Lab 5: Serverless

Servers are fun until they are not. You are running a news agency which has high peaks of traffic but they happen sporadically (infrequently). It'll be more cost effective to set up a REST API using Lambda to access and perform CRUD on tables in your noSQL DynamoDB database. Lambdas are charged only when they are working unlike EC2 instances which are charged for always, as long as they are running. This way with Lambda, your company will pay only for peaks and in other times when there's 0 traffic, it won't be charged at all! Stales news get almost no traffic.

Also, your last IT Ops person is leaving to work for Google. Company can't hire a replacement. Lambda will require almost no maintenance since they are managed app environment. All the patches, security and scaling is taken care off by AWS experts!

You'll build a REST API for all the tables not just one. As an example, you'll be using and working with messages but clients can work with any table by sending a different query or payload. Later, you'll be able to create auth and validate request and response in API Gateway (not covered in this lab). You are going to use three services:

- DynamoDB
- Lambda
- API Gateway

Task

Create a lambda CRUD microservice to save data in DB

Walk-through

If you would like to attempt the task, then skip the walk-through and go for the task directly. However, if you need a little bit more hand holding or you would like to look up some of the commands or code or settings, then follow the walk-through.

1. Create DynamoDB table
2. Create IAM role to access DynamoDB
3. Create AWS Lambda
4. Create API Gateway
5. Test
6. Clean up

1. Create DynamoDB table

The name of the table in these examples is `messages`. Feel free to modify it in the command options as you wish. The key name is `id` and the type is string (`S`)

```
aws dynamodb create-table --table-name messages \
  --attribute-definitions AttributeName=id,AttributeType=S \
  --key-schema AttributeName=id,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

You'll get back the Arn identifier:

```
{
    "TableDescription": {
        "TableArn": "arn:aws:dynamodb:us-west-1:161599702702:table/messages",
        "AttributeDefinitions": [
            {
                "AttributeName": "id",
                "AttributeType": "N"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "TableName": "messages",
        "TableStatus": "CREATING",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "id"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": 1493219395.933
    }
}
```


You can also get this info by

```
aws dynamodb describe-table --table-name messages
```

You can get the list of all tables in your selected region (aws configure):

```
aws dynamodb list-tables
```

2. Create IAM role to access DynamoDB

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "lambda.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Let's create an IAM role so our lambda can access DynamoDB. First, create a role with a trust policy from a file:

```
aws iam create-role --role-name LambdaServiceRole --assume-role-policy-document file:/
/lambda-trust-policy.json
```

If you are curious, the file has the lambda service identifier:

```
{
    "Role": {
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": "sts:AssumeRole",
                    "Principal": {
                        "Service": [
                            "lambda.amazonaws.com"
                        ]
                    },
                    "Effect": "Allow",
                    "Sid": ""
                }
            ]
        },
        "RoleId": "AROAJLHUFSSSWHS5XKZOQ",
        "CreateDate": "2017-04-26T15:22:41.432Z",
        "RoleName": "LambdaServiceRole",
        "Path": "/",
        "Arn": "arn:aws:iam::161599702702:role/LambdaServiceRole"
    }
}
```


Write down the role Arn somewhere.

Next, add the policies so the lambda function can work with the database:

```
aws iam attach-role-policy --role-name LambdaServiceRole --policy-arn arn:aws:iam::aws
:policy/AmazonDynamoDBFullAccess
```

No output is a good thing in this case. 😄

Other optional managed policy which you can use in addition to `AmazonDynamoDBFullAccess` is `AWSLambdaBasicExecutionRole`. It has the logs (CloudWatch) write permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

The commands to attach more managed policies are the same — `attach-role-policy`.

3. Create AWS Lambda

Here's the code for the function (`code/serverless/index.js`). It is very similar to Express request handler. It checks HTTP methods and performs CRUD on DynamoDB table accordingly. Table name comes from query string or from body.

```
'use strict'

console.log('Loading function')
const doc = require('dynamodb-doc')
const dynamo = new doc.DynamoDB()

// All the request info in event
// "handler" is defined on the function creation
exports.handler = (event, context, callback) => {

    // Callback to finish response
    const done = (err, res) => callback(null, {
        statusCode: err ? '400' : '200',
        body: err ? err.message : JSON.stringify(res),
        headers: {
            'Content-Type': 'application/json',
        }
    })
    // To support mock testing, accept object not just strings
    if (typeof event.body == 'string')
        event.body = JSON.parse(event.body)
    switch (event.httpMethod) {
        // Table name and key are in payload
        case 'DELETE':
            dynamo.deleteItem(event.body, done)
            break
        // No payload, just a query string param
        case 'GET':
            dynamo.scan({ TableName: event.queryStringParameters.TableName }, done)
            break
        // Table name and key are in payload
        case 'POST':
            dynamo.putItem(event.body, done)
            break
        // Table name and key are in payload
        case 'PUT':
            dynamo.updateItem(event.body, done)
            break
        default:
            done(new Error(`Unsupported method "${event.httpMethod}"`))
    }
}
```

So either copy or type the code into a file and archive it with ZIP into `db-api.zip`.

Now we can create an AWS Lambda function from the source code. Use your IAM role Arn from the IAM step. The code for the function will come from a zip file. The handle is the name of the method in `index.js` for AWS to import and invoke.

```
aws lambda create-function --function-name db-api \
  --runtime nodejs6.10 --role arn:aws:iam::161599702702:role/LambdaServiceRole \
  --handler index.handler \
  --zip-file fileb://db-api.zip \
  --memory-size 512 \
  --timeout 10
```

Memory size and timeout are optional. By default, they are 128 and 3 correspondingly.

Results will look similar to this but with different IDs of course:

```
{
    "CodeSha256": "bEsDGu7ZUb9td3SA/eYOPCw3GsliT3q+bZsqzcrW7Xg=",
    "FunctionName": "db-api",
    "CodeSize": 778,
    "MemorySize": 512,
    "FunctionArn": "arn:aws:lambda:us-west-1:161599702702:function:db-api",
    "Version": "$LATEST",
    "Role": "arn:aws:iam::161599702702:role/LambdaServiceRole",
    "Timeout": 10,
    "LastModified": "2017-04-26T21:20:11.408+0000",
    "Handler": "index.handler",
    "Runtime": "nodejs6.10",
    "Description": ""
}
```

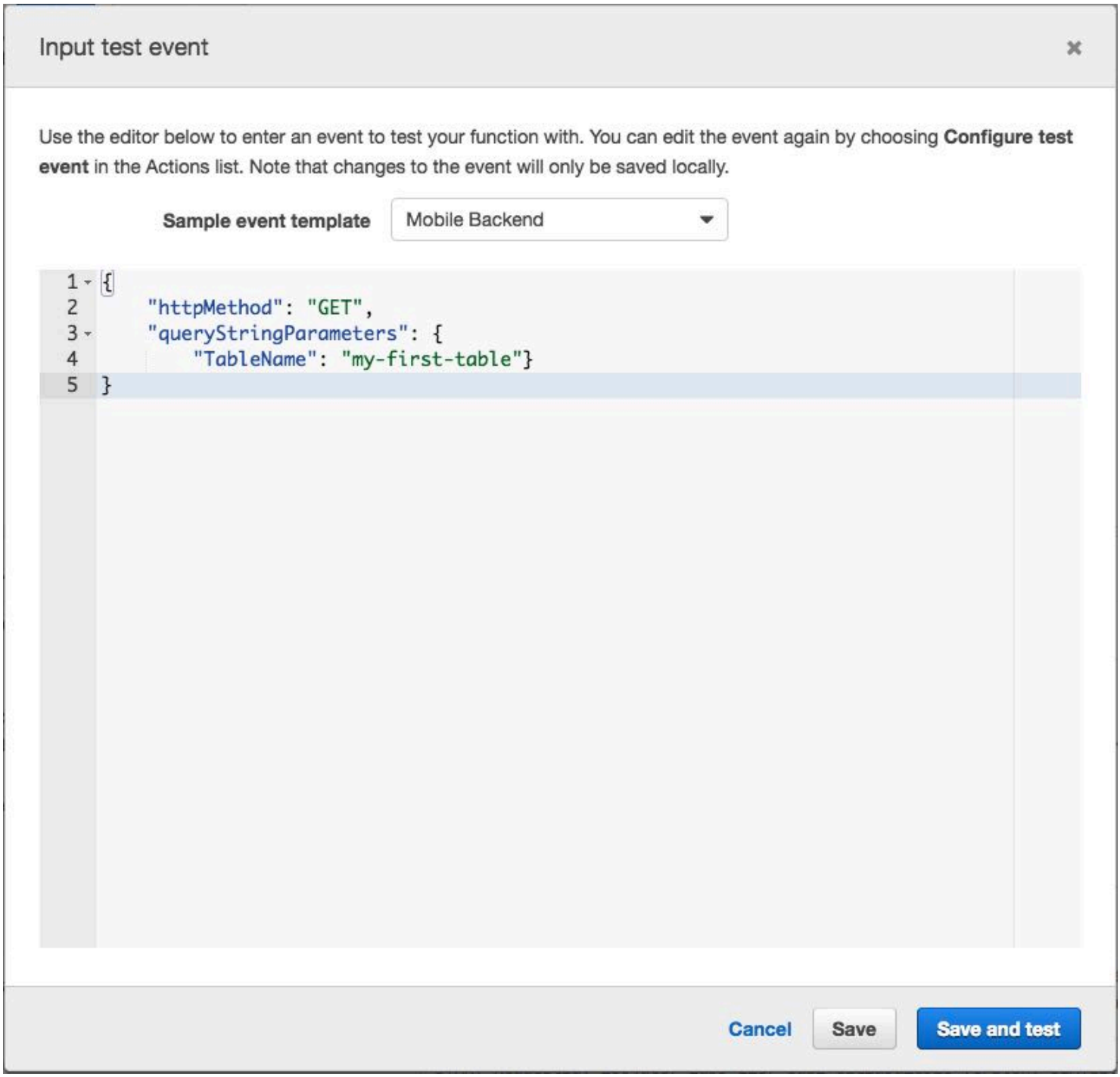
Test function with this data which mocks an HTTP request (`db-api-test.json` file):

```
{
    "httpMethod": "GET",
    "queryStringParameters": {
        "TableName": "messages"
     }
}
```

Run from a CLI (recommended) to execute function in the cloud:

```
aws lambda invoke \
--invocation-type RequestResponse \
--function-name db-api \
--payload file://db-api-test.json \
output.txt
```

Or testing can be done from the web console in Lambda dashboard (blue test button once you navigate to function detailed view):



The results should be 200 (ok status) and output in the `output.txt` file. For example, I do NOT have any record yet so my response is this:

```
{"statusCode":"200","body":"{\"Items\":[],\"Count\":0,\"ScannedCount\":0}","headers":{
"Content-Type":"application/json"}}
```

The function is working and fetching from the database. You can test other HTTP methods by modifying the input. For example, to test creation of an item:

```
{
  "httpMethod": "POST",
  "queryStringParameters": {
    "TableName": "messages"
  },
  "body": {
    "TableName": "messages",
    "Item":{
       "id":"1",
       "author": "Neil Armstrong",
       "text": "That is one small step for (a) man, one giant leap for mankind"
    }
  }
}
```

4. Create API Gateway

You will need to do the following:

1. Create REST API in API Gateway
2. Create a resource (i.e, `/db-api`, e.g., `/users`, `/accounts`)
3. Define HTTP method(s) without auth
4. Define integration to Lambda (proxy)
5. Create deployment
6. Give permissions for API Gateway resource and method to invoke Lambda

The process is not straightforward. Thus, you can use a shell script which will perform all the steps (recommended) or web console.

The shell script is in the `create-api.sh` file. It has inline comments to help you understand what is happening. Feel free to inspect `create-api.sh`. For brevity and to avoid clutter, the file is not copied into this document.

Run this command to create the API endpoint and integrate it with Lambda function (if you modified the region or the function name, you'll need to change those values in script as well):

```
sh create-api.sh
```

In the end, script will make a GET request to check that everything is working. This is an example of running the automation script for the API Gateway (your IDs and Arns will be different):

```
sh create-api.sh
{
    "id": "sdzbvm11w6",
    "name": "api-for-db-api",
    "description": "Api for db-api",
    "createdDate": 1493242759
}
API ID: sdzbvm11w6
Parent resource ID: sdzbvm11w6
{
    "path": "/db-api",
    "pathPart": "db-api",
    "id": "yjc218",
    "parentId": "xgsraybhu2"
}
Resource ID for path db-api: sdzbvm11w6
{
    "apiKeyRequired": false,
    "httpMethod": "ANY",
    "authorizationType": "NONE"
}
Lambda Arn: arn:aws:lambda:us-west-1:161599702702:function:db-api
{
    "httpMethod": "POST",
    "passthroughBehavior": "WHEN_NO_MATCH",
    "cacheKeyParameters": [],
    "type": "AWS_PROXY",
    "uri": "arn:aws:apigateway:us-west-1:lambda:path/2015-03-31/functions/arn:aws:lamb
da:us-west-1:161599702702:function:db-api/invocations",
    "cacheNamespace": "yjc218"
}
{
    "id": "k6jko6",
    "createdDate": 1493242768
}
APIARN: arn:aws:execute-api:us-west-1:161599702702:sdzbvm11w6
{
    "Statement": "{\"Sid\":\"apigateway-db-api-any-proxy-9C30DEF8-A85B-4EBC-BBB0-8D50E
6AB33E2\",\"Resource\":\"arn:aws:lambda:us-west-1:161599702702:function:db-api\",\"Eff
ect\":\"Allow\",\"Principal\":{\"Service\":\"apigateway.amazonaws.com\"},\"Action\":[\
"lambda:InvokeFunction\"],\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:exec
ute-api:us-west-1:161599702702:sdzbvm11w6/*/*/db-api\"}}}"
}
```

```
{
    "responseModels": {},
    "statusCode": "200"
}
Resource URL is https://sdzbvm11w6.execute-api.us-west-1.amazonaws.com/prod/db-api/?Ta
bleName=messages
Testing...
{"Items":[],"Count":0,"ScannedCount":0}%
```

You are all done!

5. Test

You can then manually run tests by getting resource URL and using cURL, Postman or any other HTTP client. For example, my GET looks like this (replace the URL with yours):

```
curl "https://sdzbvm11w6.execute-api.us-west-1.amazonaws.com/prod/db-api/?TableName=me
ssages"
```

But my POST has a body and header with a unique ID:

```
curl "https://sdzbvm11w6.execute-api.us-west-1.amazonaws.com/prod/db-api/?TableName=me
ssages" \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"TableName": "messages",
    "Item": {
      "id": "'$(uuidgen)'",
      "author": "Neil Armstrong",
      "text": "That is one small step for (a) man, one giant leap for mankind"
    }
  }'
```

Here's an option if you don't want to copy paste your endpoint URL. Use env var to store URL and then CURL to it. Execute this once to store the env var `API_URL`:

```
APINAME=api-for-db-api
REGION=us-west-1
NAME=db-api
APIID=$(aws apigateway get-rest-apis --query "items[?name==\`${APINAME}\`].id" --outpu
t text --region ${REGION})
API_URL="https://${APIID}.execute-api.${REGION}.amazonaws.com/prod/db-api/?TableName=m
essages"
```

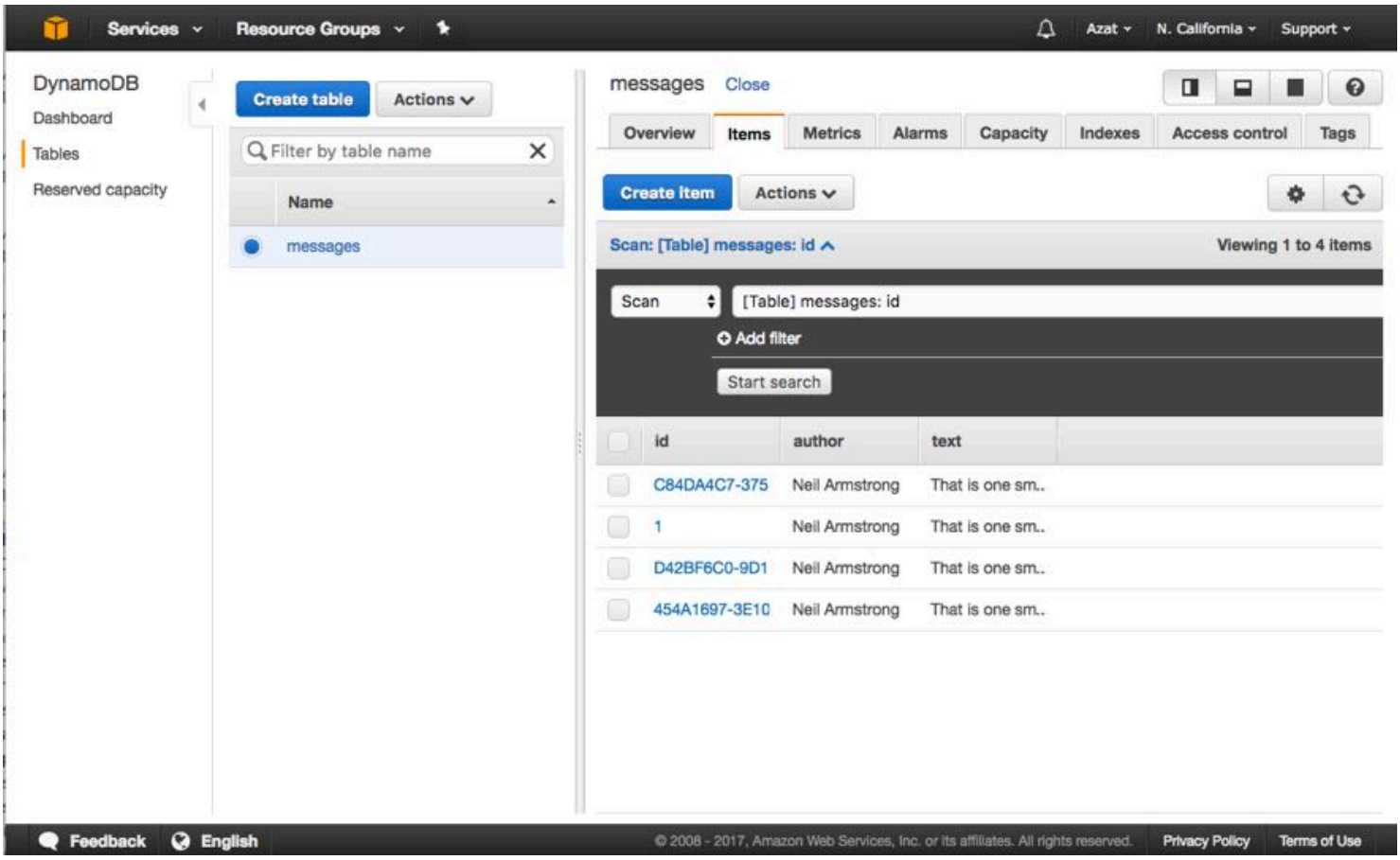
Then run for GET as many times as you want:

```
curl $API_URL
```

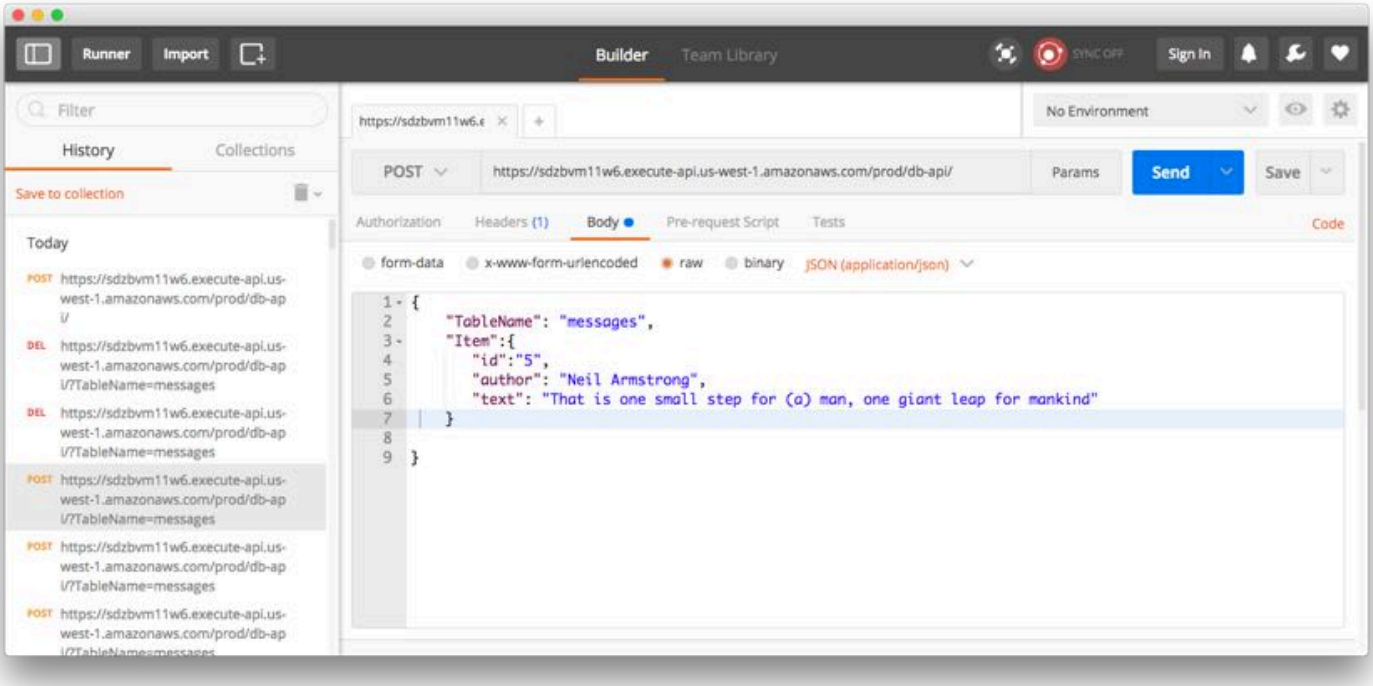
And for POST as many times as you want (thanks to `uuidgen`):

```
curl ${API_URL} \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"TableName": "messages",
    "Item": {
      "id": "'$(uuidgen)'",
      "author": "Neil Armstrong",
      "text": "That is one small step for (a) man, one giant leap for mankind"
    }
  }'
```

The new items can be observed via HTTP interface by making another GET request… or in web console in DynamoDB dashboard as shown below:



Yet another option to play with your new REST API resource. A GUI Postman. Here's how the POST request looks like in Postman. Remember to select POST, Raw and JSON (application/json):



To delete an item with DELETE HTTP request method, the payload must have a `Key`:

```
{
    "TableName": "messages",
    "Key":{
       "id":"8C968E41-E81B-4384-AA72-077EA85FFD04"
    }

}
```

Congratulations! You've built an event-driven REST API for an entire database not just a single table!

Note: For auth, you can set up token-based auth on a resource and method in API Gateway. You can set up response and request rules in the API Gateway as well. Also, everything (API Gateway, Lambda and DynamoDB) can be set up in CloudFormation instead of a CLI or web console (example of Lambda with CloudFormation).

6. Clean up

Remove API Gateway API with `delete-rest-api`. For example here's my command (for yours replace REST API ID accordingly):

```
aws apigateway delete-rest-api --rest-api-id sdzbvm11w6
```

Delete function by its name:

```
aws lambda delete-function --function-name db-api
```

Finally, delete the database too by its name:

```
aws dynamodb delete-table --table-name messages
```

Troubleshooting

- Internal server error: Check your JSON input. DynamoDB requires special format for Table Name and Key.
- Permissions: Check the permission for API resource and method to invoke Lambda. Use test in API Gateway to debug
- `UnexpectedParameter: Unexpected key '0' found in params`: Check that you are sending proper format, JSON vs. string
- `<AccessDeniedException><Message>Unable to determine service/operation name to be authorized</Message></AccessDeniedException>`: Make sure to use POST for integration-http-method as in the create-api script because API Gateway integration can only use POST to trigger functions even for other HTTP methods defined for this resource (like ANY).
- Wrong IDs: Make sure to check names and IDs if you modified the examples.