4. Lambda refactor

We have good amount of implementation in place now. Before moving to writing unit test cases lets refactor a code little bit so that we can easily write unit test cases which focuses on testing code in small parts.

This is almost similar code before with few additions in functionality. Mainly it focuses on breaking the code ins small functions so that writing unit test cases can be easy and efficient. You will also see addition of doc strings to individual functions.

Please make sure to go through this article to understand ***clean code approach.***

```
import json
import os   # new
import boto3    # new
import uuid     # new
from botocore.exceptions import ClientError     # new


def get_table_name():       # new
    """
    Read and return DynamoDB table name from environment variable

    Returns
    -------
    string
        DynamoDB table name stored in environment variable 'TABLE_NAME'
    """
    ret_val = os.environ["TABLE_NAME"]
    return ret_val

def get_table_region():     # new
    """
    Read and return DynamoDB table region from environment variable

    Returns
    -------
    string
        DynamoDB table region stored in environment variable 'REGION'
    """
    ret_val = os.environ["REGION"]
    return ret_val

def extract_event(event):   # new
    """
    Extracts and validates fields from event object passed by API
endpoint

    This function simply returns extracted values as list with first
item
    as validation success as true or false

    Parameters
    ----------
    event : dict
        event object passed from AWS API gateway endpoint

    Returns
    -------
    list
        extracted values from event with first item validation
        success as true or false
    """
    qs_validation_sucess = True

    user_name = event['queryStringParameters'].get('name')
    if user_name is None:
        qs_validation_sucess = False

    request_method = event['httpMethod']
    request_path= event['path']

    return  [qs_validation_sucess,user_name,request_method,request_path]

def prepare_response(status_code,body): # new
    """
    Generate response object to return from API endpoint

    Parameters
    ----------
    event : dict
        event object passed from AWS API gateway endpoint

    Returns
    -------
    list
        extracted values from event with first item validation
        success as true or false
    """
    response_object = {}
    response_object['statusCode'] = status_code
    response_object['headers'] = {}
    response_object['headers']['Content-Type'] = 'application/json'
    response_object['body'] = body

    return response_object

def lambda_handler(event, context):
    """
    Function handles API endpoint requests

    Parameters
    ----------
    event : dict
```

```
        event object passed from AWS API gateway endpoint
    context : dict
        object passed to lambda by default on invocation

    Returns
    -------
    dict
        lambda execution details
    """
    response_object = {}

    # Extracts required details from event object.
    # Remember structure of event object stored in apigateway-aws-proxy.
json
    qs_validation_sucess,user_name,request_method,request_path =
extract_event(event) #new

    # check for POST operation on user endpoint with query string
validation
    if request_path == "\user" and qs_validation_sucess:    # new
        if request_method == 'POST':
            # call post handler for user endpoint
            post_handler_return = user_post_handler(user_name)
            response_object = prepare_response(   # new
                    post_handler_return['ResponseMetadata']
['HTTPStatusCode'],
                    'user added successfully' if \
                    post_handler_return['ResponseMetadata']
['HTTPStatusCode']==200 \
                    else 'error occured while adding user'
                )
    else:   #new
        response_object = prepare_response(400, 'invalid query
arguments')   # new

    # return from lambda
    return response_object


def user_post_handler(user_name):   # new
    """
    POST request handler for 'user' API endpoint

    Parameters
    ----------
    user_name : string
        name passed in query string during POST operation

    Returns
    -------
    list
        extracted values from event with first item validation
        success as true or false
    """
    table_region = get_table_region()
    table_name= get_table_name()

    # create dynamodb table object
    ddb_resource = boto3.resource("dynamodb", region_name=table_region)
    ddb_table = ddb_resource.Table(table_name)

    # prepare dict to insert into DynamoDB table
    insert_item = {
        'UserID': str(uuid.uuid1()),
        'UserName' : user_name
    }

    try:
        ddb_response = ddb_table.put_item(
            TableName=table_name,
            Item=insert_item
        )
    except ClientError as e:
        raise e
        #TODO better error handling

    # returning responce from dynamoDB put_item operation
    return ddb_response
```

Lets go through individual update made.

Imported several new libraries

os : Reading environment variable for table and table region

boto3 : Perform DynamoDB operations

uuid : Generate unique ID which can be used in ID partition key filed of DynamoDB table

botocore.exceptions : Using ClientError object to catch common errors in boto3 operations. Boto3 classifies all AWS service errors and exceptions as `ClientError` exceptions. When attempting to catch AWS service exceptions, one way is to catch `ClientError` and then parse the error response for the AWS service-specific exception. For more information please refer Boto3 Error handling

```
import os
import boto3
import uuid
from botocore.exceptions import ClientError
```

Created separate function to read table name from environment variable. Notice we have also added DocString to each function. DocStrings can be added in multiple formats but here we have used pandas format.

```
def get_table_name():
    """
    Read and return DynamoDB table name from environment variable

    Returns
    -------
    string
        DynamoDB table name stored in environment variable 'TABLE_NAME'
    """
    ret_val = os.environ["TABLE_NAME"]
    return ret_val
```

Created separate function to read table region from environment variable. Please note there is not rule how we can split the program in smaller functions. Here we could also create single function to read all the environment variables like table name, region etc.. Goal is to have logic separation of functionality which is easy to test during automation testing

```
def get_table_region():
    """
    Read and return DynamoDB table region from environment variable

    Returns
    -------
    string
        DynamoDB table region stored in environment variable 'REGION'
    """
    ret_val = os.environ["REGION"]
    return ret_val
```

Added separate function to extract required data from 'event object'. This function also validates query string parameters (check if exists and not blank) and return validation results as a part of return list so that caller function can make a decision accordingly.

```
def extract_event(event):   # new
    """
    Extracts and validates fields from event object passed by API
endpoint

    This function simply returns extracted values as list with first
item
    as validation success as true or false

    Parameters
    ----------
    event : dict
        event object passed from AWS API gateway endpoint

    Returns
    -------
    list
        extracted values from event with first item validation
        success as true or false
    """
    qs_validation_success = True

    # notice use of .get methind on dict rather than directly accessing
key 'name'.
    # get() method provides a way to return even if key does not exist
in dict.
    user_name = event['queryStringParameters'].get('name')
    if user_name is None:
        qs_validation_success = False

    request_method = event['httpMethod']
    request_path= event['path']

    return  [qs_validation_success,user_name,request_method,
request_path]
```

How do it know return format from the operation read the article ‘Boto3 return object to JSON’

```
{
    "ResponseMetadata": {
    "RequestId": "I89GRGTF76RGQBPGMA82NOGK5NVV4KQNSO5AEMVJF66Q9ASUAAJG",
    "HTTPStatusCode": 200,
    "HTTPHeaders": {
    "server": "Server",
    "date": "Tue, 31 May 2022 17:46:42 GMT",
    "content-type": "application/x-amz-json-1.0",
    "content-length": "2",
    "connection": "keep-alive",
    "x-amzn-requestid":
"I89GRGTF76RGQBPGMA82NOGK5NVV4KQNSO5AEMVJF66Q9ASUAAJG",
    "x-amz-crc32": "2745614147"
    },
    "RetryAttempts": 0
    }
}
```

We have also separated code which prepares response object to be returned from lambda function. We can write more code to process input params to create response object dynamically in future. Now it will be easy to write unit test to check how response object is getting created.

```
def prepare_response(status_code,body): # new
    """
    Generate response object to return from API endpoint

    Parameters
    ----------
    event : dict
        event object passed from AWS API gateway endpoint

    Returns
    -------
    list
        extracted values from event with first item validation
        success as true or false
    """
    response_object = {}
    response_object['statusCode'] = status_code
    response_object['headers'] = {}
    response_object['headers']['Content-Type'] = 'application/json'
    response_object['body'] = body
```

Main lambda handler function accepts 'event' object and calls other relevant functions to add data in DynamoDB and return the results.

💡 ***Note :*** API handler implementation is very basic in this example. For better API route handler implementation please go through sample API route handler article.

```
def lambda_handler(event, context):
    """
    Function handles API endpoint requests

    Parameters
    ----------
    event : dict
        event object passed from AWS API gateway endpoint
    context : dict
        object passed to lambda by default on invocation

    Returns
    -------
    dict
        lambda execution details
    """

    response_object = {}

    # Extracts required details from event object.
    # Remember structure of event object stored in apigateway-aws-proxy.
json
    qs_validation_sucess,user_name,request_method,request_path
=  extract_event(event)  #new

    # check for POST operation on user endpoint with query string
validation
    if request_path == "\user" and qs_validation_sucess:    # new
        if request_method == 'POST':
            # call post handler for user endpoint
            post_handler_return = user_post_handler(user_name)
            response_object = prepare_response(   # new
                    post_handler_return['ResponseMetadata']
['HTTPStatusCode'],
                    'user added successfully' if \
                    post_handler_return['ResponseMetadata']
['HTTPStatusCode']==200 \
                    else 'error occured while adding user'
                )
    else:   #new
        response_object = prepare_response(400, 'invalid query
arguments')   # new

    # return from lambda
    return response_object
```

Finally.!! the post handler function to get user_name as input and add it to DynamoDB using boto3 put_item function.

Note : This function has basic error handling using ClientError exception. Please check Python error handling article for more details.

```
def user_post_handler(user_name):   # new
    """
    POST request handler for 'user' API endpoint

    Parameters
    ----------
    user_name : string
        name passed in query string during POST operation

    Returns
    -------
    list
        extracted values from event with first item validation
        success as true or false
    """
    table_region = get_table_region()
    table_name= get_table_name()

    # create dynamodb table object
    ddb_resource = boto3.resource("dynamodb", region_name=table_region)
    ddb_table = ddb_resource.Table(table_name)

    # prepare dict to insert into DynamoDB table
    insert_item = {
        'UserID': str(uuid.uuid1()),
        'UserName' : user_name
    }

    try:
        ddb_response = ddb_table.put_item(
            TableName=table_name,
            Item=insert_item
        )
    except ClientError as e:
        raise e
        #TODO better error handling

    # returning responce from dynamoDB put_item operation
    return ddb_response
```