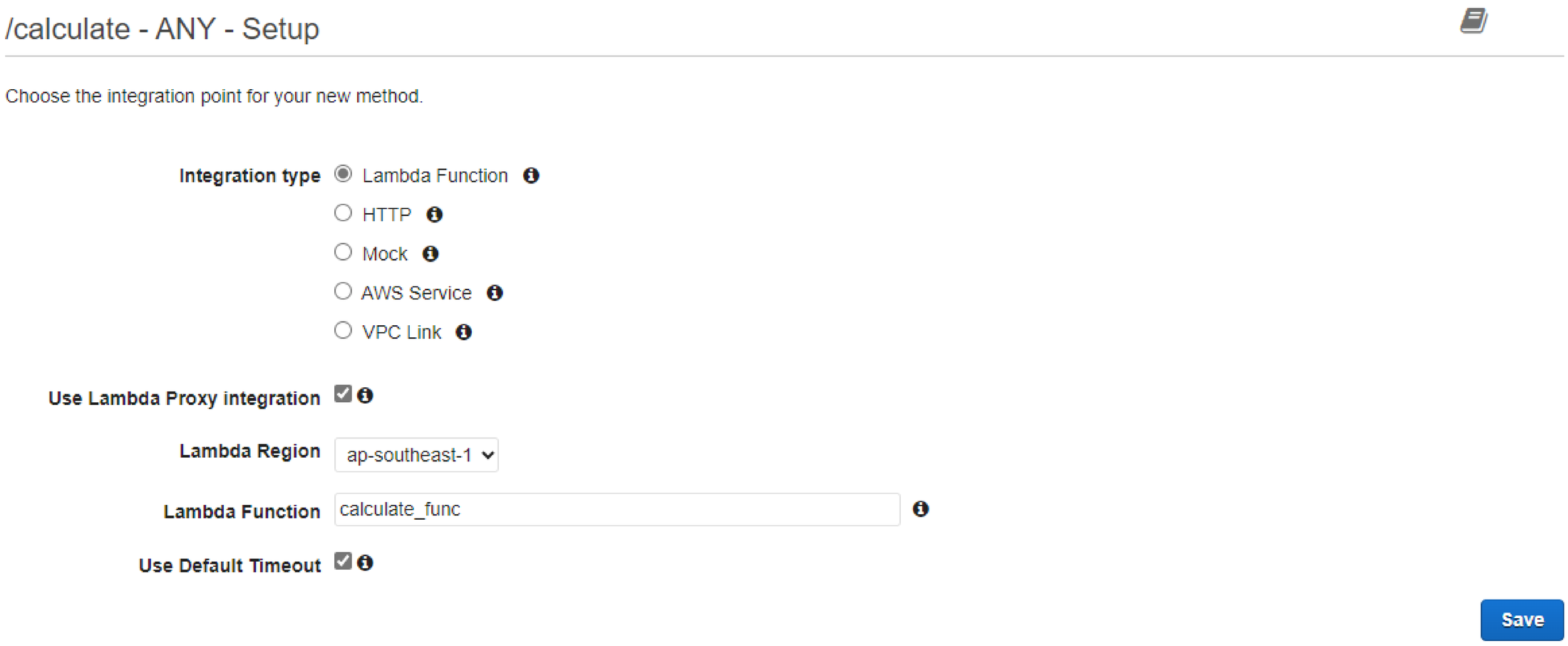
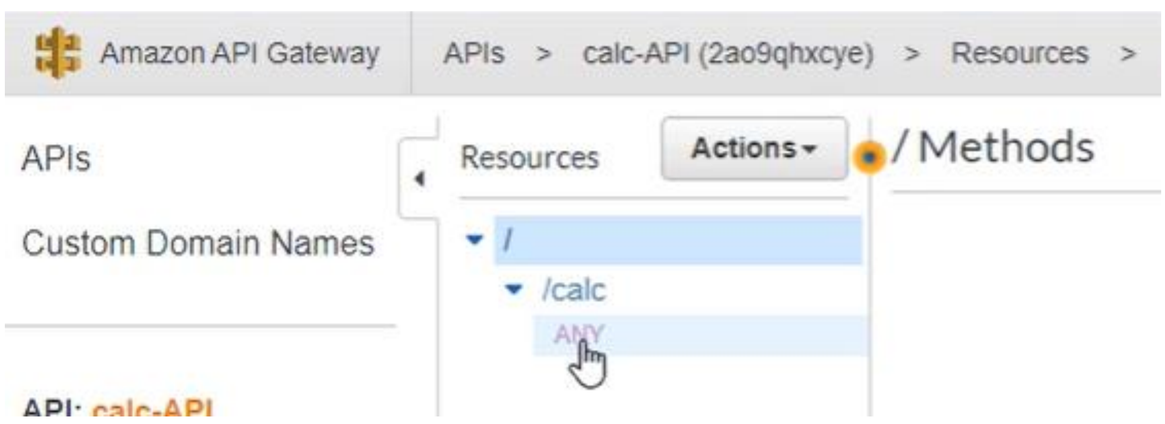
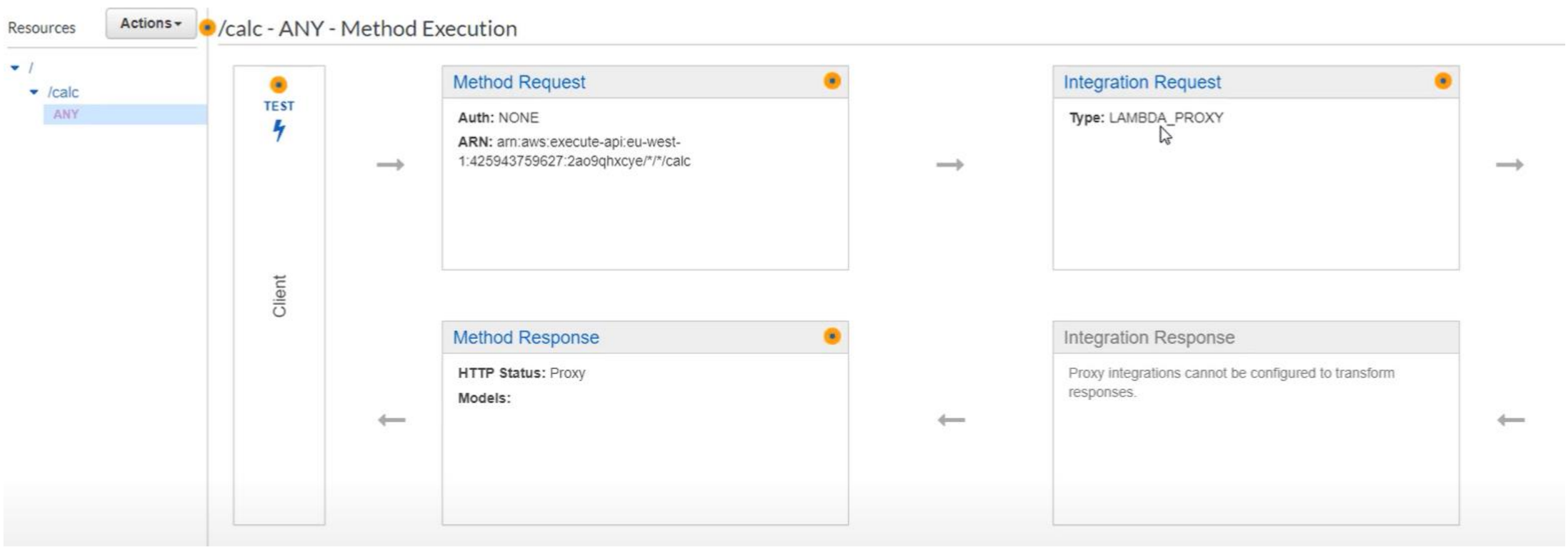
+ Create new Resource (in my case, it’s /calculate), then create new Method which is “ANY”, remember to tick on “Use Lambda Proxy integration” (Lambda function was created before – calculate_func)



NOTE:
- The button “Send” in the POSTMAN -> Click it to invoke (our) Lambda function
- GET: using Query String Parameters
- POST: using Request Body Parameters
- By default, the request created for us by AWS is an ANY request because our Lambda is in Lambda Proxy mode. (NEW VERSION, IF NEEDED, WE HAVE TO TICK ON “Use Lambda Proxy Integration” IN “Integration Request” TO APPLY LAMBDA_PROXY)



- (in ANY) the Integration Request is “LAMBDA_PROXY”:
+ “LAMBDA_PROXY” allow you to change the lambda function implementation at any time without needing to redeploy the API, this means that you can change your lambda function and then don’t have to come back to API Gateway to update it.



+ At runtime, API Gateway will map the incoming request into the input **event** parameter of the lambda function. The input includes the “Request Method”, “Path”, “Headers”, any “Query String Parameters”, any “Payload” which will be a JSON string, and other items…
+ Here’s the Input format of a Lambda function for proxy integration:
https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-lambda-proxy-integrations.html#api-gateway-simple-proxy-for-lambda-input-format

```
{
    "resource": "Resource path",
    "path": "Path parameter",
  "httpMethod": "Incoming request's method name",
  "headers": "{String containing incoming request headers}",
  "multiValueHeaders": "{List of strings containing incoming request headers}",
  "queryStringParameters": "{query string parameters}",
  "multiValueQueryStringParameters": "List of query string parameters",
  "pathParameters": "{path parameters}",
  "stageVariables": "{Applicable stage variables}",
  "requestContext": "{Request context, including authorizer-returned key-value pairs}",
  "body": "a JSON string of request payload",
  "isBase64Encoded": "a boolean flag to indicate if the applicable request payload is Base64-encode"
}
```

GET

- Query Parameters:

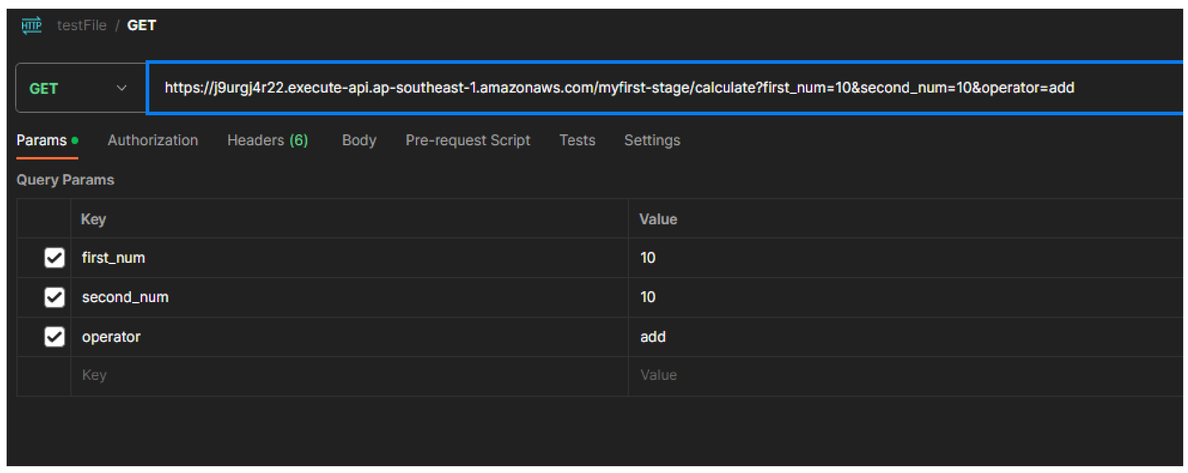
https://myendpoint.com?first_parameter=1&second_parameter=2

<u>No matter the order of parameter, the above the same as the below:</u>

https://myendpoint.com?second_parameter=2&first_parameter=1

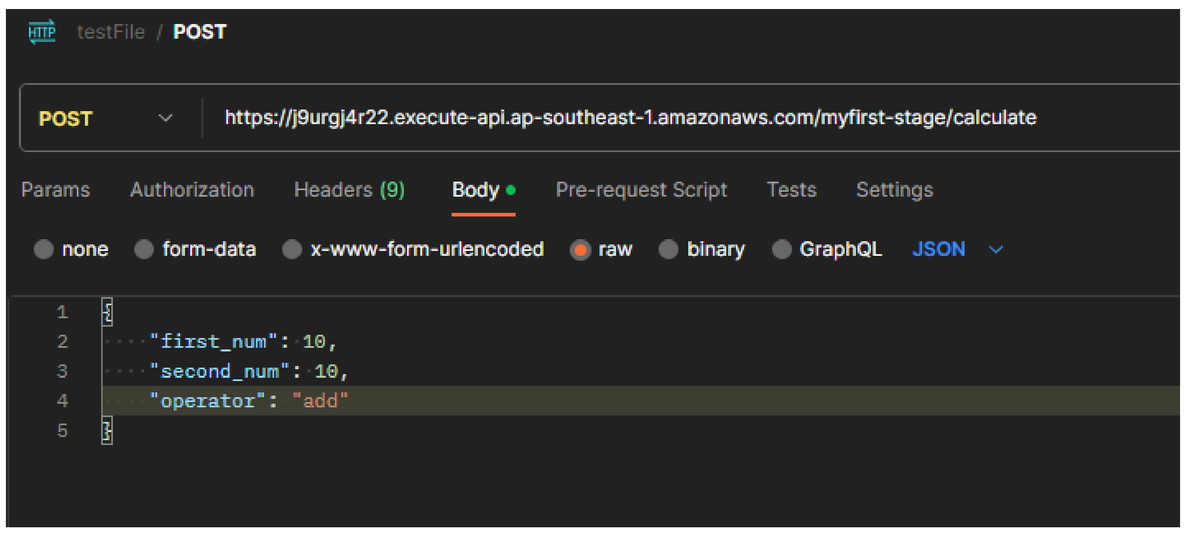
+ To create a GET Request (that will go to (our) Lambda function) which has 2 numbers and 1 operator (add), it'll look like this:

https://apiendpoint?first_num=10&second_num=10&operator=add



POST

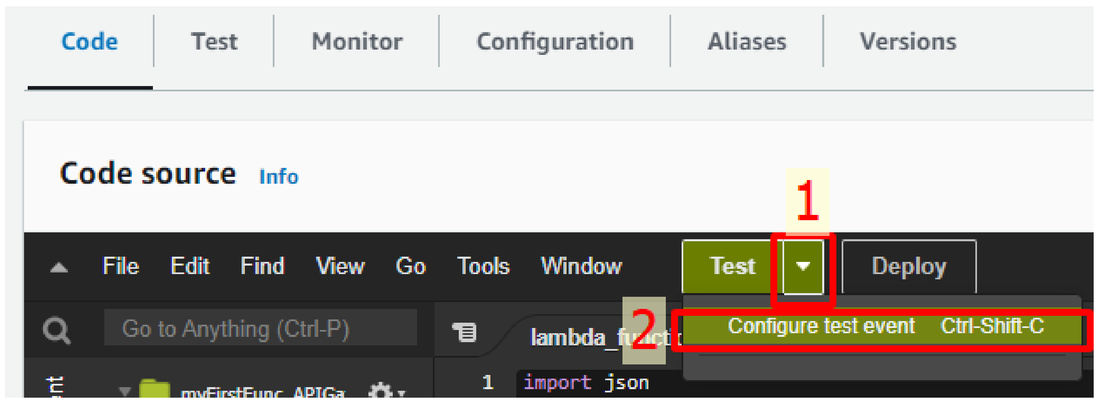
- Request Body:



+ To use these (three) parameters, we have to modified in lambda function. (next page)

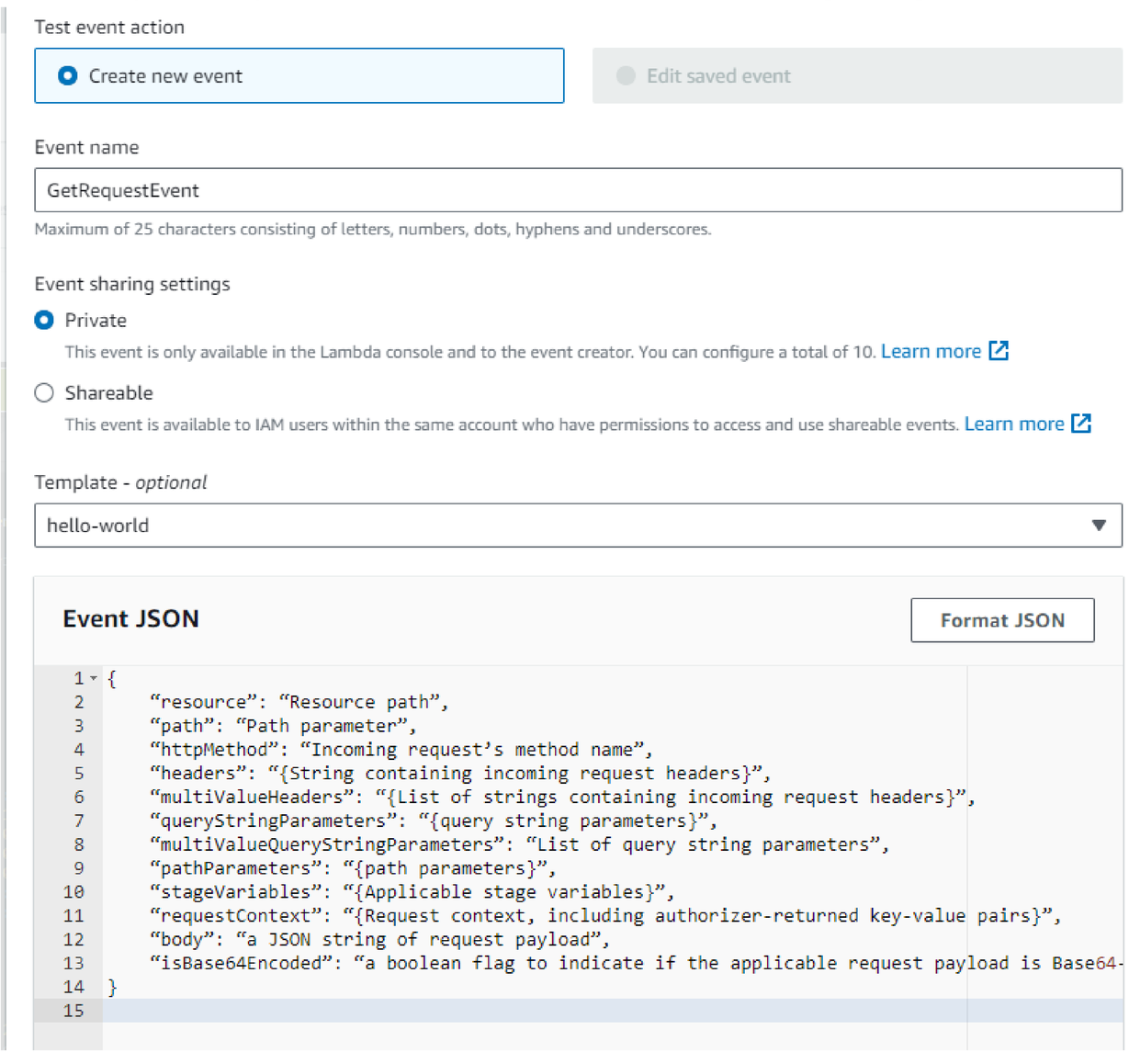
+ The first thing that we need to do is:

++ Create the get request and post request test payloads: Click on the "Test" label -> "Configure test event"

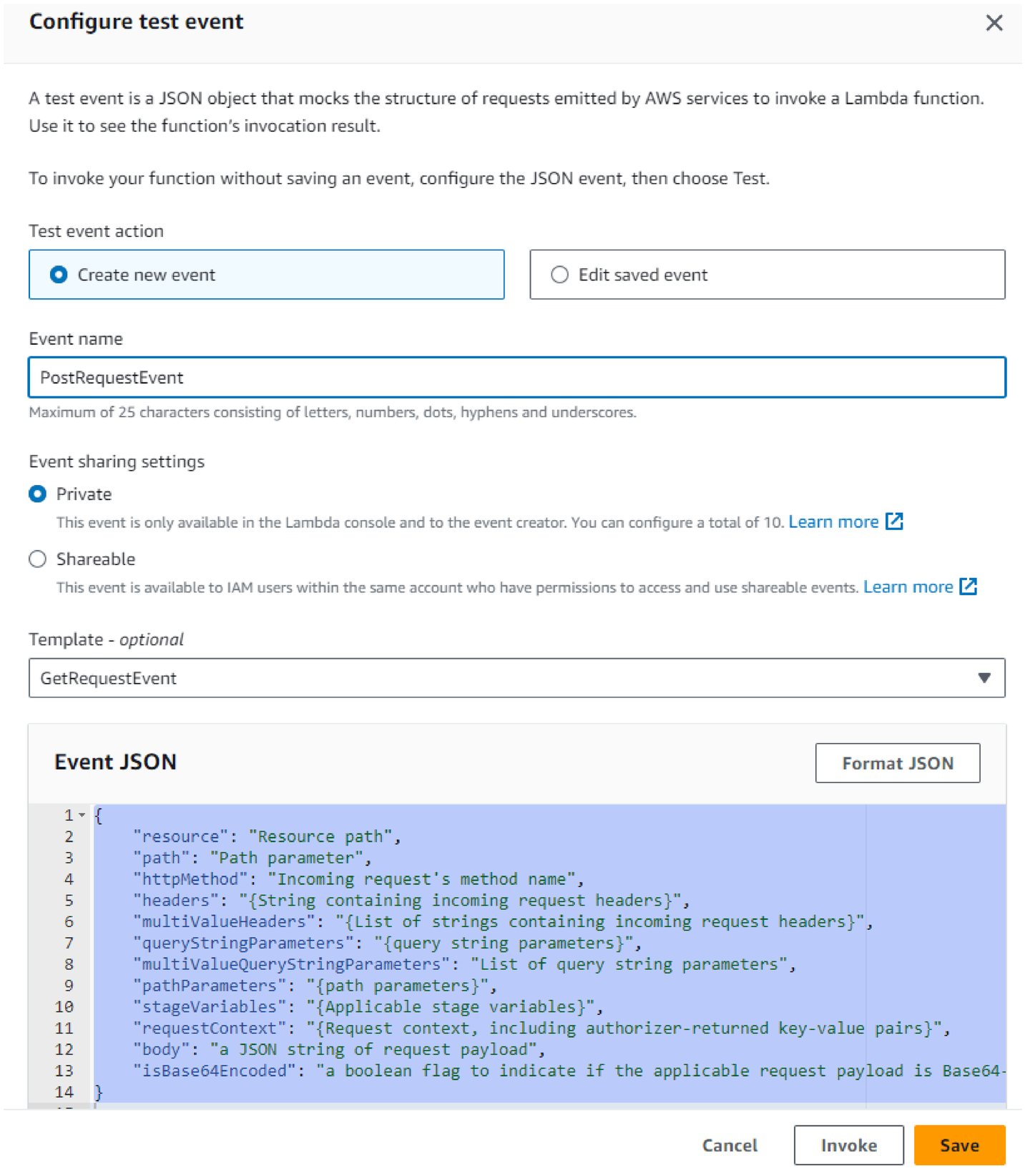


++ "Create new event" -> Enter name (GetRequestEvent)

++ It doesn't matter which Template you choose because we're going to overwrite it: (Then save)



++ Do the same for POST request (same procedure)



++ Now we need to modify in Event JSON, we gonna start with GET request first so just select the drop down beside the “Test” and click on “Configure test event”. At “Event name” => change to “Get…” and do some modify in “Event JSON”: (Note: stay at “Edit saved event”)

+++ Change resource => Put the name of our lambda function (in my case: /calculate)

+++ Change path => (In my case it’s the same as resource) (https://.../<stage_name>/calculate)

+++ Change httpMethod => We’re using GET method so it is => GET

+++ On the headers, we’re not sending any headers so you can send back null which is like this:

"headers": null,

or you can just delete it (I prefer deleteing it because it makes clearer to read)

+++ I’m not gonna send “multiValueHeaders” so => delete it

+++ “queryStringParameters” => This is GET request so I’m definitely going to have query string parameters. They’ll be in the format of a JSON string like this: (NOTICE HERE, I’M SENDING THE VALUES AS STRINGS BECAUSE THEY’RE IN THE QUERY STRING PARAMETERS)

"queryStringParameters": {"first_num":"10", "second_num":"10", "operator":"add"}

+++ The rest components I can delete them because I’m not sending them

⇒With that, we have fixed our get request event and this is the one that we went to send to our lambda function.

```
{
    "resource": "/calculate",
    "path": "/calculate",
    "httpMethod": "GET",
    "queryStringParameters":
    {
        "first_num":"10",
        "second_num":"10",
        "operator":"add"
    }
}
```

+++ Go on modify the POST Request event:

/*

We need to send a JSON string of the request payload, so remember: our request payload is JSON, but we need to send **a JSON string** of that JSON request. Here is the POST Request String Body:

"body": "{\"first_num\":10,\"second_num\":10,\"operator\":\"add\"}"

*/

```
{
    "resource": "/calculate",
    "path": "/calculate",
    "httpMethod": "POST",
    "body": "{\"first_num\":10,\"second_num\":10,\"operator\":\"add\"}"
}
```

++ Now, we'll modify lambda function to process these parameters:

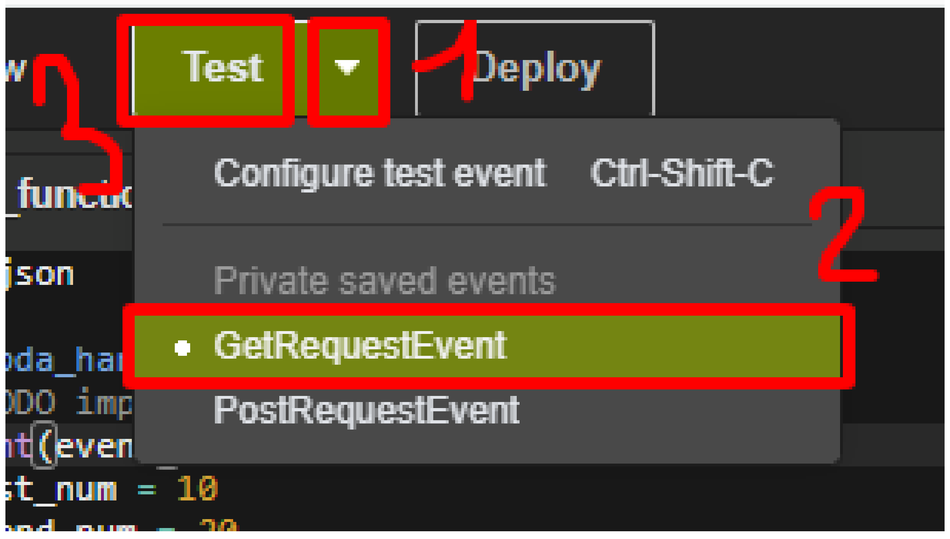
+++ First, let's test print the event parameter to see what our request is bring in, so just write "print(event)":

lambda_function × Execution results ×

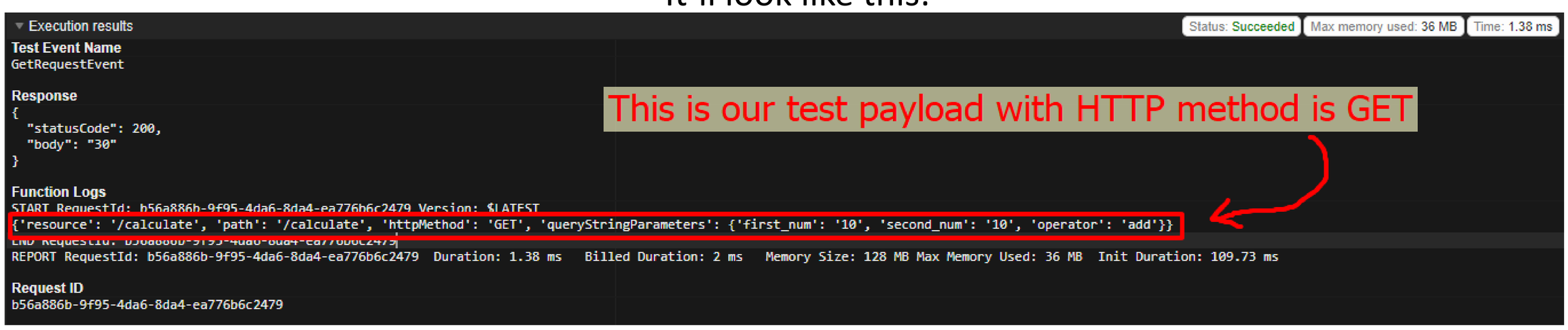
```
import json

def lambda_handler(event, context):
    # TODO implement
    print(event)
    first_num = 10
    second_num = 20
    sum = first_num + second_num
    return {
        'statusCode': 200,
        'body': json.dumps(sum)
    }
```

(Note: after every change in code, remember to "Deploy")

+++ To see the test payload of GET request:



It'll look like this:



+++ To see the test payload of POST request, do the same way.

+++ Next we need to set defaults, so the first_num and second_num will be zero, and then create an operator variable (which is "add") and then we'll make them global. This means that there will not be under any method.

```
import json

first_num = 0
second_num = 0
operator = "add"
```

```python
def lambda_handler(event, context):
    # TODO implement
```

+++ This lambda_handler function both GET and POST request, so we need to make a selection (This is because for a GET request, the data is in the query string and for a POST request the data is in the body). So inside “def lambda_handler…”, after open the event we need to make a SELECTION:

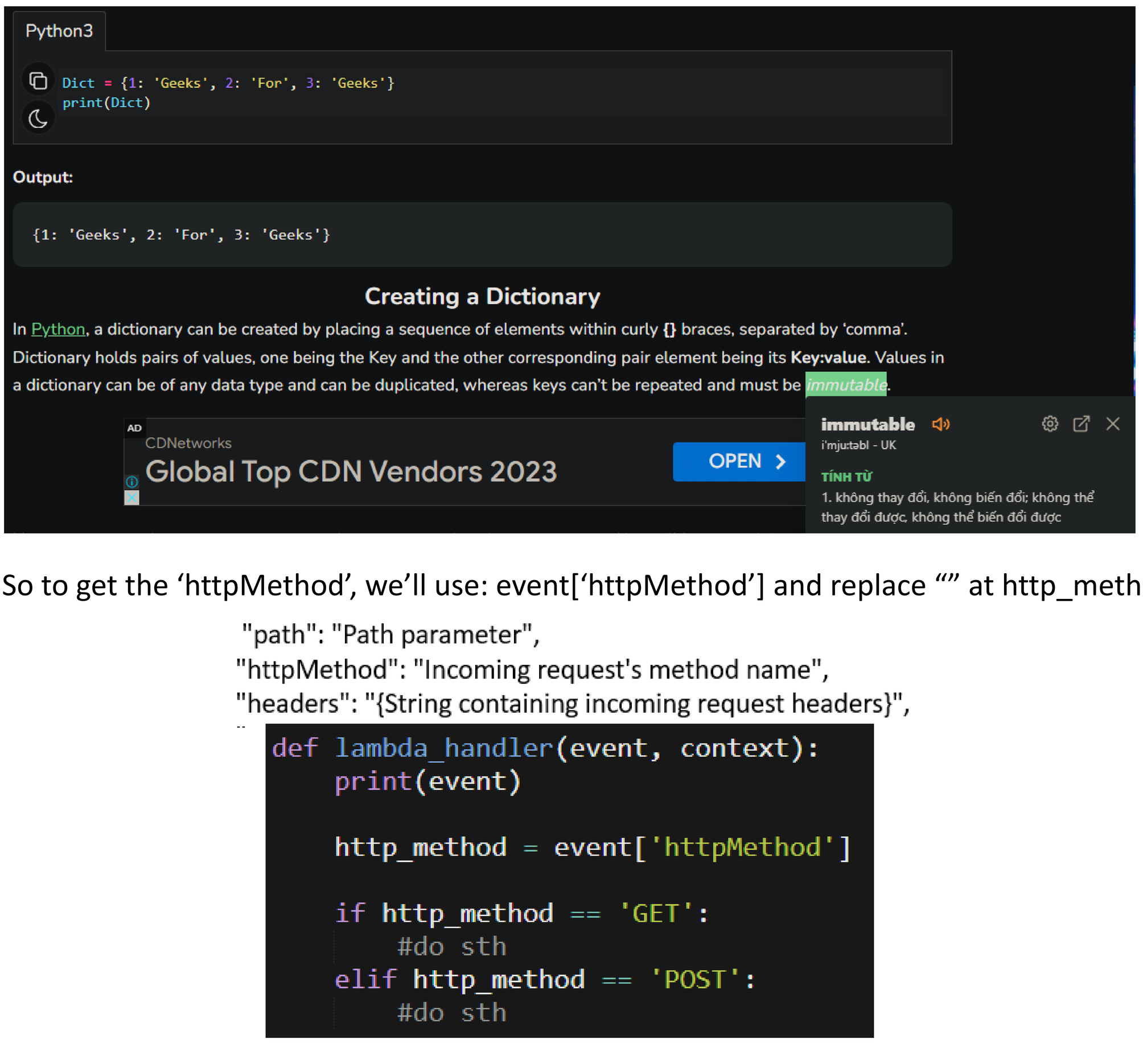
++++ The selection is best of the HTTP method, so we’re going to create a variable HTTP method with an empty value:

```python
def lambda_handler(event, context):
    print(event)

    http_method = ""
```

++++ Then we’re going to do an if statement (best of this http_method variable) to provide the different processing for the two methods we currently support:

```python
def lambda_handler(event, context):
    print(event)

    http_method = ""

    if http_method == 'GET':
        #do sth
    elif http_method == 'POST':
        #do sth
```

++++ The event parameter comes in as a dictionary, so we have to use the bracket notation ‘[]’ to access the elements.



Dictionary in python

++++ So to get the ‘httpMethod’, we’ll use: event[‘httpMethod’] and replace “” at http_method:

"path": "Path parameter",
"httpMethod": "Incoming request's method name",
"headers": "{String containing incoming request headers}",

```python
def lambda_handler(event, context):
    print(event)

    http_method = event['httpMethod']

    if http_method == 'GET':
        #do sth
    elif http_method == 'POST':
        #do sth
```

++++ Inside the GET block, we're going to get the query string parameters and assign them to our variables. We do that by using the bracket notation:

```
    "queryStringParameters":
    {
        "first_num":"10",
        "second_num":"10",
        "operator":"add"
    }
```

```
if http_method == 'GET':
    first_num = event['queryStringParameters']['first_num']
```

++++ The same for second_num and operator:

```
if http_method == 'GET':
    first_num = event['queryStringParameters']['first_num']
    second_num = event['queryStringParameters']['second_num']
    operator = event['queryStringParameters']['operator']
```

++++ Inside the POST block, we're going to **get the request body which is a string and convert it to a dictionary**, so that we can access the elements and assign them to our variables. We do that by using the json.loads() method:

The **json.loads()** method can be used to **parse a valid JSON string** and convert it into a Python Dictionary. It is mainly used for deserializing native string, byte, or byte array which consists of JSON data into Python Dictionary.

```
elif http_method == 'POST':
    body = json.loads(event['body'])
```

++++ Then I'll just access the parameters in the body just like I did for the GET request:

```
"body": "{\"first_num\":10,\"second_num\":10,\"operator\":\"add\"}"
```

```
elif http_method == 'POST':
    body = json.loads(event['body'])

    first_num = body['first_num']
    second_num = body['second_num']
    operator = body['operator']
```

++++ Now we need to create a function which does the calculation: (**NOTE: WE HAVE TO CONVERT THEM THEM TO INTEGERS, BECAUSE 'first_num' AND 'second_num' ARE STRING TYPE**)

```
def calculate(first_num, second_num, operator):
    result = 0
    if(operator == 'add'):
        result = int(first_num) + int(second_num)
    if(operator == 'subtract'):
        result = int(first_num) - int(second_num)
    if(operator == 'multiply'):
        result = int(first_num) * int(second_num)
    if(operator == 'divide'):
        if(second_num != 0):
            result = int(first_num) / int(second_num)

    return result
```

++++ Finally, add function in the 'body' part and click on "Deploy", full code is below:

```
import json

first_num = 0
second_num = 0
operator = "add"

def lambda_handler(event, context):
    print(event)

    http_method = event['httpMethod']

    if http_method == 'GET':
        first_num = event['queryStringParameters']['first_num']
        second_num = event['queryStringParameters']['second_num']
        operator = event['queryStringParameters']['operator']

elif http_method == 'POST':
    body = json.loads(event['body'])

    first_num = body['first_num']
    second_num = body['second_num']
    operator = body['operator']

return {
    'statusCode': 200,
    'body': json.dumps(calculate(first_num, second_num, operator))
}


def calculate(first_num, second_num, operator):
    result = 0
    if(operator == 'add'):
        result = int(first_num) + int(second_num)
    if(operator == 'subtract'):
        result = int(first_num) - int(second_num)
    if(operator == 'multiply'):
        result = int(first_num) * int(second_num)
    if(operator == 'divide'):
        if(second_num != 0):
            result = int(first_num) / int(second_num)

return result
```

++++ Test:

Execution results — Status: Succeeded | Max memory used: 37 MB | Time: 1.37 ms

Test Event Name
GetRequestEvent

Response
```
{
  "statusCode": 200,
  "body": "20"
}
```

Function Logs
```
START RequestId: 0a626eae-d307-42ad-a28b-e42eabf23a06 Version: $LATEST
{'resource': '/calculate', 'path': '/calculate', 'httpMethod': 'GET', 'queryStringParameters': {'first_num': '10', 'second_num': '10', 'operator': 'add'}}
END RequestId: 0a626eae-d307-42ad-a28b-e42eabf23a06
REPORT RequestId: 0a626eae-d307-42ad-a28b-e42eabf23a06  Duration: 1.37 ms   Billed Duration: 2 ms   Memory Size: 128 MB Max Memory Used: 37 MB
```

Request ID
0a626eae-d307-42ad-a28b-e42eabf23a06

Execution results — Status: Succeeded | Max memory used: 37 MB | Time: 1.05 ms

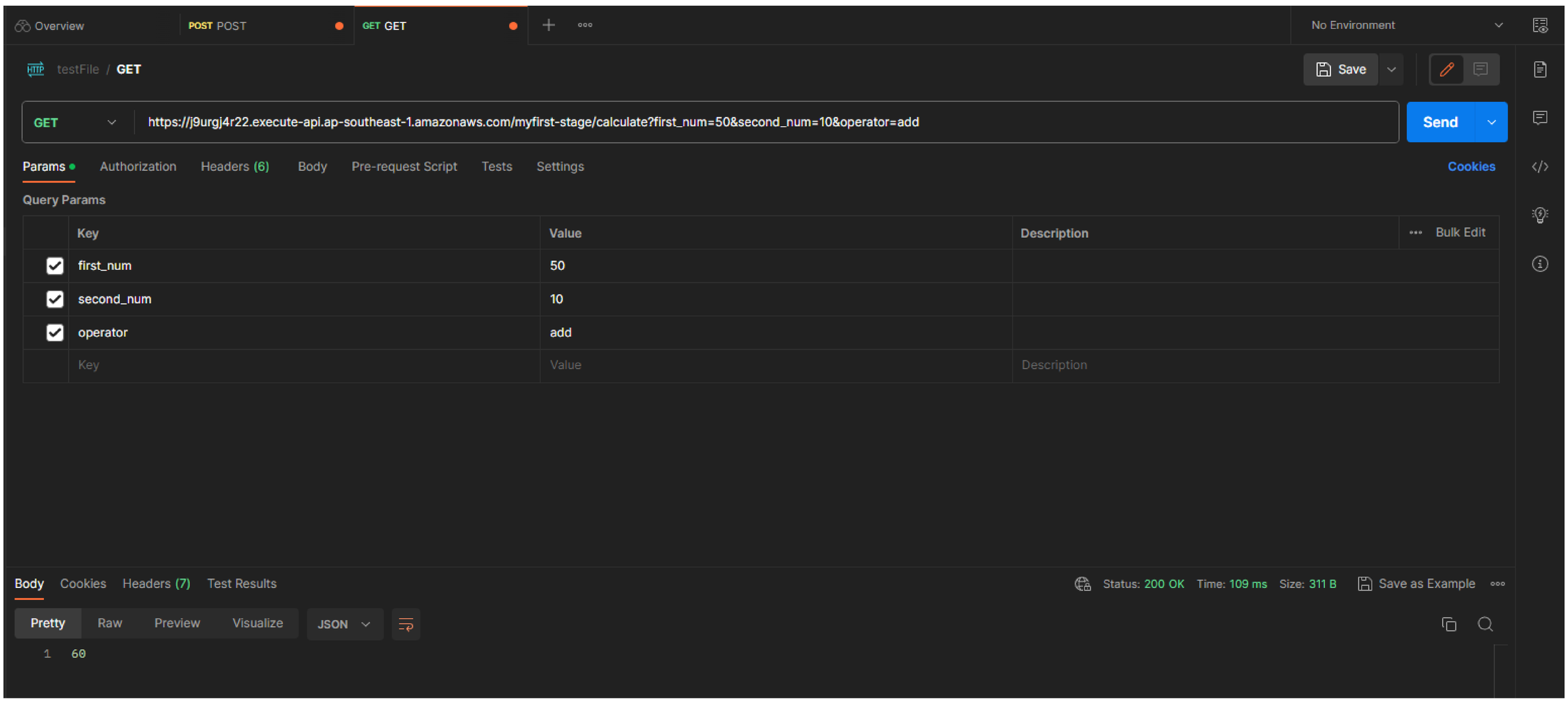
Test Event Name
PostRequestEvent

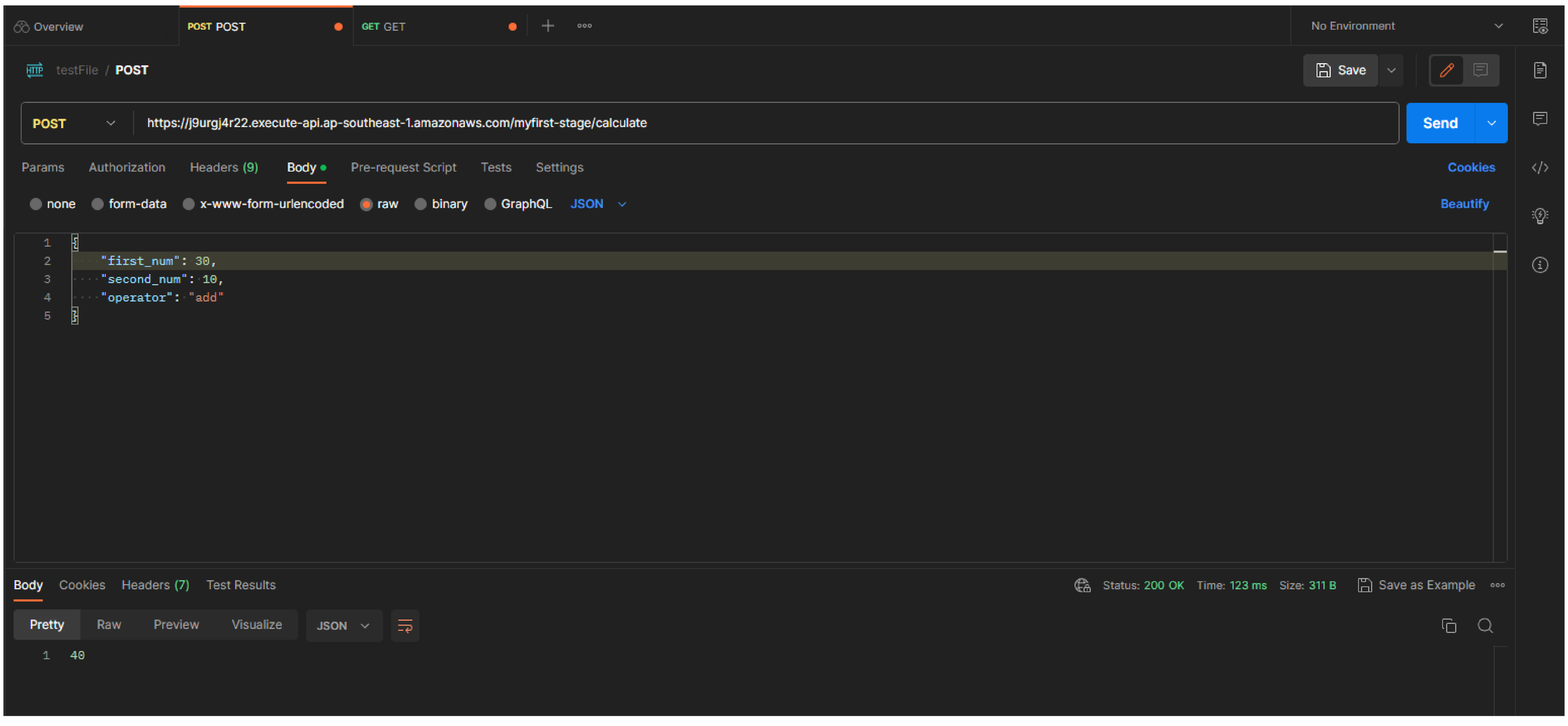
Response
```
{
  "statusCode": 200,
  "body": "20"
}
```

Function Logs
```
START RequestId: e7da73f3-9e5f-4151-a828-f32552e85b3e Version: $LATEST
{'resource': '/calculate', 'path': '/calculate', 'httpMethod': 'POST', 'body': '{"first_num":10,"second_num":10,"operator":"add"}'}
END RequestId: e7da73f3-9e5f-4151-a828-f32552e85b3e
REPORT RequestId: e7da73f3-9e5f-4151-a828-f32552e85b3e  Duration: 1.05 ms   Billed Duration: 2 ms   Memory Size: 128 MB Max Memory Used: 37 MB
```

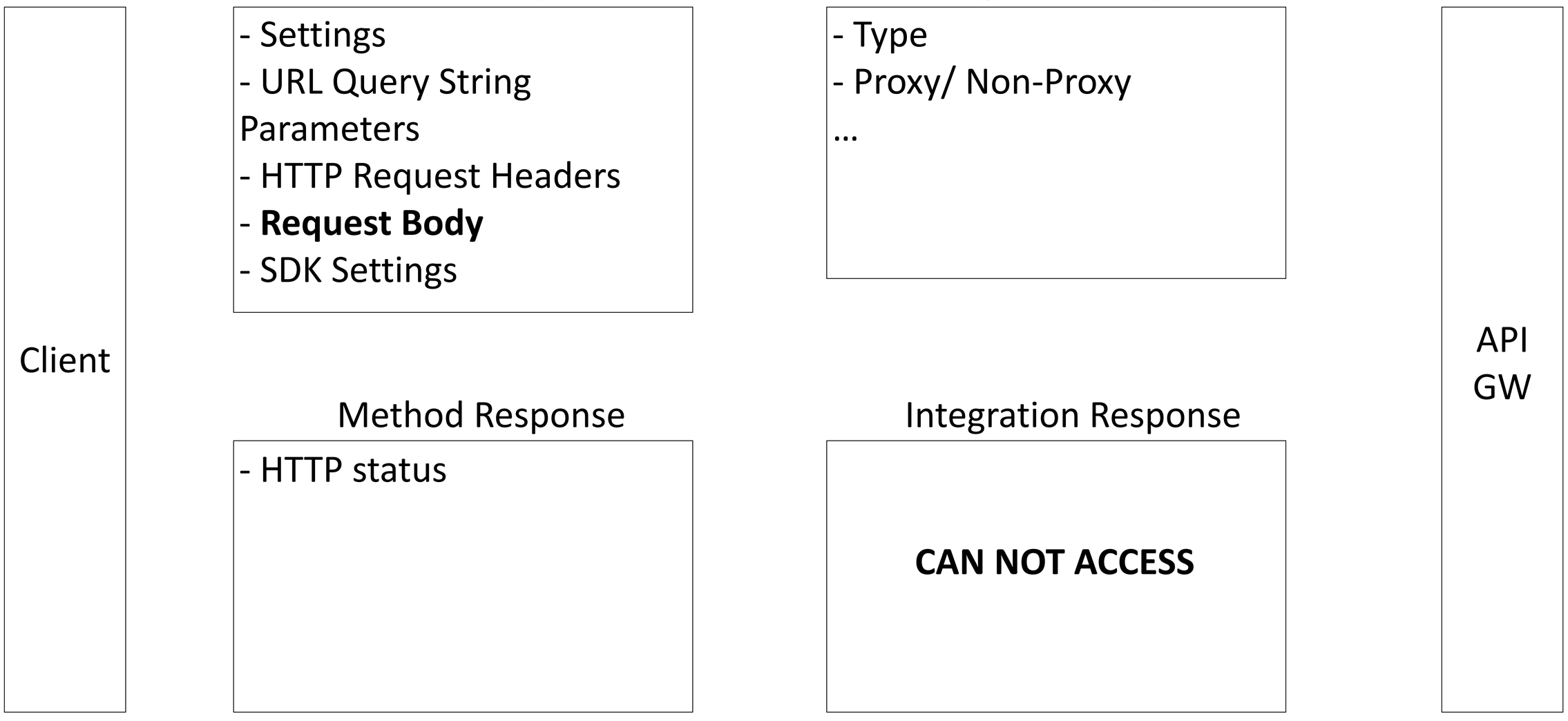
Request ID
e7da73f3-9e5f-4151-a828-f32552e85b3e

++ Now we can test using Postman:





Use Lambda Proxy integration



Khi bạn bật proxy (Use Lambda Proxy integration) trong API Gateway, quá trình xử lý yêu cầu và phản hồi sẽ đơn giản hơn vì API Gateway sẽ chuyển đổi yêu cầu và phản hồi giữa API Gateway và Lambda function dưới dạng JSON, không cần sử dụng mô tả bản mẫu (Mapping Templates). Dưới đây là các bước cơ bản API sẽ thực hiện khi bật proxy:

1. Yêu cầu (Request) từ Client: Khi một yêu cầu HTTP được gửi từ máy khách (client) đến API Gateway, API Gateway sẽ nhận yêu cầu này.

2. Gọi Lambda Function: API Gateway sẽ gọi Lambda function và chuyển toàn bộ thông tin từ yêu cầu của API Gateway tới Lambda function trong định dạng JSON.

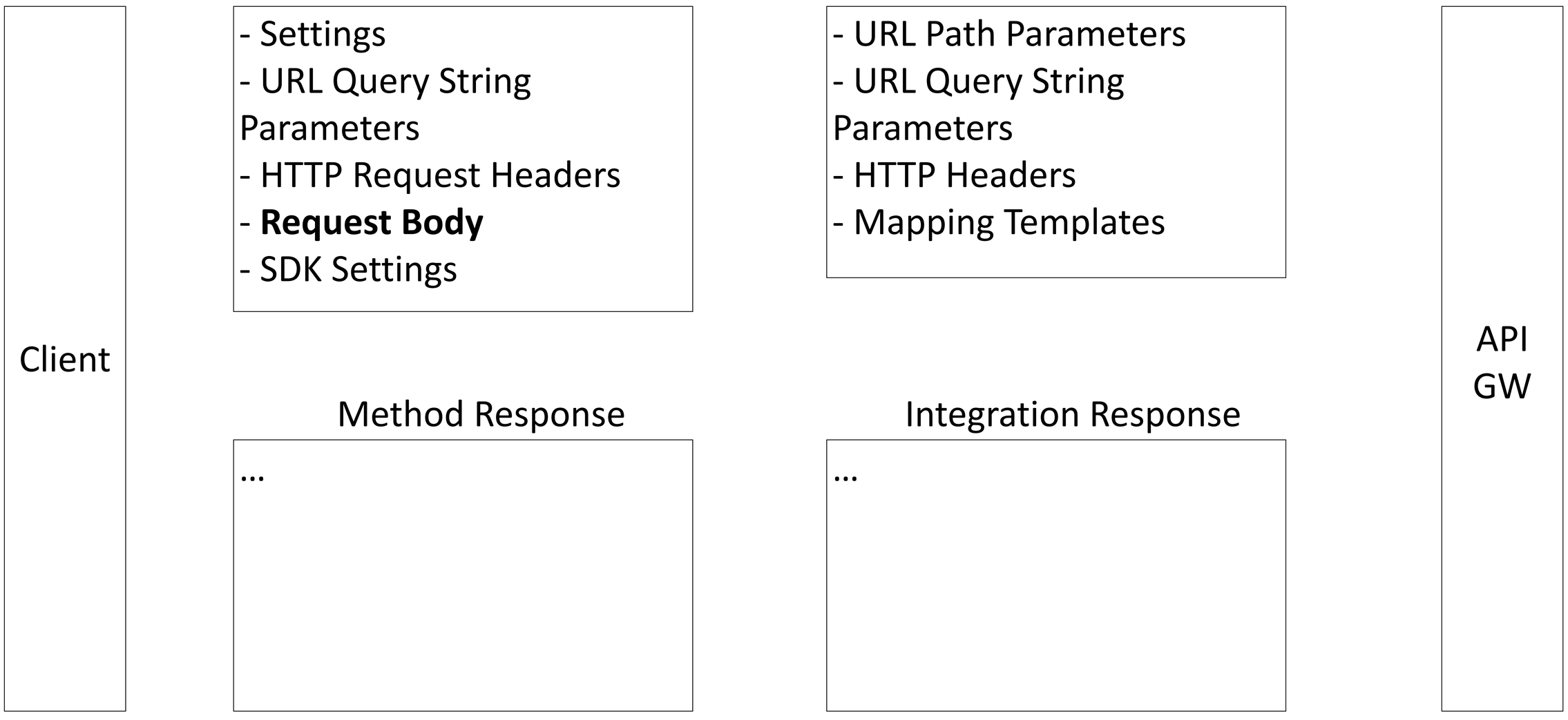
3. Xử lý trong Lambda Function: Lambda function sẽ nhận yêu cầu dưới dạng đối tượng JSON và xử lý yêu cầu dựa trên các thông tin như phương thức HTTP, tiêu đề (headers), tham số truy vấn (query parameters) và phần thân (body) của yêu cầu.

4. Phản hồi (Response) từ Lambda Function: Sau khi Lambda function hoàn thành xử lý, nó sẽ trả về một phản hồi (response) dưới dạng đối tượng JSON.

5 .Trả về Phản hồi cho Client: API Gateway sẽ chuyển phản hồi từ Lambda function về dưới dạng JSON và trả về phản hồi này cho máy khách (client).

Lợi ích của việc sử dụng tích hợp proxy Lambda là giảm bớt việc xử lý và cấu hình bởi API Gateway, hỗ trợ truyền thông tin một cách trực tiếp giữa API Gateway và Lambda function. Nó đơn giản hóa cấu hình và giúp tối ưu hóa hiệu suất của hệ thống. Tuy nhiên, điều quan trọng là bạn phải đảm bảo Lambda function của bạn có thể xử lý yêu cầu và phản hồi trong định dạng JSON.

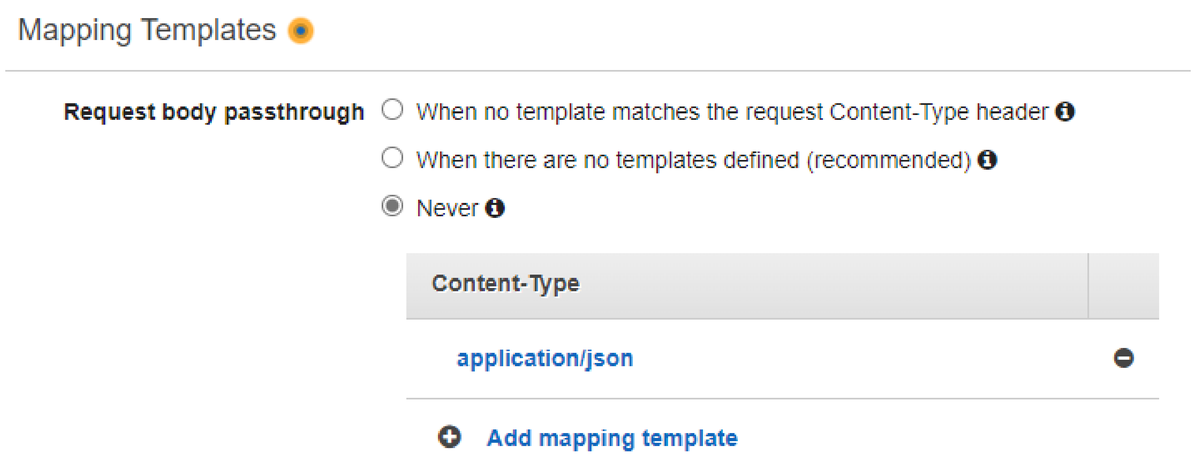
DON’T Use Lambda Proxy integration



Khi bạn không sử dụng tích hợp proxy Lambda, API Gateway sẽ thực hiện các bước sau để xử lý yêu cầu và phản hồi:

1. Yêu cầu (Request) từ Client: Khi một yêu cầu HTTP được gửi từ máy khách (client) đến API Gateway, API Gateway sẽ nhận yêu cầu này.

2. Bộ điều hướng (Request Mapper): API Gateway sử dụng mô tả bản mẫu (mapping template) để chuyển đổi định dạng yêu cầu từ định dạng của API Gateway thành định dạng phù hợp với đầu vào của Lambda function. Bộ điều hướng này cũng cho phép bạn trích xuất thông tin từ yêu cầu (ví dụ: query parameters, headers, body) và truyền nó vào Lambda function.



3. Gọi Lambda Function: Sau khi yêu cầu đã được chuyển đổi, API Gateway sẽ gọi Lambda function và truyền các thông tin đã chuyển đổi từ bước trước làm tham số đầu vào cho Lambda.

4. Xử lý trong Lambda Function: Lambda function sẽ xử lý yêu cầu dựa trên thông tin nhận được và thực hiện các tác vụ cụ thể.

5. Phản hồi (Response) từ Lambda Function: Sau khi Lambda function hoàn thành xử lý, nó sẽ trả về một phản hồi, đóng gói trong định dạng JSON hoặc một định dạng phù hợp khác.

6. Bộ điều hướng (Response Mapper): API Gateway sử dụng mô tả bản mẫu để chuyển đổi định dạng phản hồi từ định dạng của Lambda function thành định dạng phản hồi của API Gateway.

7. Phản hồi cho Client: Sau khi phản hồi đã được chuyển đổi, API Gateway trả về phản hồi đó cho máy khách (client).

Lợi ích của việc sử dụng mô tả bản mẫu là bạn có thể linh hoạt điều chỉnh định dạng yêu cầu và phản hồi giữa API Gateway và Lambda function. Nó cho phép bạn tùy chỉnh xử lý yêu cầu và phản hồi theo ý muốn, và hỗ trợ nhiều định dạng dữ liệu khác nhau nếu cần thiết.