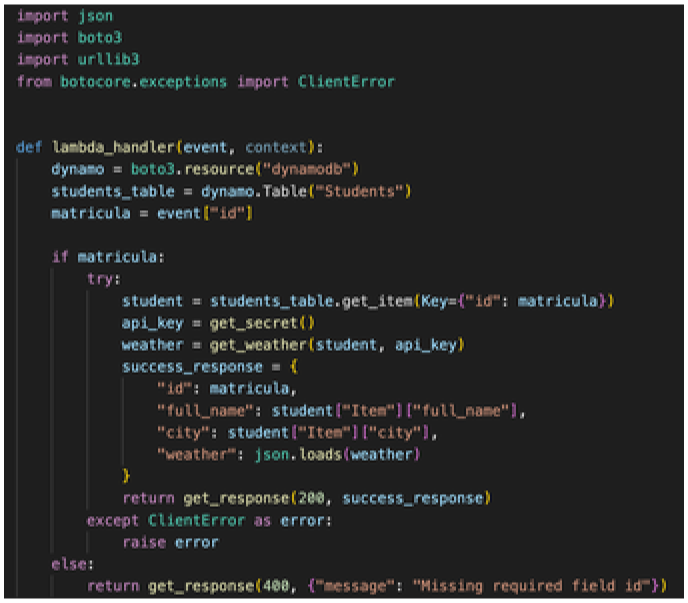
```
aws dynamodb scan --table-name Students
```



```
{
    "city": {
        "S": "Saltillo"
    },
    "full_name": {
        "S": "Gaby Weather"
    },
    "id": {
        "S": "12345"
    },
    "personal_website": {
        "S": "gabyWeather.com"
    }
```

Modify Read in your Lambda function to return the weather of the city assigned to the student's DynamoDB table record. 30 points.

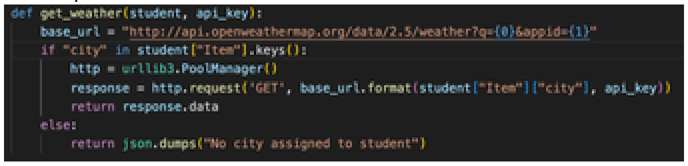
As seen in class this is the main weather code will be using:



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


def lambda_handler(event, context):
    dynamo = boto3.resource("dynamodb")
    students_table = dynamo.Table("Students")
    matricula = event["id"]

    if matricula:
        try:
            student = students_table.get_item(Key={"id": matricula})
            api_key = get_secret()
            weather = get_weather(student, api_key)
            success_response = {
                "id": matricula,
                "full_name": student["Item"]["full_name"],
                "city": student["Item"]["city"],
                "weather": json.loads(weather)
            }
            return get_response(200, success_response)
        except ClientError as error:
            raise error
    else:
        return get_response(400, {"message": "Missing required field id"})
```

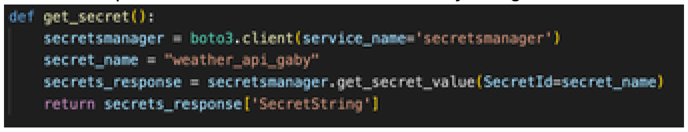
We use a lambda_handler function to coordinate and manage what instructions will be used from the trigger event, it receives two parameters: an event (the request sent by the user) and a context (gives the handler information about what will be executed based on where the function is located).

It receives an id as the event parameter which will be called *matricula*, which will be searched throughout the database table, first, we call a function that will return us the API key (stored as secret); then we have a variable called weather that stores the information given from the student and the api_key, then we have a succes_response object that stores the parameters of that search; if to works we would get a 200 response, otherwise it will be a 400 error response.



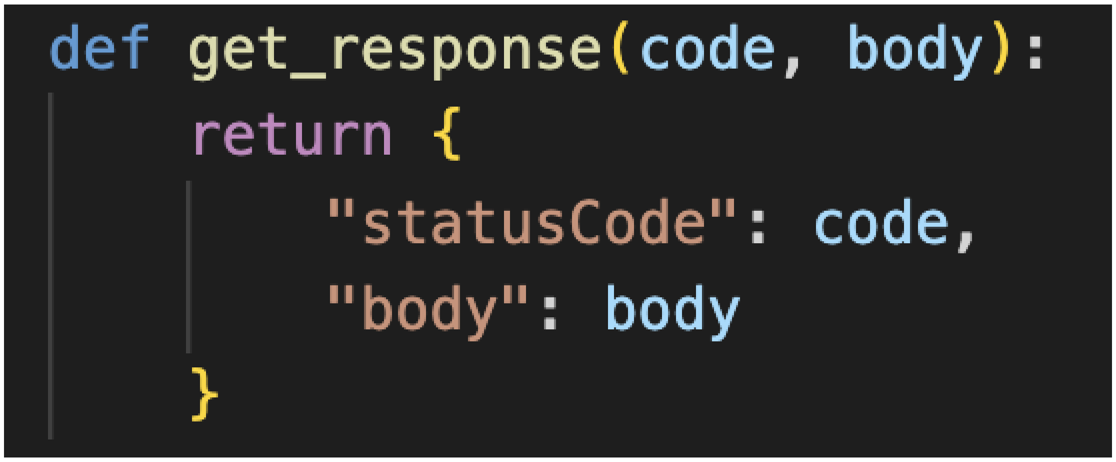
```python
def get_weather(student, api_key):
    base_url = "http://api.openweathermap.org/data/2.5/weather?q={0}&appid={1}"
    if "city" in student["Item"].keys():
        http = urllib3.PoolManager()
        response = http.request('GET', base_url.format(student["Item"]["city"], api_key))
       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

Then we defined a function called **get_weather()** which retrieves the information from the openweathermap.com API and the student's object; if the city exists inside the student's table, we make an HTTP call with **urllib3**, a user-friendly HTTP client for Python into our PoolManager that allows for arbitrary requests while transparently keeping track of necessary connection pools. Then we use the **GET()** method and return the data from that response; if no city was found for that specific student it would return “no city assigned to student”.



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

Here we will use a secrets manager service, in order to retrieve the value of a given secret, in this case, the weather API key; we have a secret_name stored in AWS Secrets Manager, in this case, weather_api_gaby that calls the OpenWeatherMap service. In the third line, we get the value of the secret with the name of the second line.



```python
def get_response(code, body):
    return {
        "statusCode": code,
        "body": body
    }
```

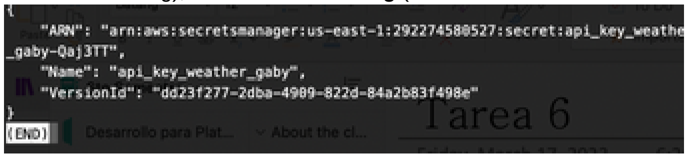
Finally, we have the **get_response()** method that returns the status code (e.g. 200, 400 …) and a body with the information gathered.

Once we have the code we can start running the following commands:

```
aws secretsmanager create-secret --name api_key_weather_gaby --secret-
string f4edb3afca5c9e19aec9f0210b53735b
```

Here we use AWS **secrets manager** (the name of the service that provides encryption to the API key)**create-secret**, (

we choose the name of the secret value we are storing), and the **secret string** (the actual secret that will be encrypted and stored).

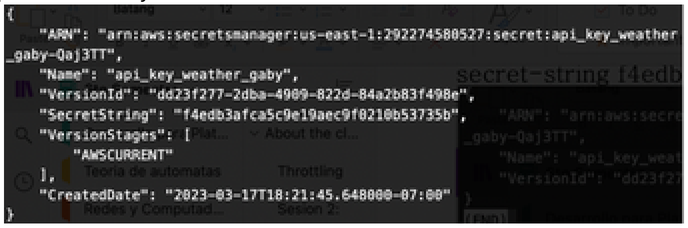


```
{
    "ARN": "arn:aws:secretsmanager:us-east-1:292274580527:secret:api_key_weathe
_gaby-Qaj3TT",
    "Name": "api_key_weather_gaby",
    "VersionId": "dd23f277-2dba-4909-822d-84a2b83f498e"
}
(END)
```

Then we use:

```
aws secretsmanager get-secret-value --secret-id api_key_weather_gaby
```

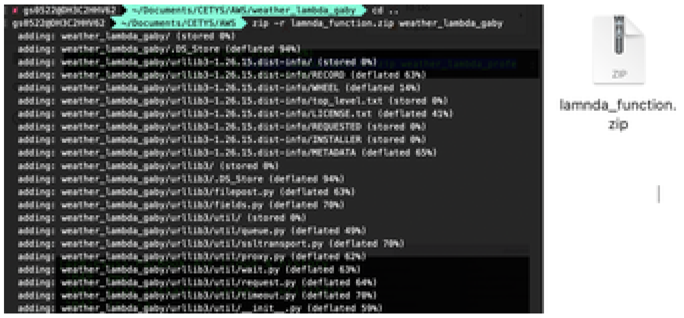
To retrieve the contents of the encrypted fields in my weather API.



```
{
    "ARN": "arn:aws:secretsmanager:us-east-1:292274580527:secret:api_key_weather
_gaby-Qaj3TT",
    "Name": "api_key_weather_gaby",
    "VersionId": "dd23f277-2dba-4909-822d-84a2b83f498e",
    "SecretString": "f4edb3afca5c9e19aec9f0210b53735b",
    "VersionStages": [
        "AWSCURRENT"
    ],
    "CreatedDate": "2023-03-17T18:21:45.648000-07:00"
}
```

Now we have to ZIP the Weather lambda function previously explained:

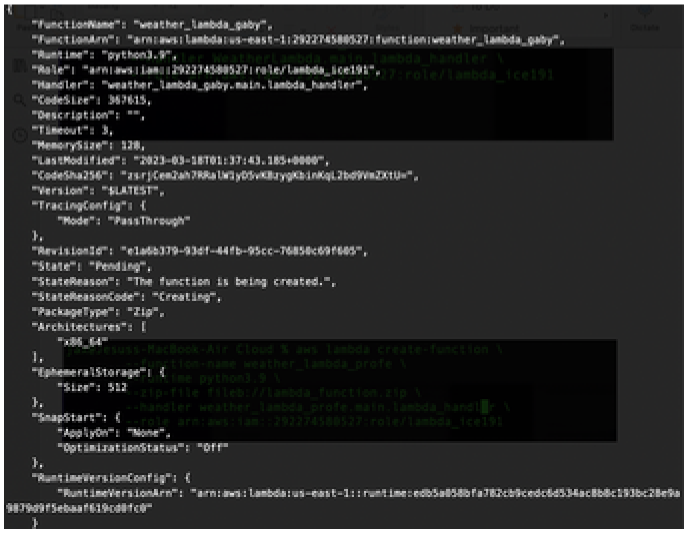
```
Zip -r lambda_function.zip weather_lambda_gaby
```



```
gs0522@DH3C2HHV62 ~/Documents/CETYS/AWS/weather_lambda_gaby cd ..
gs0522@DH3C2HHV62 ~/Documents/CETYS/AWS zip -r lamnda_function.zip weather_lambda_gaby
  adding: weather_lambda_gaby/ (stored 0%)
  adding: weather_lambda_gaby/.DS_Store (deflated 94%)
  adding: weather_lambda_gaby/urllib3-1.26.15.dist-info/ (stored 0%)
  adding: weather_lambda_gaby/urllib3-1.26.15.dist-info/RECORD (deflated 63%)
  adding: weather_lambda_gaby/urllib3-1.26.15.dist-info/WHEEL (deflated 14%)
  adding: weather_lambda_gaby/urllib3-1.26.15.dist-info/top_level.txt (stored 0%)
  adding: weather_lambda_gaby/urllib3-1.26.15.dist-info/LICENSE.txt (deflated 41%)
  adding: weather_lambda_gaby/urllib3-1.26.15.dist-info/REQUESTED (stored 0%)
  adding: weather_lambda_gaby/urllib3-1.26.15.dist-info/INSTALLER (stored 0%)
  adding: weather_lambda_gaby/urllib3-1.26.15.dist-info/METADATA (deflated 65%)
  adding: weather_lambda_gaby/urllib3/ (stored 0%)
  adding: weather_lambda_gaby/urllib3/.DS_Store (deflated 94%)
  adding: weather_lambda_gaby/urllib3/filepost.py (deflated 63%)
  adding: weather_lambda_gaby/urllib3/fields.py (deflated 70%)
  adding: weather_lambda_gaby/urllib3/util/ (stored 0%)
  adding: weather_lambda_gaby/urllib3/util/queue.py (deflated 49%)
  adding: weather_lambda_gaby/urllib3/util/ssltransport.py (deflated 70%)
  adding: weather_lambda_gaby/urllib3/util/proxy.py (deflated 62%)
  adding: weather_lambda_gaby/urllib3/util/wait.py (deflated 63%)
  adding: weather_lambda_gaby/urllib3/util/request.py (deflated 64%)
  adding: weather_lambda_gaby/urllib3/util/timeout.py (deflated 70%)
  adding: weather_lambda_gaby/urllib3/util/__init__.py (deflated 59%)
```

Now we need to upload/create this lambda function inside AWS:

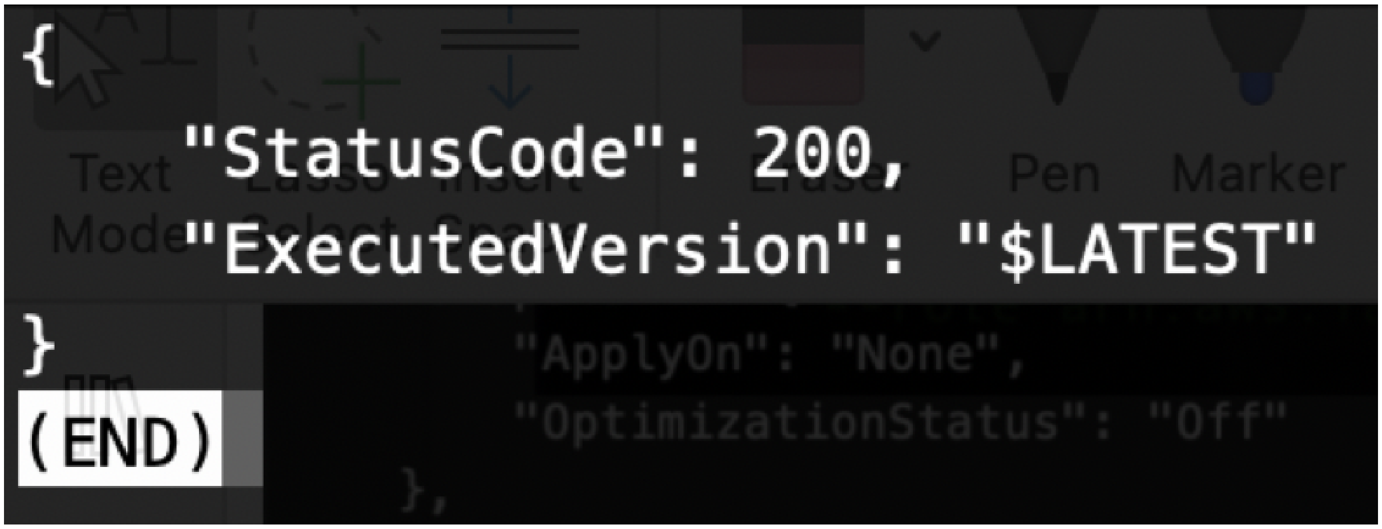
```
aws lambda create-function \
--function-name weather_lambda_gaby \
--runtime python3.9 \
--zip-file fileb://lambda_function.zip \
--handler weather_lambda_gaby.main.lambda_handler \
--role arn:aws:iam::292274580527:role/lambda_ice191
```



```
{
    "FunctionName": "weather_lambda_gaby",
    "FunctionArn": "arn:aws:lambda:us-east-1:292274580527:function:weather_lambda_gaby",
    "Runtime": "python3.9",
    "Role": "arn:aws:iam::292274580527:role/lambda_ice191",
    "Handler": "weather_lambda_gaby.main.lambda_handler",
    "CodeSize": 367615,
    "Description": "",
    "Timeout": 3,
    "MemorySize": 128,
    "LastModified": "2023-03-18T01:37:43.185+0000",
    "CodeSha256": "zsrjCem2ah7RRalW1yD5vKBzygKbinKqL2bd9VmZXtU=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "e1a6b379-93df-44fb-95cc-76850c69f605",
    "State": "Pending",
    "StateReason": "The function is being created.",
    "StateReasonCode": "Creating",
    "PackageType": "Zip",
    "Architectures": [
        "x86_64"
    ],
    "EphemeralStorage": {
        "Size": 512
    },
    "SnapStart": {
        "ApplyOn": "None",
        "OptimizationStatus": "Off"
    },
    "RuntimeVersionConfig": {
        "RuntimeVersionArn": "arn:aws:lambda:us-east-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a
9879d9f5ebaaf619cd0fc0"
    }
```

Now we can invoke our Lambda function to see what response we get inside my response_gaby.json

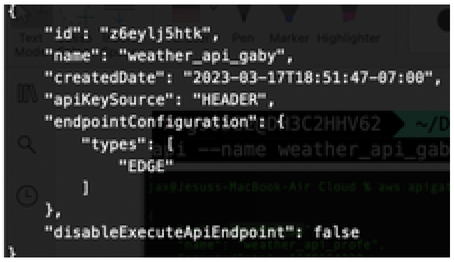
```
aws lambda invoke --function-name weather_lambda_gaby --cli-binary-
format raw-in-base64-out --payload '{"id": "009930"}' response_gaby.json
```



```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
(END)
```

Now we can create the rest-API:

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

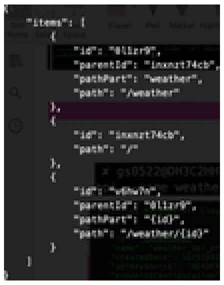


```
{
    "id": "z6eylj5htk",
    "name": "weather_api_gaby",
    "createdDate": "2023-03-17T18:51:47-07:00",
    "apiKeySource": "HEADER",
    "endpointConfiguration": {
        "types": [
            "EDGE"
        ]
    },
    "disableExecuteApiEndpoint": false
}
```

Now we can see what API call

resources were uploaded:

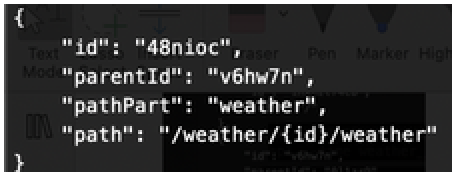
```
aws apigateway get-resources --rest-api-id lhca9t5rsb
```



```
{
    "items": [
        {
            "id": "0l1zr9",
            "parentId": "inxnzt74cb",
            "pathPart": "weather",
            "path": "/weather"
        },
        {
            "id": "inxnzt74cb",
            "path": "/"
        },
        {
            "id": "v6hw7n",
            "parentId": "0l1zr9",
            "pathPart": "{id}",
            "path": "/weather/{id}"
        }
    ]
}
```

Now we create a resource that uses a parent id to associate a child resource under the root, the id of the root resource is specified as the value of the parentId property.

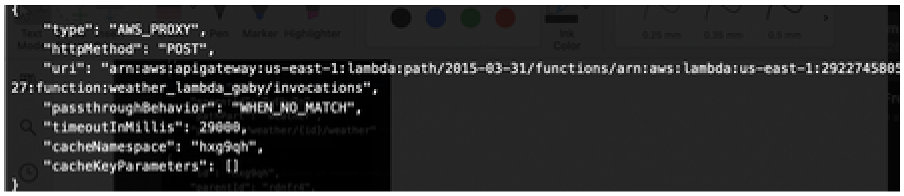
```
aws apigateway create-resource --rest-api-id lhca9t5rsb --parent-id
v6hw7n --path-part weather
```



```
{
    "id": "48nioc",
    "parentId": "v6hw7n",
    "pathPart": "weather",
    "path": "/weather/{id}/weather"
}
```

Now we send an AWS integration request, this is an HTTP request that API Gateway submits to the backend, passing along the client-submitted request data, and transforming the data, if necessary.

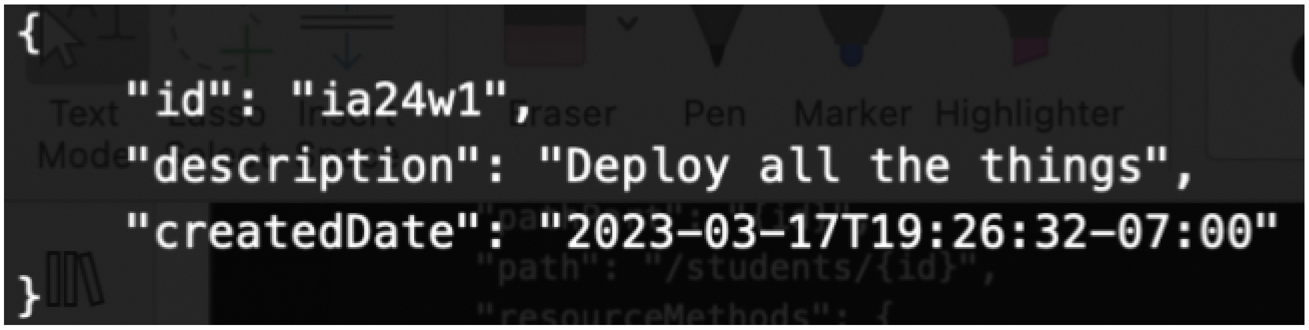
```
aws apigateway put-integration \
> --rest-api-id lhca9t5rsb \
> --resource-id hxg9qh \
> --http-method GET \
> --integration-http-method POST \
> --type AWS_PROXY \
> --uri arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions
/arn:aws:lambda:us-east-1:292274580527:function:weather_lambda_gaby
/invocations
```



```
{
    "type": "AWS_PROXY",
    "httpMethod": "POST",
    "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:29227458 05
27:function:weather_lambda_gaby/invocations",
    "passthroughBehavior": "WHEN_NO_MATCH",
    "timeoutInMillis": 29000,
    "cacheNamespace": "hxg9qh",
    "cacheKeyParameters": []
}
```

We can do a deployment of our Weather API:

```
 aws apigateway create-deployment \
> --rest-api-id lhca9t5rsb \
> --stage-name dev \
> --description 'Deploy all the things'
```



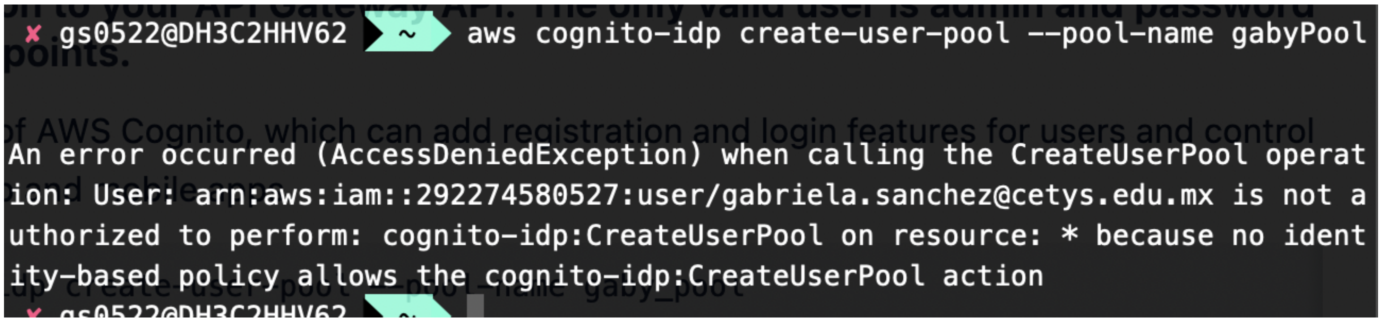
```
{
    "id": "ia24w1",
    "description": "Deploy all the things",
    "createdDate": "2023-03-17T19:26:32-07:00"
}
```

Add authorization to your API Gateway API. The only valid user is admin and password abc123!@#. 30 points.

We can make use of AWS Cognito, which can add registration and login features for users and control access to your web and mobile apps.

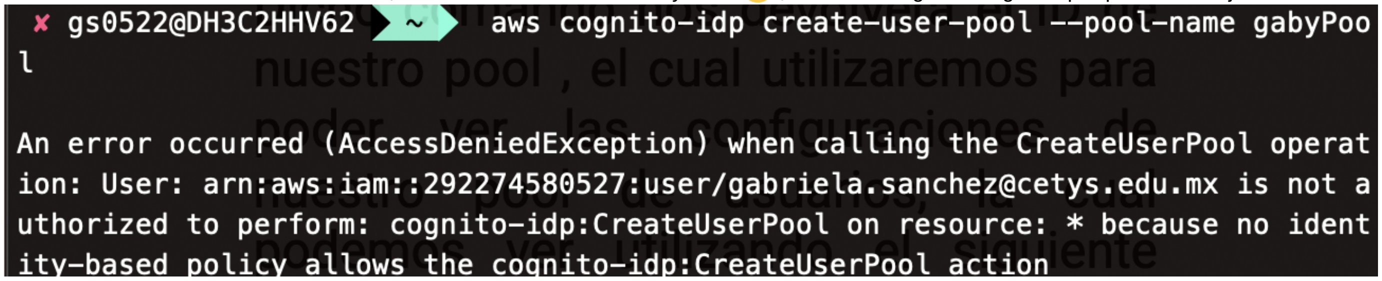
```
aws cognito-idp create-user-pool --pool-name gaby_pool

aws cognito-idp create-user-pool-client --user-pool-id us-east-
1_5R2IBSHXC --client-name GabyPoolCLient --no-generate-secret --
explicit-auth-flows "ALLOW_ADMIN_USER_PASSWORD_AUTH"
"ALLOW_REFRESH_TOKEN_AUTH"
```

```
aws apigateway create-authorizer $API --name "gaby-authorizer" --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_gaby/invocations"
```



```
✘ gs0522@DH3C2HHV62 ~ aws cognito-idp create-user-pool --pool-name gabyPool

An error occurred (AccessDeniedException) when calling the CreateUserPool operat
ion: User: arn:aws:iam::292274580527:user/gabriela.sanchez@cetys.edu.mx is not a
uthorized to perform: cognito-idp:CreateUserPool on resource: * because no ident
ity-based policy allows the cognito-idp:CreateUserPool action
```

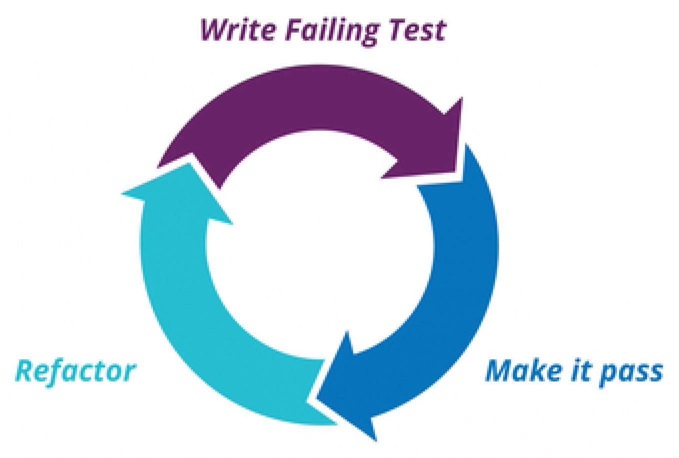
PROFE NO ME DEJA HACERLO AAAA , lo intent con dif commandos y nada 😞 , mañana le sigo investigando porque no me deja



```
✘ gs0522@DH3C2HHV62 ~ aws cognito-idp create-user-pool --pool-name gabyPoo
l

An error occurred (AccessDeniedException) when calling the CreateUserPool operat
ion: User: arn:aws:iam::292274580527:user/gabriela.sanchez@cetys.edu.mx is not a
uthorized to perform: cognito-idp:CreateUserPool on resource: * because no ident
ity-based policy allows the cognito-idp:CreateUserPool action
```

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 don't overdo it; in other words, you build your code to pass the tests, with all the requirements already specified.