6. Test Cases for api_lambda

Lets start with simple test case for get_table_name which is `test_get_table_name_success` . This function reads environment variable `TABLE_NAME` from lambda environment. Since this is our development environment, using Pytest we can easily configure any necessary environment variables with the help of 'monkeypatch' which is a built-in pytest fixture that allows us to set environment variables in the test runs.

We can create a function and use monkeypatch as shown below. If we execute this function using pytest then that session will have `MY_ENV_VARIABLE_1` and `MY_ENV_VARIABLE_2` environment variables with value 'some-value' during execution of the test cases.

```
def env_set(monkeypatch):
    monkeypatch.setenv('MY_ENV_VARIABLE_1', 'some-value')
    monkeypatch.setenv('MY_ENV_VARIABLE_2', 'some-value')
```

Now we have net problem how to execute above function automatically before running test cases ? We can simply call this function at the start of each test case but pytest provides better approach of using fixtures.

💡 Fixtures are functions attached to test cases which executes before the test case. For more details on fixtures make sure to read article *understand pytest*

How to make env_set function as fixture? simply decorate it. using `@pytest.fixture` .

How to execute this fixture before each test cases ? simply use `@pytest.fixture(autouse=True)`

All the fixtures are usually part of conftest.py so lets move this fixture code to conftest.py.

```
# conftest.py

import pytest

@pytest.fixture(autouse=True)
def env_set(monkeypatch):
    monkeypatch.setenv('TABLE_NAME', 'user-table')
```

Now when we call get_table_name() from src.api_lambda then it should return 'user-table' while running tests.

To call get_table_name() function from test cases file test_ap_lambda.py we are importing the respective function.

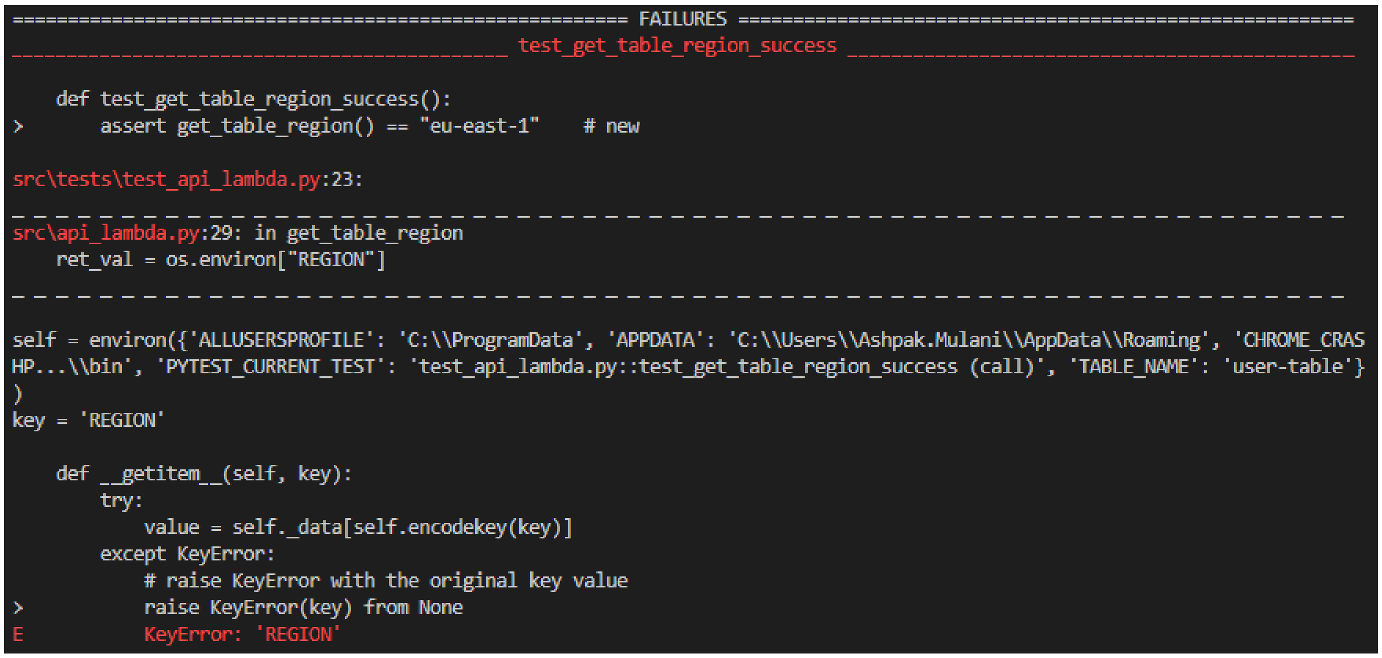
```
import boto3
import moto
import pytest

# import functions fomr source file which we are calling during test
case execution
from src.api_lambda import (        # new
    get_table_name,
    get_table_region,
    prepare_response,
    user_post_handler
)

def test_user_post_handler_success():
    assert True

def test_prepare_response_success():
    assert True

def extract_event():
    assert True

def test_get_table_region_success():
    assert True

# call the respective function and expect it to read environment
variables set using
# monkeypatch
def test_get_table_name_success():
    assert get_table_name() == "user-table"     # new
```

execute the tests again and you should see all test cases passed !!!

Lets add one more test case to test region.

```
def test_get_table_region_success():
    assert get_table_region() == "eu-east-1"    # new
```

If we execute tests again we get following failure. Why ? because we have not set environment variable name ‘REGION’ before executing the test case. We have also understood one fail test scenario which can occur when required environment variable is not present as ‘KeyError’



```
======================================================== FAILURES ========================================================
_____________________________________________ test_get_table_region_success _____________________________________________

    def test_get_table_region_success():
>       assert get_table_region() == "eu-east-1"    # new

src\tests\test_api_lambda.py:23:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
src\api_lambda.py:29: in get_table_region
    ret_val = os.environ["REGION"]
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = environ({'ALLUSERSPROFILE': 'C:\\ProgramData', 'APPDATA': 'C:\\Users\\Ashpak.Mulani\\AppData\\Roaming', 'CHROME_CRAS
HP...\\bin', 'PYTEST_CURRENT_TEST': 'test_api_lambda.py::test_get_table_region_success (call)', 'TABLE_NAME': 'user-table'}
)
key = 'REGION'

    def __getitem__(self, key):
        try:
            value = self._data[self.encodekey(key)]
        except KeyError:
            # raise KeyError with the original key value
>           raise KeyError(key) from None
E           KeyError: 'REGION'
```

Lets add environment variable to fixture in conftest.py and test cases should pass now.

```
# conftest.py

import pytest

@pytest.fixture(autouse=True)
def env_set(monkeypatch):
    monkeypatch.setenv('TABLE_NAME', 'user-table')
    monkeypatch.setenv('REGION', 'user-table')  # new
```

We have also learned around 'KeyError' from our last execution which occurs when there is not specific environment variable.

In following two test cases we are deleting environment variable which are set by fixture at the begging of the test case and then raising KeyError using pytest to assert the error while calling get_table_name() and get_table_region() functions.

Tip : notice how we are asserting using context manager. This is one of the way to assert in pytest without using assert keyword.

```
def test_get_table_name_with_no_env_var(monkeypatch):   # new
    monkeypatch.delenv("TABLE_NAME")
    with pytest.raises(KeyError):
        get_table_name()

def test_get_table_region_with_no_env_var(monkeypatch):   # new
    monkeypatch.delenv("REGION")
    with pytest.raises(KeyError):
        get_table_region()
```

Lets start working on extract_event() function test cases. In success scenario this function does the following things

- Accepts event object from API endpoint

- Read QueryStringParameter frm event and get 'name' param.
- Check if 'name' param exists and not blank
- Read ‘httpMethod' and 'path’ from event object
- Return list with name validation results, name, http method and path details

Now on a high level if we start thinking around unit test cases following cases come to mind quickly

1. event object with all correct fields
2. event object with no 'name' param
3. event object with name param but with blank value
4. 'httpMethod' and 'path' will always be there in event object because they are default keys passed by API gateway. We wont be writing test case to check key not exists scenario

Following test case is covering 1st successful scenario of all correct fields in event.

```
def test_extract_event_success():

    # creating event object same as API gateway event but only using
required keys
    # rather than using all the keys in real event pbject
    event = {
        "body": "eyJ0ZXN0IjoiYm9keSJ9",
        "resource": "/{proxy+}",
        "path": "/user",
        "httpMethod": "POST",
        "isBase64Encoded": True,
        "queryStringParameters": {
                "name": "dummy_user"
        },
        "multiValueQueryStringParameters": {
            "name": [
                    "dummy_user"
            ]
        }
    }

    qs_validation_success,user_name,request_method,request_path =
extract_event(event)
    assert qs_validation_success == True
    assert user_name == 'dummy_user'
    assert request_method == 'POST'
    assert request_path == '/user'
```

If you notice we have created event object within the same test case., but we can easily move it to conftest.py and attach it to this test case so that event object is created automatically before this test case executes.

Following is addition done to conttest.py

```
#conftest.py

@pytest.fixture
def event_object_all_fields():
    return{
        "body": "eyJ0ZXN0IjoiYm9keSJ9",
        "resource": "/{proxy+}",
        "path": "/user",
        "httpMethod": "POST",
        "isBase64Encoded": True,
        "queryStringParameters": {
                "name": "dummy_user"
        },
        "multiValueQueryStringParameters": {
            "name": [
                    "dummy_user"
            ]
        }
    }
```

Following is update done to testcase to link the event object fixture

```
def test_extract_event_success(event_object_all_fields):
    qs_validation_success,user_name,request_method,request_path = \
    extract_event(event_object_all_fields)

    assert qs_validation_success == True
    assert user_name == 'dummy_user'
    assert request_method == 'POST'
    assert request_path == '/user'
```

Test cases for second scenario of not having ‘name' in query string parameter . We have added one more fixture with no 'name’ key in query string param.

```
#Conftest.py

@pytest.fixture
def event_object_no_qs_name():
    return{
        "body": "eyJ0ZXN0IjoiYm9keSJ9",
        "resource": "/{proxy+}",
        "path": "/user",
        "httpMethod": "POST",
        "isBase64Encoded": True,
        "queryStringParameters": {
        },
        "multiValueQueryStringParameters": {
        }
    }
```

created new test case linked with new fixture and asserted respective return values from function.

```
# test_api_lambda.py

def test_extract_event_no_qs_name(event_object_no_qs_name):
    qs_validation_success,user_name,request_method,request_path = \
    extract_event(event_object_no_qs_name)

    assert qs_validation_success == False
    assert user_name == None
    assert request_method == 'POST'
    assert request_path == '/user'
```

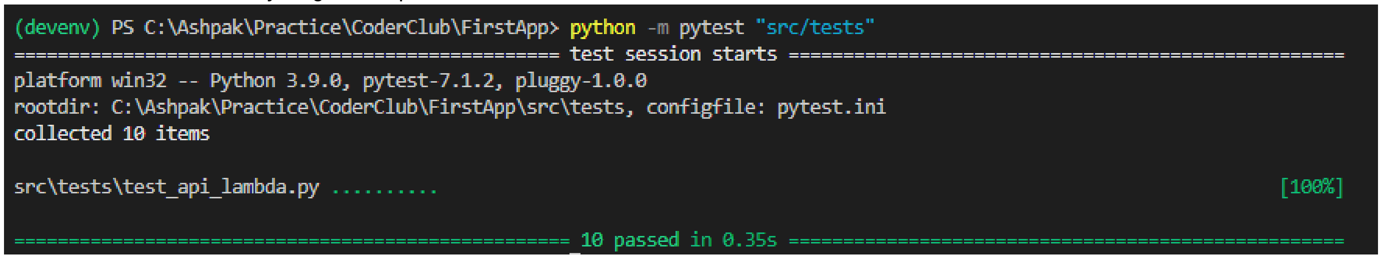
Test case for third scenario with blank name query string. We have added one more fixture with no 'name' key in query string param.

```
#Conftest.py

@pytest.fixture
def event_object_blank_qs_name():
    return{
        "body": "eyJ0ZXN0IjoiYm9keSJ9",
        "resource": "/{proxy+}",
        "path": "/user",
        "httpMethod": "POST",
        "isBase64Encoded": True,
        "queryStringParameters": {
                "name": ""
        },
        "multiValueQueryStringParameters": {
            "name": [
                    ""
            ]
        }
    }
```

equivalent test case

```
def test_extract_event_blank_qs_name(event_object_blank_qs_name):
    qs_validation_success,user_name,request_method,request_path = \
    extract_event(event_object_blank_qs_name)

    assert qs_validation_success == False
    assert user_name == ""
    assert request_method == 'POST'
    assert request_path == '/user'
```

Run the test cases and everything should pass.



```
(devenv) PS C:\Ashpak\Practice\CoderClub\FirstApp> python -m pytest "src/tests"
=================================================== test session starts ====================================================
platform win32 -- Python 3.9.0, pytest-7.1.2, pluggy-1.0.0
rootdir: C:\Ashpak\Practice\CoderClub\FirstApp\src\tests, configfile: pytest.ini
collected 10 items

src\tests\test_api_lambda.py ..........                                                                              [100%]

==================================================== 10 passed in 0.35s ====================================================
```

Going good so far…keep it up we have came a long way..!!!!

Now lets start writing test cases for prepare_response() function. This function accepts status_code and body as param and returns a dict for response object from lambda function to API endpoint. Since this is internal function (called by other functions) there is not any data validation logic present in it.

Here we calling functions with required attributes and asserting if return value is dict and one of the key StatusCode to make sure return object is formed correctly using input params.

```
# test_api_lambda.py

def test_prepare_response_success():

    ret = prepare_response(200,'success')

    assert isinstance(ret, dict)
    assert ret['statusCode'] == 200
```

Finally we will write test case for user_post_handler() function. This function accepts user_name as param and performs following activities

- get table name and region from environment variable through respective functions
- create boto3 resource object for DynamoDB
- Create DynamoDB table object through resource object
- prepare dict using uuid and user name
- add dict to DynamoDB table using put_item
- handles ClientError during put_item
- return put_item operation response

Lets start thinking around test scenarios now.

1. We don't have to write any test cases for get table name and region from environment variable because those individual functions are already covered.
2. DynamoDB resource creation success using boto3
3. DynamoDB table object creation success
4. Success scenario for correct dict object inserted in DynamoDB table
5. Invalid data (type) in dict while inserting data in DynamoDB table
6. Invalid table name while inserting data in table
7. Event if same user name passed twice DynamoDB should have two separate items because UUID will be unique every time which can create different combination ok PK (ID) and SK (user name)

If you notice point 2,3 we won’t be able to write unit test case for the scenario, because related code is inside same big function. In point 2 and # we are creating boto3 resource object for DynamoDB service and creating a table object using resource object, if we move this code to separate function then apart from easily writing unit test cases we can also cache function results because it may be used multiple times in code. It make sense to refactore user_post_handler() function put code for point 2,3 in separate functions.

This is one of the benefits of having unit test cases. We end up breaking monolithic code into smaller testable units.

Following is just updated part in api_handler_lambda function. We have separated boto3 resource creation and dynamodb table object creation in different functions. (Not pasting all code as its a big code now)

Tip : we have used functools caching mechanism tocache results of boto3 resource and dynamodb table to bring in efficiency. For more information on functool caching please check article “*most ignored caching*'

```
import functools  # new

@functools.lru_cache(1)   # new
def get_dynamo_db_resource(region_name):    # new
    return boto3.resource("dynamodb", region_name=region_name)

@functools.lru_cache(1)   # new
def get_dynamodb_table(table_name,table_region):    # new
    dynamodb = get_dynamo_db_resource(table_region)
    table = dynamodb.Table(table_name)
    return table


def user_post_handler(user_name):
    table_region = get_table_region()
    table_name= get_table_name()

    # create dynamodb table object
    ddb_table = get_dynamodb_table(table_name, table_region)  # new

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

Now lets add unit test case for `get_dynamo_db_resource` and `get_dynamodb_table` functions to check if functions are returning correct objects matching with passed params.

Following are success test cases added. We are using moto library to mock all calls to AWS services. In both the functions we are creating objects for boto3 resources and DynamoDB table using respective functions and asserting them with normally created objects with boto3, to make sure functions are returning correct objects as per provided parameters.

Tip : For more information in moto please go though article 'understand moto'

```
import boto3
import pytest
import os       # new
from moto import mock_dynamodb    # new

from src.api_lambda import (
    get_table_name,
    get_table_region,
    extract_event,
    prepare_response,
    get_dynamo_db_resource,  # new
    get_dynamodb_table ,    # new
    user_post_handler
)

def test_user_post_handler_success():
    assert True

def test_get_dynamo_db_resource():  # new
    with mock_dynamodb():
        resource = get_dynamo_db_resource('eu-east-1')
        assert resource == boto3.resource("dynamodb", 'us-east1')


def test_get_dynamo_db_table():  # new
    with mock_dynamodb():
        db_table = get_dynamodb_table('test_table','us-east-1')
        resource = boto3.resource("dynamodb", 'us-east1')
        table = resource.Table('test_table')
        assert db_table == table

..........  rest of the code .................
```

By adding unit test cases above we have covered scenario 2 and 3 of unit test cases for user_post_handler() function. Now continue with scenario 4.

We need to insert correct sample data in DynamoDB table. Environment variables stores Table name and region (which we will be mocking through monkeypatch for testing) and before running code to insert item in DynamoDB table inside unit test, we need to make sure DynamoDB table is in place.

Lets write code to create a DynamoDB table as per schema mentioned in the project summary (UserID - PK, UserName - SK). Following function will create a DynamoDB table and we will need to make sure we are calling this function at the begging of running unit test cases. To call this function when using Pytest for testing we can make it as a fixture and link the fixture to respective test cases to make sure it's called before test case execution.

We have added @pytest.fixture decorator on to of the function to make it a fixture and since we are defining all fixtures in conftest.py we are going to add following code in conftest.py file

```
#conftest.py

@pytest.fixture
def dynamodb_table():
    resource = boto3.resource('dynamodb','us-east-1')
    resource.create_table(
        TableName=os.environ["TABLE_NAME"],
        AttributeDefinitions=[
            {'AttributeName': 'UserID', 'AttributeType': 'S'},
            {'AttributeName': 'UserName', 'AttributeType': 'S'}
        ],
        KeySchema=[
            {'AttributeName': 'UserID', 'KeyType': 'HASH'},
            {'AttributeName': 'UserName', 'KeyType': 'RANGE'}
            ],
        ProvisionedThroughput={
            'ReadCapacityUnits': 5,
            'WriteCapacityUnits': 5,
        }
    )
```

We need to make sure above function code is creating the DynamoDB table in mocked environment and not in real AWS environment. To make sure it’s creating table in mocked environment we can use moto mock_dynamodb(). We are using this as a context manager for a reason, because we can control its context easily. If you notice we have used 'yield' at the end of the function because we want to make sure same mocked context is present inside our unit test cases as well, which are linked to this fixture.

💡 using yield in pytest fixture is common thing because it allows execution of code before yield as ‘setup' and destroys resources created automatically after test case execution as ‘teardown’. Please go through ‘understand pytest’ and 'understand moto’ articles for more details.

```
@pytest.fixture
def dynamodb_table():
    with mock_dynamodb():   # new
        resource = boto3.resource('dynamodb','us-east-1')
        resource.create_table(
            TableName=os.environ["TABLE_NAME"],
            AttributeDefinitions=[
                {'AttributeName': 'UserID', 'AttributeType': 'S'},
                {'AttributeName': 'UserName', 'AttributeType': 'S'}
            ],
            KeySchema=[
                {'AttributeName': 'UserID', 'KeyType': 'HASH'},
                {'AttributeName': 'UserName', 'KeyType': 'RANGE'}
                ],
            ProvisionedThroughput={
                'ReadCapacityUnits': 5,
                'WriteCapacityUnits': 5,
            }
        )

        yield   # new
```

Now lest move on to writing unit test case for ‘user_post_handler' function. As per code below we are passing fixture 'dynamodb_table’ to this unit test case to make sure fixture code is executed and table is ready before code in unit test case starts its execution.

Since we have used 'yeild' at the end of the mock_dynamodb context, it makes sure mock context continues in the unit test case code and executes all the code under unit test case in same mock_dynamodb context. This is the reason you dont see either @mock_dynamodb decorator or mock_dynamodb() context used in unit test code. If we used separate mock in unit test code then it will create separate mock context than what is created in fixture and our code in unit test case will execute in that separate mock context and wont be able to find the dynamodb table.

```
def test_user_post_handler_success(dynamodb_table):
    ret = user_post_handler('sample_user')
    assert ret['ResponseMetadata']['HTTPStatusCode'] == 200
```

good so far..!!!

In the test case above we are just checking return code from user post handler function. We can make this test case more stronger by reading item from DynamoDB table and making sure it has same value what we have inserted.

```
def test_user_post_handler_success(dynamodb_table):
    ret = user_post_handler('sample_user')

    db_resource = boto3.resource('dynamodb',os.environ["REGION"])   #
new
    table = db_resource.Table(os.environ["TABLE_NAME"])    # new
    resp = table.scan(ProjectionExpression="UserID, UserName")     #
new
    assert resp['Items'][0]['UserName'] == 'sample_user'    # new
    assert resp['Count'] == 1   # new

    assert ret['ResponseMetadata']['HTTPStatusCode'] == 200
```

Above test case will certainly pass, but we can optimize it more by leveraging boto3.resource object context from same fixture 'dynamodb_table'. Instead of plain 'yield' we can yield resource object from fixture as show below.

```
@pytest.fixture
def dynamodb_table():
    with mock_dynamodb():
        resource = boto3.resource('dynamodb','us-east-1')
        resource.create_table(
            TableName=os.environ["TABLE_NAME"],
            AttributeDefinitions=[
                {'AttributeName': 'UserID', 'AttributeType': 'S'},
                {'AttributeName': 'UserName', 'AttributeType': 'S'}
            ],
            KeySchema=[
                {'AttributeName': 'UserID', 'KeyType': 'HASH'},
                {'AttributeName': 'UserName', 'KeyType': 'RANGE'}
                ],
            ProvisionedThroughput={
                'ReadCapacityUnits': 5,
                'WriteCapacityUnits': 5,
            }
        )
        yield resource # new
```

and use returned object from fixture instead of creating one more resource object as shown below. This concludes success unit test case for function which add data to dynamodb table by creating new table and verify it by reading back the data for added item.

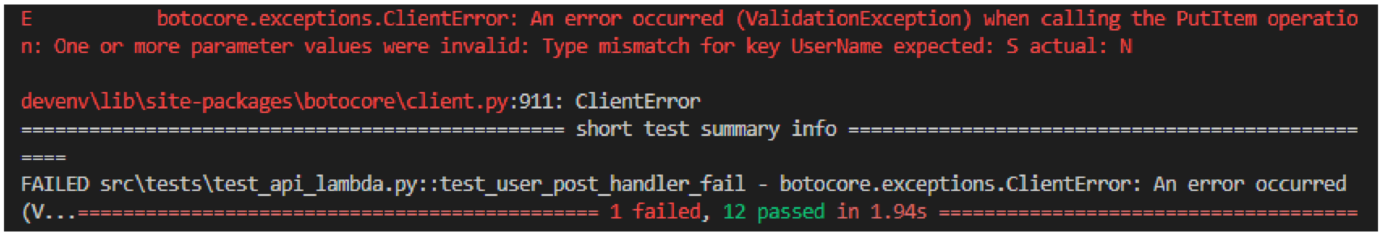
```
def test_user_post_handler_success(dynamodb_table):
    ret = user_post_handler('sample_user')

    table = dynamodb_table.Table(os.environ["TABLE_NAME"])    # new
    resp = table.scan(ProjectionExpression="UserID, UserName")
    assert resp['Items'][0]['UserName'] == 'sample_user'
    assert resp['Count'] == 1

    assert ret['ResponseMetadata']['HTTPStatusCode'] == 200
```

Now lest start writing unit test case for scenario 5 for failure test case

For failure test case we will use same approach as success test case but lets try passing integer value in user name in test case and see what happens.

```
def test_user_post_handler_fail(dynamodb_table):
    user_post_handler(1234)
```

if we run `python -m pytest "src/tests"` then we get following error



```
E           botocore.exceptions.ClientError: An error occurred (ValidationException) when calling the PutItem operatio
n: One or more parameter values were invalid: Type mismatch for key UserName expected: S actual: N

devenv\lib\site-packages\botocore\client.py:911: ClientError
=============================================== short test summary info ===============================================
====
FAILED src\tests\test_api_lambda.py::test_user_post_handler_fail - botocore.exceptions.ClientError: An error occurred
(V...============================================== 1 failed, 12 passed in 1.94s =====================================
```

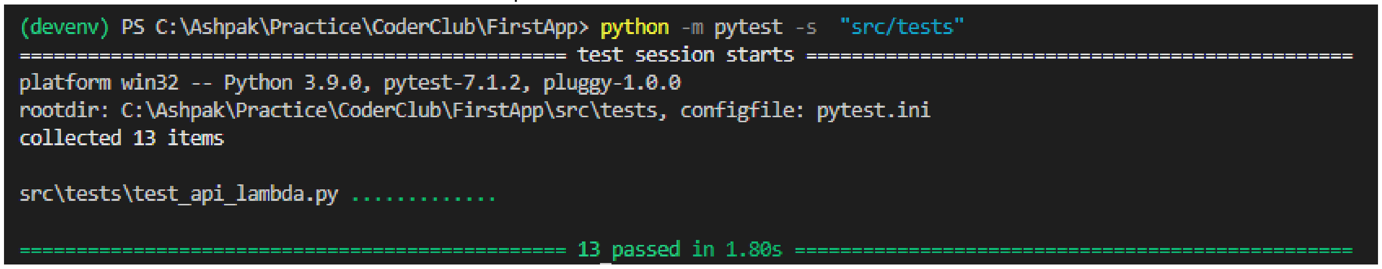
This error helps us understand what we should be asserting in our test case to make it a success. We simply need to assert pytest exception. We can also add one more check to confirm error contains the part string from above exception to make sure we are asserting only error we wanted and not other.

> I always use this trick for finding what to assert in failed test cases by making code fail deliberately

Following code raises Exception and add match expression to make sure its the same error what we are looking for. Since we are using context manager '`with`' we don't have to explicitly use `assert` statement.

```
def test_user_post_handler_fail(dynamodb_table):
    with pytest.raises(Exception, match="Type mismatch for key UserName
expected: S actual: N"):
        user_post_handler(1234)
```

if we execute the tests we should see all the cases as passed now.

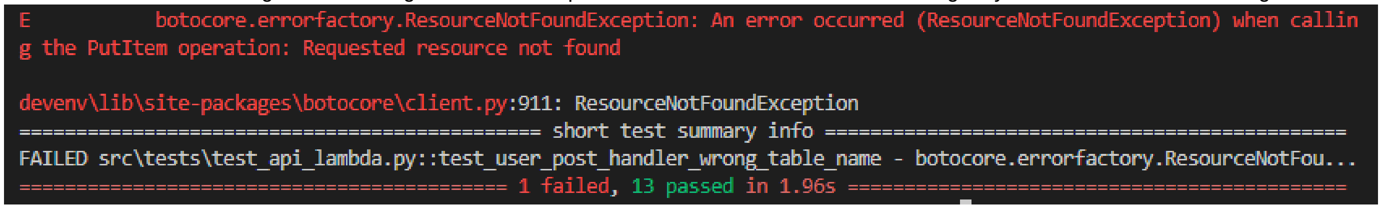


```
(devenv) PS C:\Ashpak\Practice\CoderClub\FirstApp> python -m pytest -s  "src/tests"
================================================ test session starts ================================================
platform win32 -- Python 3.9.0, pytest-7.1.2, pluggy-1.0.0
rootdir: C:\Ashpak\Practice\CoderClub\FirstApp\src\tests, configfile: pytest.ini
collected 13 items

src\tests\test_api_lambda.py .............

================================================ 13 passed in 1.80s ================================================
```

Scenario 6 : invalid/non-existing dynamodb table. We can easily simulate this test case by simply not using fixture which creates the table. We can use mock_dynamodb() context in this test case before calling user_post_handler function to make sure calls in user_post_handler function are reaching mock environment.

```
def test_user_post_handler_wrong_table_name():
    with mock_dynamodb():
        ret = user_post_handler('sample_user')
```

if we execute tests now we get the following error. This is expected because we have not created target dynamodb table before adding item in it.



```
E         botocore.errorfactory.ResourceNotFoundException: An error occurred (ResourceNotFoundException) when callin
g the PutItem operation: Requested resource not found

devenv\lib\site-packages\botocore\client.py:911: ResourceNotFoundException
============================================= short test summary info =============================================
FAILED src\tests\test_api_lambda.py::test_user_post_handler_wrong_table_name - botocore.errorfactory.ResourceNotFou...
=========================================== 1 failed, 13 passed in 1.96s ============================================
```

lest add one more context manager to raise and asset exception for resource not found as shown below.

All test case should pass now.

```
def test_user_post_handler_wrong_table_name():
    with mock_dynamodb(), pytest.raises(Exception, match="Requested
resource not found"):
            ret = user_post_handler('sample_user')
```

In scenario 7, to test if multiple items are created in dynamodb table if same user name is passed multiple times, we can simply tweak already written test case 'test_user_post_handler_success' to add multiple items by calling `user_post_handler()` function multiple times. We can then verify inserted items by running scan on table as shown below

```
def test_user_post_handler_multiple_items(dynamodb_table):
    ret_1 = user_post_handler('sample_user')
    ret_2 = user_post_handler('sample_user')

    table = dynamodb_table.Table(os.environ["TABLE_NAME"])
    resp = table.scan(ProjectionExpression="UserID, UserName")

    assert resp['Count'] == 2
    assert resp['Items'][0]['UserName'] == 'sample_user'
    assert resp['Items'][1]['UserName'] == 'sample_user'
    assert ret_1['ResponseMetadata']['HTTPStatusCode'] == 200
    assert ret_2['ResponseMetadata']['HTTPStatusCode'] == 200
```

Now finally test lambda handler which combines all other functions. Following is high level list of functionalities from lambda handler function.

1. Extracts and validates different keys from event object
2. calls post handler function if validation succeeds and request method is POST for /user endpoint
3. prepare and return success response object if return from post handler is success
4. prepare and return failure object with error code 400 if validation failed
5. prepare and return failure object with error code 400 if endpoint url is not /user
6. prepare and return failure object with error code 400 if request method is not POST

scenario 1 and 2 will be tested automatically when we call the lambda function using 'event' object and we have already tested different units of that functionality using multiple tests cases for individual functions.

For scenario 3, we need to pass event object for success scenario with all correct fields and check if lambda function is returning success in response object. Since lambda function will be adding items in DynamoDB table we need to make sure table is in place by passing our fixture dynamodb_table which created a table and yeilds mocked context so that code in test case can be executed in same context.

```
def test_lambda_handler_success(dynamodb_table):

    event = {
        "path": "/user", "httpMethod": "POST",
        "queryStringParameters": {"name": "sample_user_name" }
            }

    resp = lambda_handler(event, None)

    assert resp['statusCode'] == 200
    assert resp['body'] == 'user added successfully'
```

For scenario 4 lets pass event object for failure scenario, with blank or no ‘name' field in query string param under event object. We are asserting status code as 400 and response body as ‘invalid query argument’ because that’s what we are returning in this scenario.

```
def test_lambda_handler_validation_failure(dynamodb_table):
    event = {
        "path": "/user", "httpMethod": "POST",
        "queryStringParameters": { }
            }

    resp = lambda_handler(event, None)

    assert resp['statusCode'] == 400
    assert resp['body'] == 'invalid query arguments'
```

For scenario 5, if we pass URL other than /user then we end up having test case like below whihc doesnt seems correct in terms of response body.

```
def test_lambda_handler_wrong_path(dynamodb_table):
    event = {
        "path": "/user/wrongpath", "httpMethod": "POST",
        "queryStringParameters": {"name": "sample_user_name" }
            }

    resp = lambda_handler(event, None)

    assert resp['statusCode'] == 400
    assert resp['body'] == 'invalid query arguments'
```

Pay close attention now, we are not getting glimpse of how TDD works. Where we first write test case then then update the code to pass the test case.

For wrong URL path response body should be something like “invalid request URL“, so correct test case would be

```
def test_lambda_handler_wrong_path(dynamodb_table):
    event = {
        "path": "/user/wrongpath", "httpMethod": "POST",
        "queryStringParameters": {"name": "sample_user_name" }
            }

    resp = lambda_handler(event, None)

    assert resp['statusCode'] == 400
    assert resp['body'] == 'invalid request url'
```

If we run this test case now it will fail because our code is not equipped to return ‘invalid request URL'. This means we are in RED zone now. So now let’s update our code to make sure this test case passes. As shown below we have added one more if else to return correct body message.

```
    if request_path == "/user":
        if qs_validation_success:   # new
            if request_method == 'POST':
                # call post handler for user endpoint
                post_handler_return = user_post_handler(user_name)
                response_object = prepare_response(
                        post_handler_return['ResponseMetadata']
['HTTPStatusCode'],
                        'user added successfully' if \
                        post_handler_return['ResponseMetadata']
['HTTPStatusCode']==200 \
                        else 'error occured while adding user'
                    )
        else:
            response_object = prepare_response(400, 'invalid query
arguments')
    else:   # new
        response_object = prepare_response(400, 'invalid query
arguments')
```

Now if we run our test, it should pass, which means we are in GREEN phase. All our test cases are passing now. If you notice just to make test case pass we ended up adding nested if else which is not readable. So we have refactor updated code a bit to make it readable. We have removed nested if, else and made it little simple by refactoring the code.

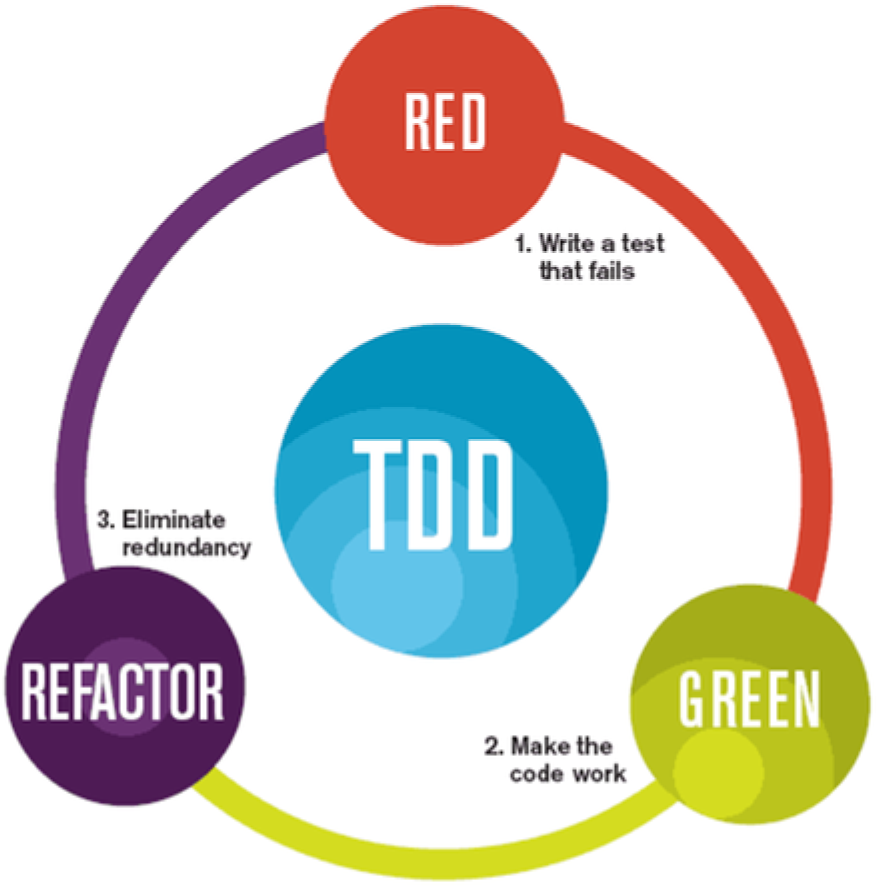
```
if request_path != "/user": # new
        response_object = prepare_response(400, 'invalid request url')
        return response_object

    if not qs_validation_success:   # new
```

```
        response_object = prepare_response(400, 'invalid query
arguments')
        return response_object

    if request_method == 'POST':
        post_handler_return = user_post_handler(user_name)
        response_object = prepare_response(
                post_handler_return['ResponseMetadata']
['HTTPStatusCode'],
                'user added successfully' if \
                post_handler_return['ResponseMetadata']
['HTTPStatusCode']==200 \
                else 'error occured while adding user'
            )
   return response_object
```

Now if we run test cases, everyting should pass. To summarize we followed threee phases

*RED (failing test case) ---> GREEN (updated code to pass test case) ---> Ref*actor(made code more readable)

Which is exactly what TDD process is as shown in diagram below.

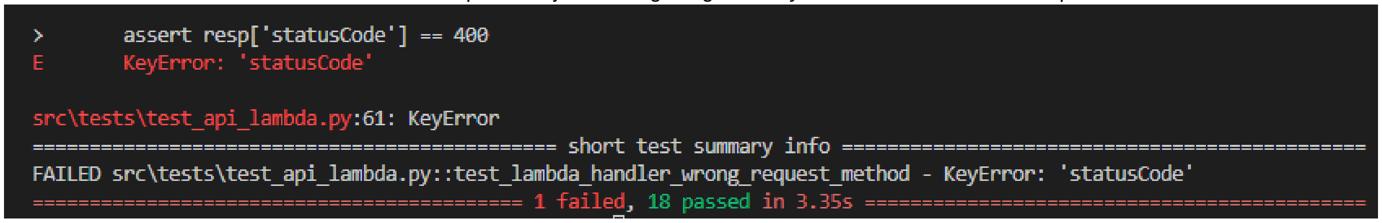


We will follow TDD for scenarios 6 ie. RED-GREEN-REFACTOR approach.

In this scenarios if we request other methods than POST then lambda should return error. Error message should be something like “invalid request method” with return code as 400. Our test case would look like below.

```
def test_lambda_handler_wrong_request_method(dynamodb_table):
    event = {
        "path": "/user", "httpMethod": "DELETE",
        "queryStringParameters": {"name": "sample_user_name" }
            }

    resp = lambda_handler(event, None)

    assert resp['statusCode'] == 400
    assert resp['body'] == 'invalid request method'
```

if we run test case it will fail because it seems response object is not getting correctly formed. So we are in RED phase now.



```
>       assert resp['statusCode'] == 400
E       KeyError: 'statusCode'

src\tests\test_api_lambda.py:61: KeyError
============================================== short test summary info ==============================================
FAILED src\tests\test_api_lambda.py::test_lambda_handler_wrong_request_method - KeyError: 'statusCode'
=========================================== 1 failed, 18 passed in 3.35s ============================================
```

Lets update the code by simply adding 'else' part and prepare response object. Test cases should bepassing now. So we are in GREEN phase again.

```
   if request_method == 'POST':
        post_handler_return = user_post_handler(user_name)
        response_object = prepare_response(
                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 request
method')
```

after adding else there is not much to refactor for now so we can skip REFACTOR stage.