

## Module 6 Lab 2: DynamoDB Read

### AWS Lambda

The screenshot shows the AWS Lambda Functions page. On the left, there's a sidebar with links for Dashboard, Applications, Functions (which is selected), and Layers. The main area has a header 'Functions (1)' with a search bar. Below it is a table with columns: Function name, Description, Runtime, Code size, and Last modified. A single row is shown for 'myHelloWorldLambda', which was created with Node.js 12.x, has 268 bytes of code, and was last modified 28 minutes ago.

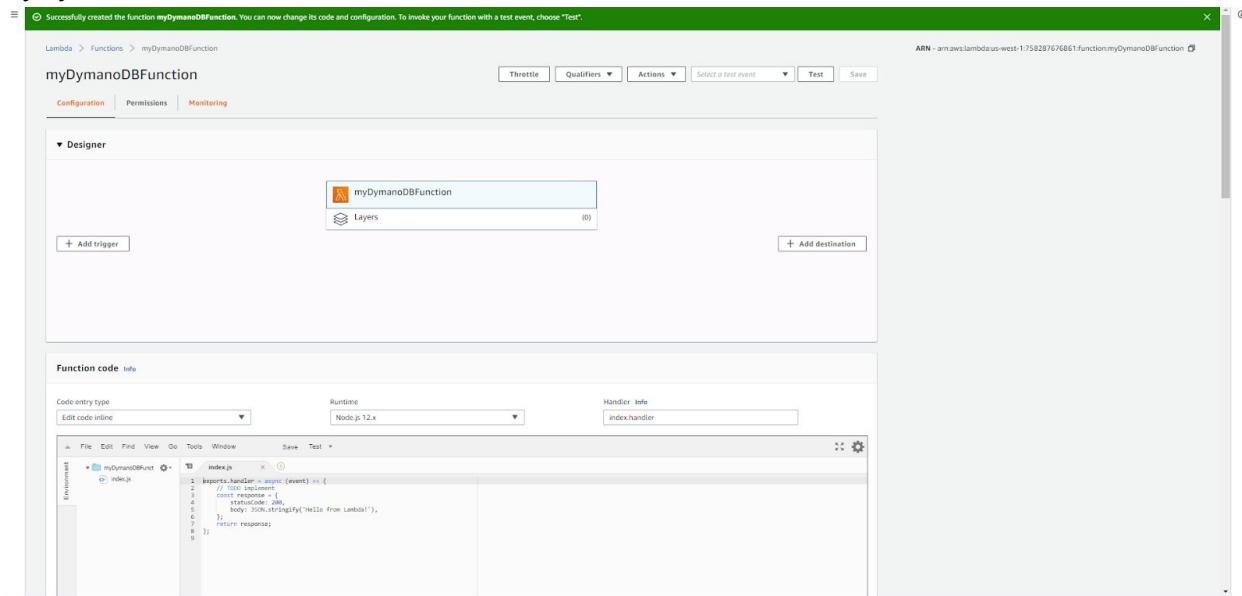
At the main menu for AWS Lambda. In this lab the following AWS services will be used: lambda, DynamoDB, and API Gateway.

### Create function

The screenshot shows the 'Create function' wizard. Step 1: Choose options. It offers three ways to start: 'Author from scratch' (selected), 'Use a blueprint' (Node.js Express app), and 'Browse serverless app repository'. The 'Basic information' section requires entering a function name ('myDynamoDBFunction') and choosing a runtime ('Node.js 12.x'). The 'Permissions' section shows 'Create a new role with basic Lambda permissions' is selected. A note says Lambda will create an execution role with permission to upload logs to Amazon CloudWatch Logs. At the bottom, it says Lambda will create an execution role named myDynamoDBFunction-role-n\$40dyt, with permission to upload logs to Amazon CloudWatch Logs. There are 'Cancel' and 'Create function' buttons at the bottom right.

Name the new DynamoDB function. Note that basic lambda permissions have been selected but an update to the role will occur in the preceding steps.

## myDymanoDBFunction



Successfully created and waiting code. Note that the handler name must match file, index.handler.

## Amazon DynamoDB

Create DynamoDB tables with a few clicks. Just specify the desired read and write throughput for your table, and DynamoDB handles the rest.

Once you have created a DynamoDB table, use the AWS SDKs to write, read, modify, and query items in DynamoDB.

Using the AWS Management Console, you can monitor performance and adjust the throughput of your tables, enabling you to scale seamlessly.

DynamoDB documentation & support  
Getting started guide | FAQ | Developer guide | Forums | Report an issue

A DynamoDB table creation.

## Create DynamoDB table

Create DynamoDB table

DynamoDB is a schema-less database that only requires a table name and primary key. The table's primary key is made up of one or two attributes that uniquely identify items, partition the data, and sort data within each partition.

Table name\* smc

Primary key\* Partition key

sid Number

Add sort key

Table settings

Default settings provide the fastest way to get started with your table. You can modify these default settings now or after your table has been created.

Use default settings

- No secondary indexes
- Provisioned capacity set to 5 reads and 5 writes
- Basic alarms with 80% upper threshold using SNS topic "dynamodb"
- Encryption at Rest with DEFAULT encryption type

You do not have the required role to enable Auto Scaling by default.  
Please refer to documentation.

Add tags NEW

Additional charges may apply if you exceed the AWS Free-Tier levels for CloudWatch or Simple Notification Service. Advanced alarm settings are available in the CloudWatch management console.

Create

Table name, primary key and data type set

## Table smc Overview

smc Close

Overview Items Metrics Alarms Capacity Indexes Global Tables Backups Contributor Insights Triggers Access control Tags

Recent alerts

No CloudWatch alarms have been triggered for this table.

Stream details

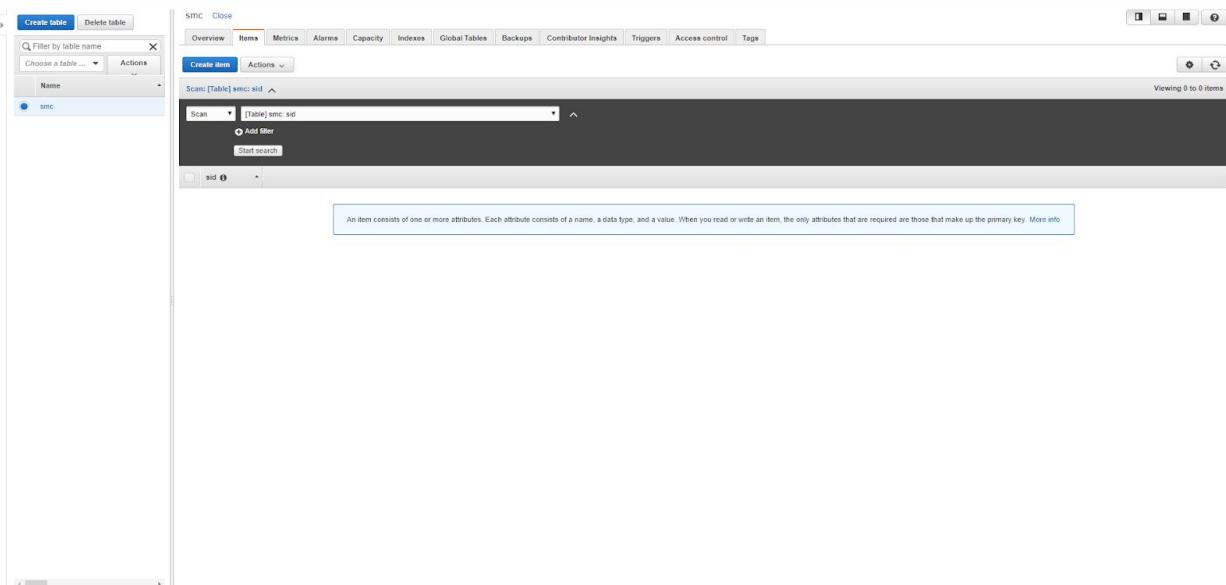
Stream enabled No  
View type -  
Latest stream ARN -  
Manage Stream

Table details

Table name	smc
Primary key	sid (Number)
Primary key type	Partition key
Point-in-time recovery	DISABLED Enable
Encryption type	DEFAULT Manage Encryption
KMS Master Key ARN	Not Applicable
Table status	Active
CloudWatch Contributor Insights	DISABLED Manage Contributor Insights PREVIEW
Time to live attribute	DISABLED Manage TTL
Creation date	March 17, 2020 at 5:07:13 PM UTC-7
Deletion mode	Provisioned
Last change to auto-scaling mode	-
Provisioned read capacity units	5 (Auto Scaling Disabled)
Provisioned write capacity units	5 (Auto Scaling Disabled)
Last decrease time	-
Last increase time	-
Storage size (in bytes)	0 bytes
Item count	0 Manage live count

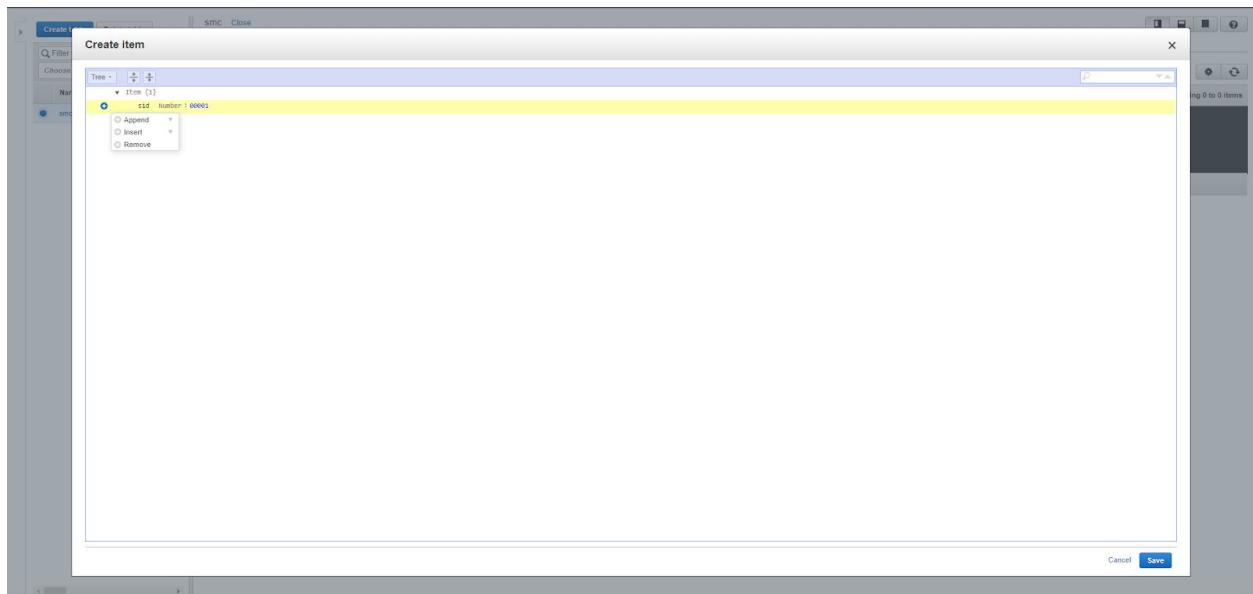
DynamoDB table has been created

## SMC Items



Under the tab “Items” new entries into the database can be made in TREE or JSON format.

### Create item



Here the item will Append to the list as a STRING datatype.

## EmployeeData

The screenshot shows the AWS Lambda console interface. On the left, there's a sidebar with 'Create table' and 'Delete table' buttons. The main area is titled 'EmployeeData' with tabs for Overview, Items, Metrics, Alarms, Capacity, Indexes, Global Tables, Backups, Contributor Insights, Triggers, Access control, and Tags. The 'Items' tab is selected. A search bar at the top says 'Scan: [Table] EmployeeData: eIDNumber'. Below it, a table displays four items:

eIDNumber	fName	dept	eIDName	IName
001	Robert	Sales	Robert Tinkfield	Tinkfield
002	Carl	IT	Carl Zefreable	Zefreable
003	Julio	Payables	Julio Fink	Fink
004	Serg	Outreach	Serg Flemming	Flemming

EmployeeData table created with four employees

## Identity and Access Management (IAM)

The screenshot shows the AWS IAM console. On the left, a sidebar lists 'Access management' (Groups, Users, Roles, Policies, Identity providers, Account settings), 'Access reports' (Access analyzer, Archive rules, Analyzer details), 'Credential report', 'Organization activity', and 'Service control policies (SCPs)'. A search bar at the bottom says 'Search IAM' and an 'AWS account ID:' field are present.

The main area has sections for 'IAM Resources' (Users: 3, Groups: 3, Roles: 24, Identity Providers: 0, Customer Managed Policies: 10) and 'Security Status' (3 out of 5 complete). Under 'Security Status', there are five items with checkboxes:

- Delete your root access keys
- Activate MFA on your root account
- Create individual IAM users
- Use groups to assign permissions
- Apply an IAM password policy

On the right, there's a video player showing a video thumbnail of two people, a progress bar at 0:00 / 2:16, and a 'Additional Information' section with links to 'IAM best practices', 'IAM documentation', 'Web Identity Federation Playground', 'Policy Simulator', and 'Videos, IAM release history and additional resources'.

A new role will be created in IAM

## Create role

Create role

2

Attach permissions policies

Choose one or more policies to attach to your new role.

Create policy

Filter policies ▾ Q Dynamo Showing 9 results

	Policy name ▾	Used as
<input checked="" type="checkbox"/>	AmazonDynamoDBFullAccess	Permissions policy (1)
<input type="checkbox"/>	AmazonDynamoDBFullAccesswithDataPipeline	None
<input type="checkbox"/>	AmazonDynamoDBReadOnlyAccess	None
<input type="checkbox"/>	AWSApplicationAutoscalingDynamoDBTablePolicy	None
<input type="checkbox"/>	AWSLambdaDynamoDBExecutionRole	None
<input type="checkbox"/>	AWSLambdaInvocation-DynamoDB	None
<input type="checkbox"/>	DAXtoDynamoDBPolicy	Permissions policy (1)
<input type="checkbox"/>	DynamoDBCloudWatchContributorInsightsServiceRolePolicy	None

Set permissions boundary

\* Required Cancel Previous Next: Tags

This role will not have DynamoDB access but will have Lambda access

## Create role (tags)

Create role

3

Add tags (optional)

IAM tags are key-value pairs you can add to your role. Tags can include user information, such as an email address, or can be descriptive, such as a job title. You can use the tags to organize, track, or control access for this role. [Learn more](#)

Key	Value (optional)	Remove
staff	jaguirre	x
devTeam	PurpleProject	x
Add new key		

You can add 48 more tags.

Cancel Previous Next: Review

Tags created

## Create role (review)

Role name myDynamoDBFunctionRole

Role description Allows Lambda functions to call AWS services on your behalf.

Trusted entities AWS service: lambda.amazonaws.com

Policies AmazonDynamoDBFullAccess

Permissions boundary Permissions boundary is not set

Key	Value
staff	jaguirre
devTeam	PurpleProject

\* Required      Cancel      Previous      Create role

Role name includes a reference to the AWS service

## Identity and Access Management (IAM)

Permissions policies (1 policy applied)

AmazonDynamoDBFullAccess

Policy summary { JSON }

```
1 * {
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:*",
        "dax:*",
        "application-autoscaling:DeleteScalingPolicy",
        "application-autoscaling:DeregisterScalableTarget",
        "application-autoscaling:DescribeScalableTargets",
        "application-autoscaling:DescribeScalingActivities",
        "application-autoscaling:DescribeScalingPolicies",
        "application-autoscaling:PutScalingPolicy",
        "application-autoscaling:RegisterScalableTarget",
        "cloudwatch:DeleteAlarms",
        "cloudwatch:DescribeAlarmHistory",
        "cloudwatch:DescribeAlarms"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

Simulate policy

Permissions boundary (not set)

Role created with JSON policy summary.

## myDynamoDBFunction

The screenshot shows the AWS Lambda function configuration interface. At the top, there are tabs for Configuration, Permissions, and Monitoring. Below the tabs, the Designer section is open, showing a single trigger named 'myDynamoDBFunction'. A button '+ Add trigger' is available to add more triggers. To the right of the trigger, there is a section for 'Layers' with '(0)' listed. Buttons '+ Add destination' and '+ Add trigger' are also present. At the bottom of the Designer section, there is a toolbar with File, Edit, Find, View, Go, Tools, Window, Save, Test, and a gear icon. In the bottom right corner of the main window, there is a smaller preview window showing the 'Configure test event' dialog.

Test event for the lambda function to be used next

## Configure test event

The screenshot shows the 'Configure test event' dialog box. It has a title bar 'Configure test event' with a close button. Below the title, there is a note: 'A function can have up to 10 test events. The events are persisted so you can switch to another computer or web browser and test your function with the same events.' There are two radio buttons: 'Create new test event' (selected) and 'Edit saved test events'. Under 'Event template', there is a dropdown menu set to 'Hello World'. Under 'Event name', there is a text input field containing 'myDynamoDBFunctionTest'. Below the input field, there is a code editor showing a JSON object: {  
1: {  
2: {  
3: }  
}. On the right side of the dialog, there is a preview window showing the Lambda function configuration page with the 'myDynamoDBFunction' function selected. The preview window includes tabs for Configuration, Permissions, and Monitoring, and shows the Designer section with a single trigger named 'myDynamoDBFunction'. A button '+ Add destination' is visible in the preview window. The bottom right of the dialog contains links for Feedback, English (US), Privacy Policy, and Terms of Use.

Empty to show that the code will run.

## myDynamoDBFunction

The screenshot shows the AWS Lambda function execution interface. At the top, there's a navigation bar with tabs for Throttle, Qualifiers, Actions, myDynamoDBFunction..., Test, and Save. Below the navigation bar, a message says "Execution result: succeeded (logs)". Underneath, a "Details" section is expanded, showing the log output: { "statusCode": 200, "body": "\nHello from Lambda!\n" }. A note below the logs says, "The area below shows the result returned by your function execution. Learn more about returning results from your function."

The hello world code works, but now a DynamoDB test is required.

## myDynamoDBFuncion -Scan

The screenshot shows the AWS Lambda function configuration interface. At the top, there's a navigation bar with tabs for Throttle, Qualifiers, Actions, myDynamoDBFunction..., Test, and Save. Below the navigation bar, a "Function code" tab is selected. The code editor shows the following JavaScript code in index.js:

```
1 console.log("starting lambda function");
2 const AWS = require('aws-sdk');
3 const docClient = new AWS.DynamoDB.DocumentClient({region: 'us-west-1'});
4
5 exports.handler = (event, context, callback) => {
6     var params = {
7         TableName: 'EmployeeData',
8         Key: {
9             eIDNumber: '002'
10         }
11     }
12
13     docClient.get(params, function(err, data){
14         if(err){
15             callback(err,null);
16         } else{
17             callback(null, data);
18         }
19     });
20 };
21
22
23 }
```

On the right side of the code editor, under "Handler info", it says "index.handler". Below the code editor, there's an "Execution Result" panel. It shows the "Execution results" section with the response: { "statusCode": 200, "body": "\nHello from Lambda!\n" }. The status is listed as "Succeeded" with "Max Memory Used: 70 MB" and "Time: 20.02 ms".

This code snippet will return EmployeeData where eIDNumber is equal to '002'. Saved then Tested. However the Execution role is not updated, therefore no return from the database should occur.

## myDynamnoDBFunction table GETITEM fail

The screenshot shows the AWS Lambda function configuration page for 'myDynamnoDBFunction'. In the 'Execution role' section, it says 'Execution result: failed (logs)'. Below this, a log entry is shown: { "errorType": "AccessDeniedException", ... }. The 'Basic settings' section shows a memory of 128 MB and a timeout of 0 min 3 sec.

As expected, the function has no access to the table and returns an Execution failure.

## myDynamoDBFunction

The screenshot shows the AWS Lambda function configuration page for 'myDynamoDBFunction'. In the 'Execution role' section, the 'Existing role' dropdown is set to 'myDynamoDBFunctionRole'. The 'Basic settings' section shows a memory of 128 MB and a timeout of 0 min 3 sec. Other sections like VPC and AWS X-Ray are also visible.

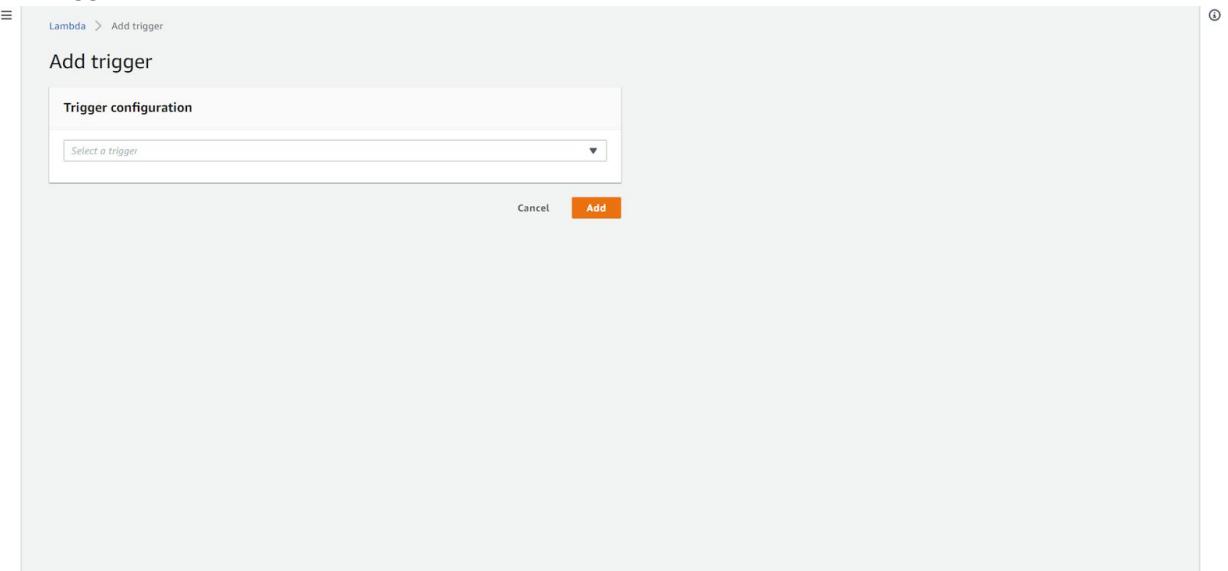
Existing role updated to role created to allow DynamoDB full access.

## myDynamoDBFunction



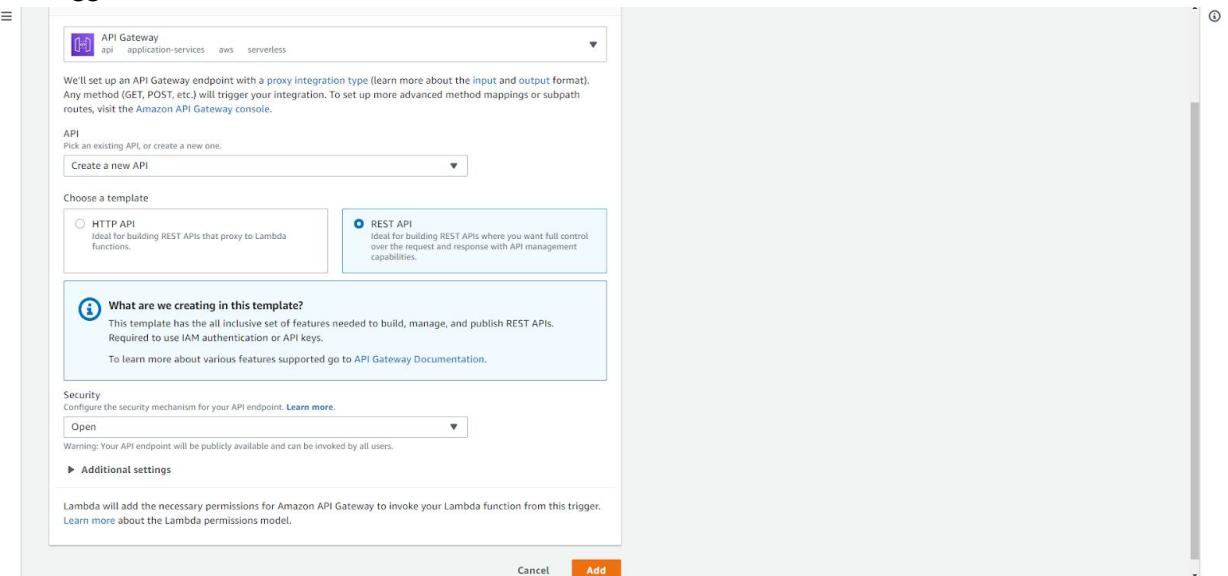
Success! We have a working code base to retrieve data from DynamoDB. Note the fields brought back.

## Add trigger



A new trigger for the lambda read function.

## Add trigger



A new REST API with security set to Open. Choosing Open is NOT recommended for production use.

## myDynamoDBFunction

The trigger myDynamoDBFunction-API was successfully added to function myDynamoDBFunction. The function is now receiving events from the trigger.

Configuration Permissions Monitoring

Designer

myDynamoDBFunction

Layers (0)

+ Add destination

API Gateway

+ Add trigger

API Gateway

myDynamoDBFunction-API

Enabled Delete

Trigger has been added successfully

## Amazon API

API: myDynamoDBFunc...

I Resources

Stages

Authorizers

Gateway Responses

Models

Resource Policy

Documentation

Dashboard

Settings

Usage Plans

API Keys

Client Certificates

Settings

Create Resource

Enable CORS

Edit Resource Documentation

Delete Resource

API ACTIONS

Deploy API

Import API

Edit API Documentation

Delete API

Method Response

HTTP Status: Proxy

Models:

Integration Response

Proxy integrations cannot be configured to transform responses.

LambdaDynamoDBFunction

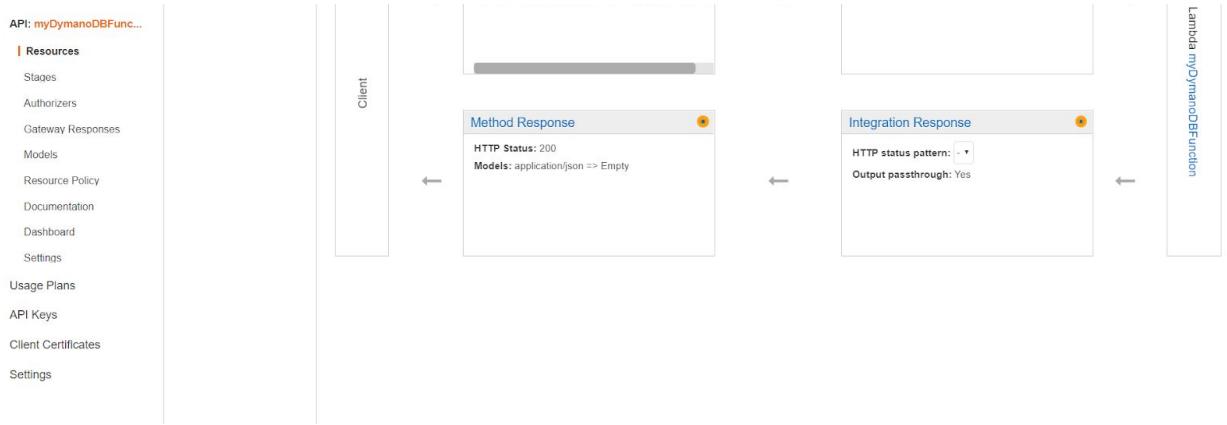
Deleting the ANY method.

## Amazon API Gateway

The screenshot shows the AWS Lambda integration configuration for a GET method of a resource named '/myDynamoDBFunction'. The 'Integration type' is set to 'Lambda Function' with the region 'us-west-1' and the function name 'myDynamoDBFunction'. The 'Use Lambda Proxy integration' checkbox is checked. A 'Save' button is visible at the bottom right.

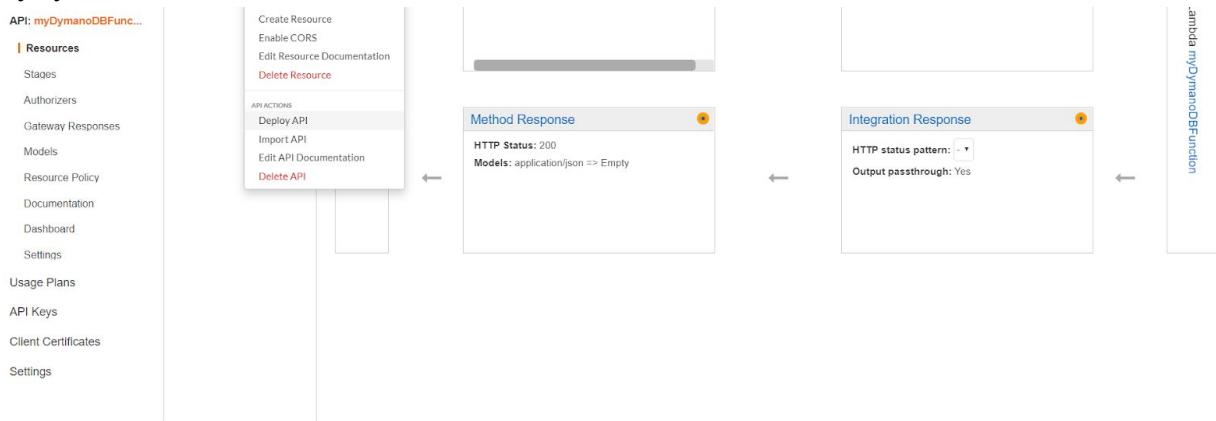
When there's an API call to this gateway, the lambda function will be called.

## /myDynamoDBFunction - GET - Method Execution



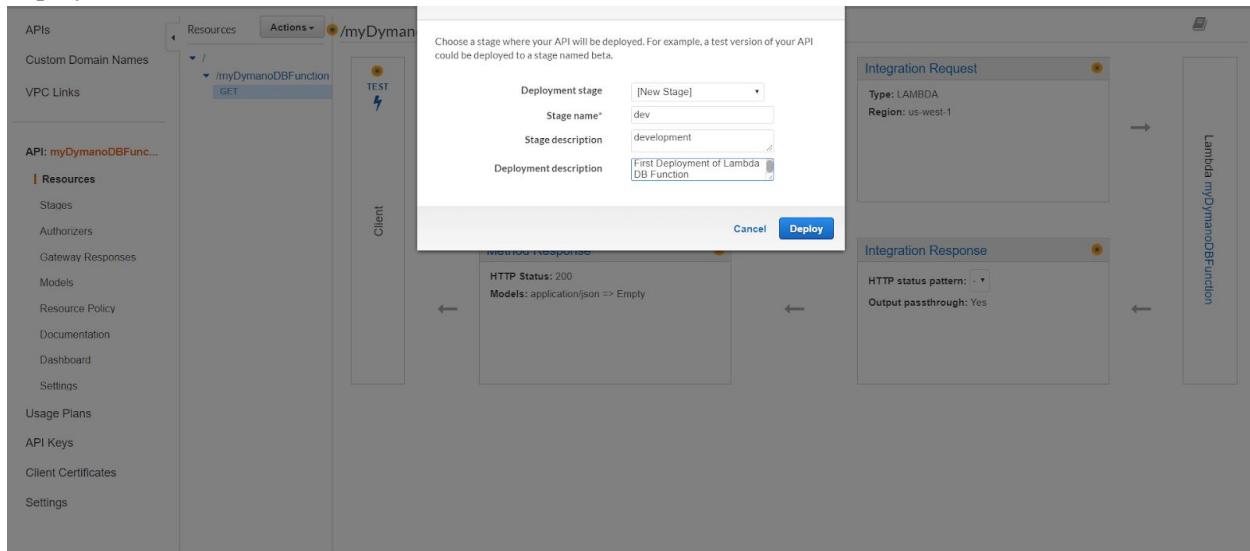
The GET Method Execution is defined.

## /myDynamoDBFunction - GET - Method Execution



With the GET request selected choosing Deploy API

## Deploy API



Deployment ready.

## dev - GET - /myDynamoDBFunction

The screenshot shows the AWS API Gateway interface. On the left, a sidebar lists various API management features like APIs, Custom Domain Names, VPC Links, and others. Under 'APIs', the 'myDynamoDBFunc...' API is selected. In the main area, the 'Stages' tab is active, showing the 'dev' stage under the 'default' stage. A tree view shows a resource path '/myDynamoDBFunction' with a 'GET' method. To the right, a panel titled 'dev - GET - /myDynamoDBFunction' displays the 'Invoke URL' as <https://8q8z1tpi8.execute-api.us-west-1.amazonaws.com/dev/myDynamoDBFunction>. Below it, a note says 'Use this page to override the dev stage settings for the GET to /myDynamoDBFunction method.' There are 'Settings' options to 'Inherit from stage' or 'Override for this method'. At the bottom right is a 'Save Changes' button.

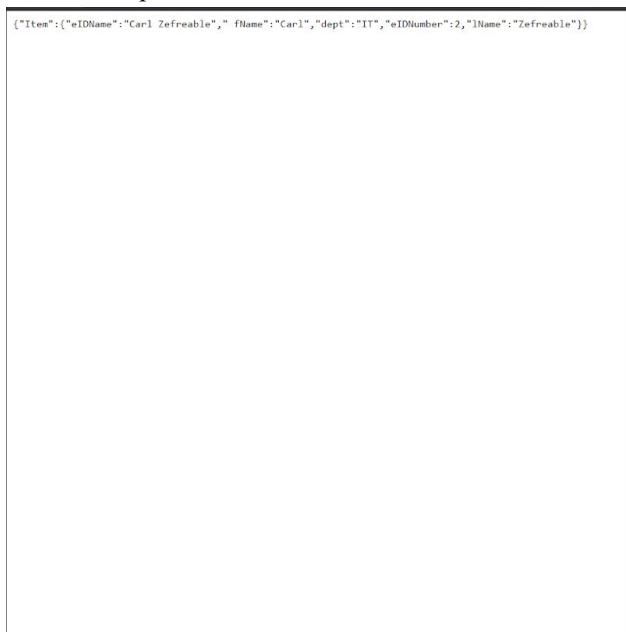
The invoke URL is ready to be tested.

## Invoke URL

A screenshot of a web browser window. The address bar shows the invoke URL: <https://8q8z1tpi8.execute-api.us-west-1.amazonaws.com/dev/myDynamoDBFunction>. The page content displays a JSON response: {"Item": {"eIDName": "Carl Zefreable", "fName": "Carl", "dept": "IT", "eIDNumber": 2, "lName": "Zefreable"}}, indicating that the Lambda function successfully retrieved the item from DynamoDB.

As expected from the previous test, the correct items “eIDNumber:2”are returned.

## Screen comparison of Invoke URL



On the left the items returned with the correct Invoke URL.

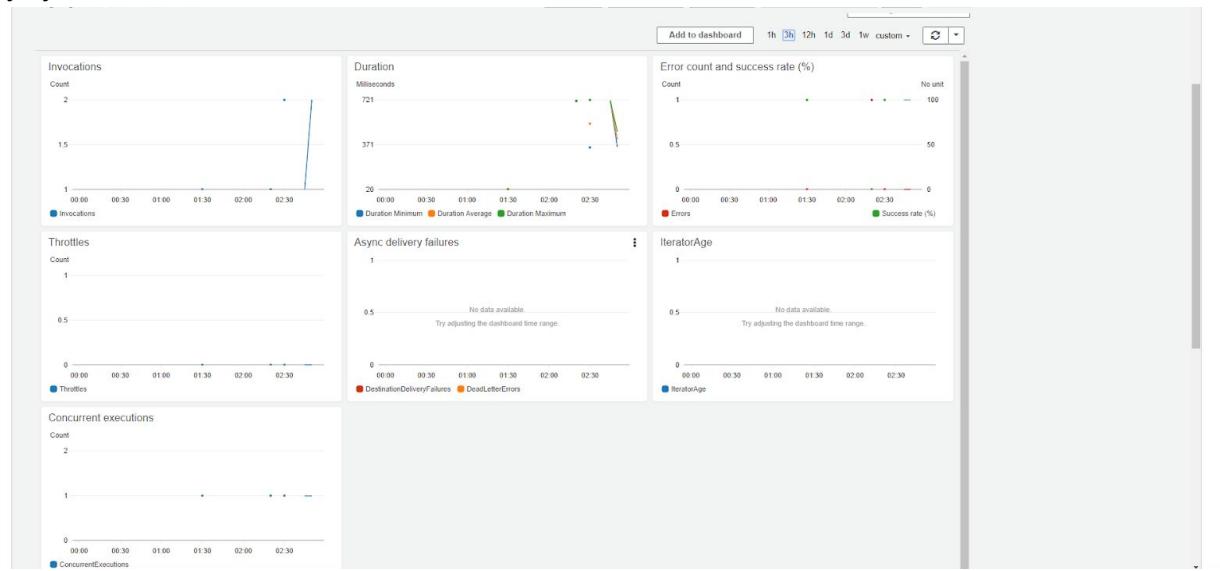
## CloudWatch Log Output

The screenshot shows the AWS CloudWatch Logs Insights interface. On the left, there's a navigation sidebar with links like Dashboards, Alarms, ALARM (with 2 notifications), INSUFFICIENT, OK, Billing, and Logs (which is selected). The main area displays a log entry:

Time (UTC +00:00)	Message
2020-03-18 02:23:48	2020-03-18T02:23:48Z undefined INFO starting lambda function START RequestId: aa106ce5-7ce4-48fc-9c39-93e27408fbcd Version: \$LATEST

The ERROR timestamp references the incorrect Role Permissions.

## myDynamoDBFunction CloudWatch metrics



Various CloudWatch Metrics.

**Function code**

```

Code entry type: Edit code inline
Runtime: Node.js 12.x
Handler: index.handler

Environment:
File Edit Find View Go Tools Window Save Test
index.js
1 // console.log("starting lambda function");
2 const AWS = require('aws-sdk');
3 const docClient = new AWS.DynamoDB.DocumentClient({region: 'us-west-1'});
4
5 exports.handler = (event, context, callback) => {
6   var params = {
7     TableName: "EmployeeData",
8     Key: {
9       "EmployeeID": "882"
10    }
11  }
12  docClient.get(params, function(err, data){
13    if(err) {
14      callback(null);
15    } else{
16      callback(null, data);
17    }
18  });
19 }
20
21
22
23

```

**Execution Result**

```

Status: Succeeded | Max Memory Used: 94 MB | Time: 72.81 ms
Execution results:
{
  "Item": {
    "EmployeeID": "882",
    "EmployeeName": "John Doe"
  }
}

```

If I wanted to return the whole table (EmployeeData), I'd have to change the code I entered previously to reflect my new needs. Will I have to change anything with the API Gateway for it to work?

## myDynamoDBFunction

The screenshot shows the AWS Lambda function editor for 'myDynamoDBFunction'. The top navigation bar includes 'Throttle', 'Qualifiers', 'Actions', 'myDynamoDBFunction...', 'Test', and a 'Save' button. The runtime is set to 'Node.js 12.x' and the handler is 'index.handler'. The left sidebar shows the 'Environment' section with 'myDynamoDBFunc' selected, containing 'index.js' and 'index.json'. The main code editor window displays the following JavaScript code:

```
1 const AWS = require('aws-sdk');
2 const documentClient = new AWS.DynamoDB.DocumentClient();
3
4 exports.handler = async event => {
5   const params = {
6     TableName: "EmployeeData"
7   };
8   try {
9     const data = await documentClient.scan(params).promise();
10    const response = {
11      statusCode: 200,
12      body: JSON.stringify(data.Items)
13    };
14    return response;
15  } catch (e) {
16    return {
17      statusCode: 500
18    }
19  }
}
```

The status bar at the bottom indicates '(48 Bytes) 6:32 JavaScript Spaces: 2'.

A duplicate of the first code snippet was made and named “index.js.1”. Line 5 of the current code notes that all all data items will be returned if successful.

## myDymanoDBFunction Error log

The screenshot shows the AWS Lambda function error log for 'myDymanoDBFunction'. The top navigation bar includes 'Throttle', 'Qualifiers', 'Actions', 'myDymanoDBFunction...', 'Test', and a 'Save' button. The error message is displayed under the 'Execution result: failed (logs)' section:

Execution result: failed (logs)

Details

The area below shows the result returned by your function execution. [Learn more](#) about returning results from your function.

```
[{"errorMessage": "2020-03-18T03:16:33.913Z d3782407-fa96-4e08-b030-3e59379bf2df Task timed out after 3.00 seconds"}]
```

An error occurred. Another go may help.

## myDynamoDBFunction Error log

The screenshot shows the AWS Lambda function configuration page for 'myDynamoDBFunction'. At the top, there are tabs for Throttle, Qualifiers, Actions, myDynamoDBFunction..., Test, and Save. Below these, a scrollable log area displays the following error message:

```
at Object.module.exports.load (/var/runtime/UserFunction.js:140:17)
}
}

Summary
Code SHA-256
Request ID
```

The log ends with a truncated error message: '...'. At the bottom of the log area, there are tabs for Summary, Code SHA-256, and Request ID.

This time a different error log.

The screenshot shows the AWS Lambda function configuration page for 'myDynamoDBFunction'. At the top, there are tabs for Throttle, Qualifiers, Actions, myDynamoDBFunction..., Test, and Save. Below these, the 'Function code' section is open, showing the code editor. The code editor has tabs for index.js and index.js.1. The code in index.js is as follows:

```
const AWS = require("aws-sdk");
const documentClient = new AWS.DynamoDB.DocumentClient();

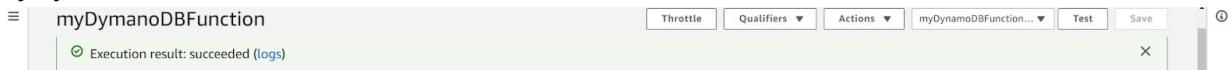
exports.handler = async event => {
  const params = {
    TableName: "EmployeeData"
  };
  try {
    const data = await documentClient.scan(params).promise();
    const response = {
      statusCode: 200,
      body: JSON.stringify(data.Items)
    };
    return response;
  } catch (e) {
    return {
      statusCode: 500
    };
  }
};


```

The code editor also shows the runtime as Node.js 12.x and the handler as index.handler. Below the code editor, the 'Execution Result' section shows the status as Failed, with a max memory used of 30 MB and a duration of 4447.16 ms. The response field is empty.

A misspelling of DynamoDB.

## myDynamoDBFunction



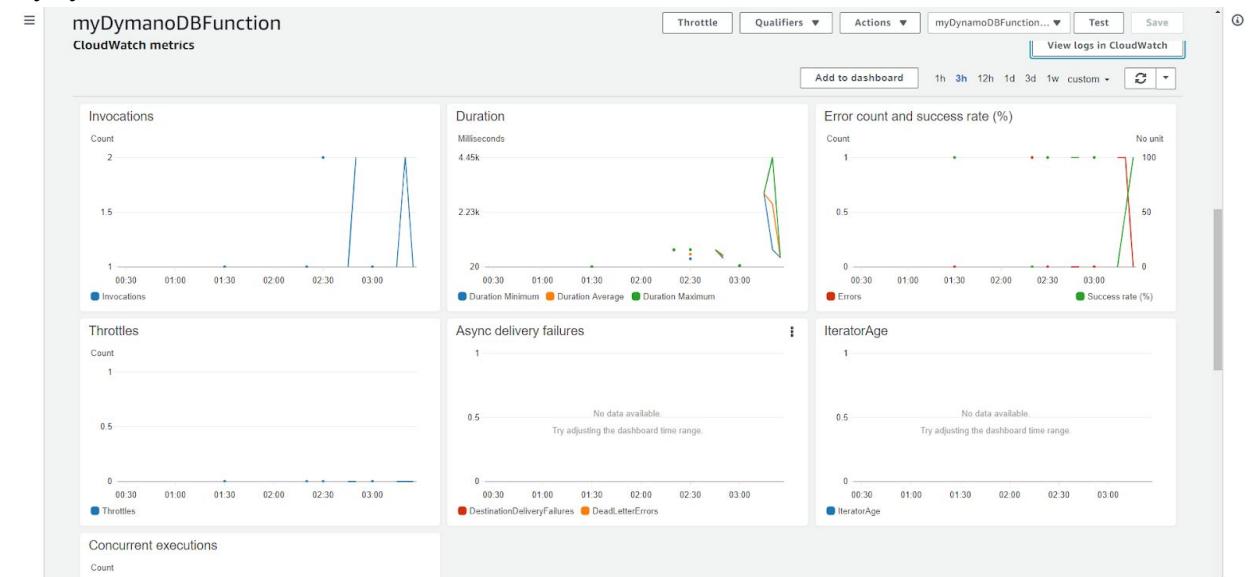
Success! Now to see if the Invoke URL is working and showing a similar output.

## Invoke URL

```
{"statusCode":200,"body":[{"eIDName":"Julio Fink","fName":"Julio","lName":"Fink"}, {"eIDName":"Carl Zefreable","fName":"Carl","lName":"Zefreable"}, {"eIDName":"Serg Flemming","fName":"Serg","lName":"Flemming"}, {"eIDName":"Robert Tinkfield","fName":"Robert","lName":"Tinkfield"}]}
```

Success! All the entries are returned without any changes to the Amazon API Gateway.

## myDynamoDBFunction CloudWatch metrics



The increase of Invocations reference the updated codebase.

## myDynamoDBFunction

The code editor shows the `index.js` file with the following content:

```

1  console.log("starting lambda function");
2
3  const AWS = require('aws-sdk');
4  const docClient = new AWS.DynamoDB.DocumentClient({region: 'us-west-1'});
5
6  exports.handler = (event, context, callback) => {
7
8    var params = {
9      Item: {
10        IDNumber: 005,
11        fName: "Peter",
12        lName: "Rabbit",
13        dept: "Marketing",
14        eName: "Peter Rabbit"
15      },
16      TableName: 'EmployeeData'
17    };
18
19    docClient.put(params, function (err, data){
20      if(err){
21        callback(err, null);
22      } else {
23        callback(null, data);
24      }
25    });
26
27  }
28
29

```

Next, inserting an entry is tested. But before that another duplication named “index.js.2”. The code displayed here reflects the new requirement to add the new employee “Peter Rabbit”.

The screenshot shows the AWS Lambda function execution details page for 'myDynamoDBFunction'. At the top, there are tabs for Throttle, Qualifiers, Actions, and Save. The Actions tab is selected. Below the tabs, it says 'Execution result: succeeded (logs)'. A 'Details' link is present. A note states: 'The area below shows the result returned by your function execution. Learn more about returning results from your function.' There is a JSON placeholder '{ }' and a 'Request ID' field.

The function seems to have worked. DynamoDB may contain the new entry.

The screenshot shows the AWS DynamoDB console for the 'EmployeeData' table. The table has columns: eIDNumber, fName, dept, eIDName, lName, and fName. A scan operation is shown with results for 5 items:

eIDNumber	fName	dept	eIDName	lName	fName
2	Carl	IT	Carl Zefreable	Zefreable	
3	Juilo	Payables	Julo Fink	Fink	
5		Marketing	Peter Rabbit	Rabbit	Peter
1	Robert	Sales	Robert Tinkfield	Tinkfield	
4	Serg	Outreach	Serg Flemming	Flemming	

Success! But something unexpected occurred. A duplicate field (fName) was created to hold Peter.

## myDymanoDBFunction

The screenshot shows the AWS Lambda function editor for 'myDymanoDBFunction'. It displays three separate function scripts:

- index.js1:** Contains code to insert a new item into the 'EmployeeData' table with eIDNumber 005, fName 'Peter', lName 'Rabbit', dept 'Marketing', and cIDName 'Peter Rabbit'.
- index.js2:** Contains code to insert a new item into the 'EmployeeData' table with eIDNumber 006, fName 'Peter', lName 'Rabbit', dept 'Marketing', and cIDName 'Peter Rabbit'.
- index.js3:** Contains code to insert a new item into the 'EmployeeData' table with eIDNumber 007, fName 'Peter', lName 'Rabbit', dept 'Marketing', and cIDName 'Peter Rabbit'.

The Handler info is set to 'index.handler' and the Runtime is 'Node.js 12.x'.

There now exists three unique function scripts for each requirement.

## Invoke URL

The screenshot shows a browser window with the URL `8g8z1pl8.execute-api.us-west-1.amazonaws.com/dev/myDymanoDBFunction`. The response body is a JSON object containing two items, each representing an employee record from the 'EmployeeData' table.

```
{"statusCode":200,"body":[{"\eIDName":"Juilo Fink","fName":"Juilo","dept":"Payables","eIDNumber":3,"lName":"Fink"}, {"\eIDName":"Carl Zefreable","fName":"Carl","dept":"IT","eIDNumber":2,"lName":"Zefreable"}]}
```

Switching back to the GETITEMS script, we can see "Peter Rabbit" exists.