

Using AWS Lambda

Overview

This document describes how to use AWS Lambda and is based on the AWS Developer's Guide available online (<https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>)

Lambda Defined

AWS Lambda supports a Serverless computing model by allowing you to run code without provisioning or managing servers. Lambda executes your code on-demand, when needed, and automatically scales to larger workloads. Consumers only pay for the compute time utilized when the code is running.

Lambda is a versatile service that is ideal for the cloud where cost models are efficient and optimized to save money when resources are idle. In addition, significant energy savings are realized since servers are not required.

Popular User Cases

AWS Lambda is often used to respond to other events occurring in the cloud. For example, notifications when an Amazon S3 bucket is modified or a new record is added to a DynamoDB table. Another popular use case is responding to a HTTP request using Amazon API Gateway. We will look closer at this service next week. Finally, AWS SDKs can be used to invoke Lambda code using API calls directly.

Lambda Components

AWS Lambda includes the following components:

- a. Lambda function - The foundation of Lambda comprised of custom code and any dependent libraries.
- b. Event source - An AWS service (e.g. S3, DynamoDB, API Gateway, SNS) that triggers your function and executes its logic.
- c. Downstream resources - An AWS service (e.g. S3, DynamoDB, CloudWatch, API Gateway, SNS, Lambda) that a Lambda function calls once it is triggered.

Lambda Tools

Several AWS tools are available for coding, implementing and deploying Lambda functions. In this course, we will primarily use the Lambda Console. Tools include:

- a. AWS Lambda Console - Design the Lambda application including creating and updating Lambda function code, configuring and selecting event sources and downstream resources and setting permissions the function requires to operate.
- b. AWS CLI - A command-line interface you can use to leverage Lambda's API operations, such as creating functions and aligning event sources.
- c. AWS Serverless Application Model (SAM) - A command-line interface you can use to develop, test, and analyze your serverless applications locally before uploading them to the Lambda runtime.

Hello, Lambda

For a simple first look at Lambda functions, let us use the AWS Lambda Console to create an Hello, Lambda function.

You will use your AWS Educate assigned classroom for this quick exercise. The steps include:

1. After logging in to your AWS Educate Classroom, open your AWS console and Navigate to the Lambda console. Figure 1 shows the Lambda console. Click “Create Function” to continue.

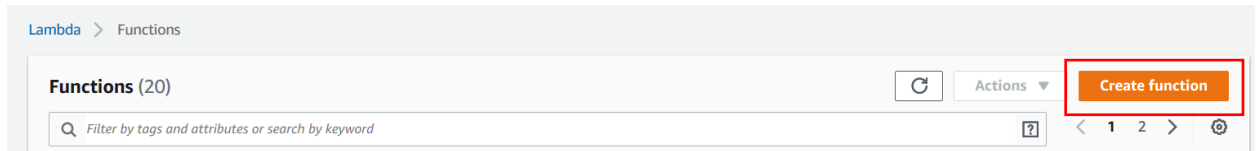


Figure 1 AWS Lambda Console

2. As shown in figure 2, select the “Author From Scratch” option, name the function “hello-lambda” and select the programming language of Python 3.x. Scroll down and Click “Create Function” to continue.

A screenshot of the AWS Lambda 'Create Function' wizard. At the top, there are two options: 'Author from scratch' (selected with a blue circle) and 'Use a blueprint'. Below these is the 'Basic information' section. It contains three fields: 'Function name' with the value 'hello-lambda', 'Runtime' with the value 'Python 3.7', and 'Permissions' with a link to 'Choose or create an execution role'. The 'Function name' field has a note: 'Enter a name that describes the purpose of your function. Use only letters, numbers, hyphens, or underscores with no spaces.' The 'Runtime' field has a note: 'Choose the language to use to write your function.'

Figure 2 Creating a Function

3. Once the function is created you will see a graphical depiction of the Lambda application components. As shown in figure 3, the function has no initial event trigger and a CloudWatch log downstream resource is automatically created for you.

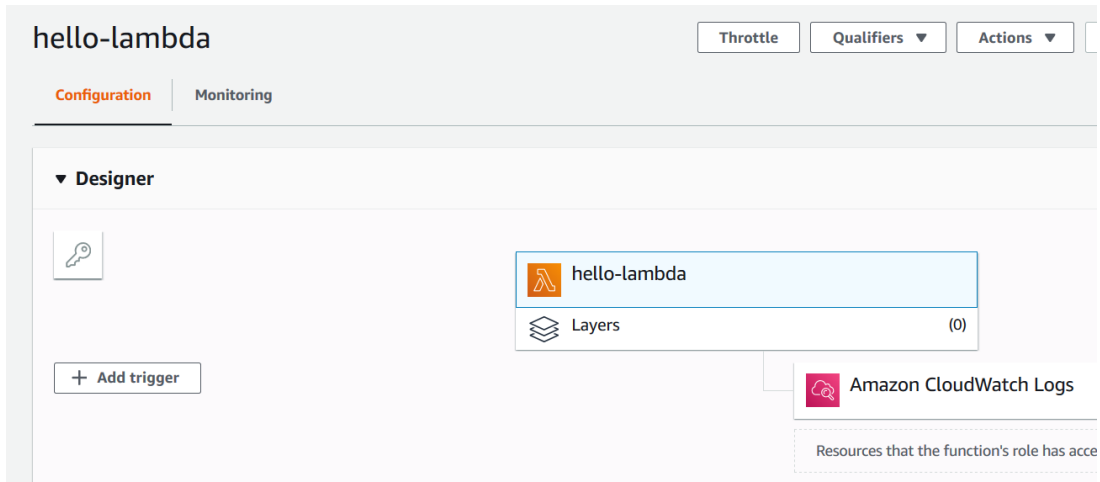


Figure 3 Graphical representation of the Lambda Function

4. Scrolling down further in the Lambda console reveals the starter template code in Python provided. This can be edited to match the specific requirements for your function. We will use the default code as this is essentially a Hello from Lambda function. More details will be provided about the functions and the naming requirements later in this document. For now, realize the Lambda console saves the sample code in the `lambda_function.py` file and in within the code `lambda_handler()` function receives the event as a parameter when the Lambda function is invoked. The event is the values resulting from the trigger.

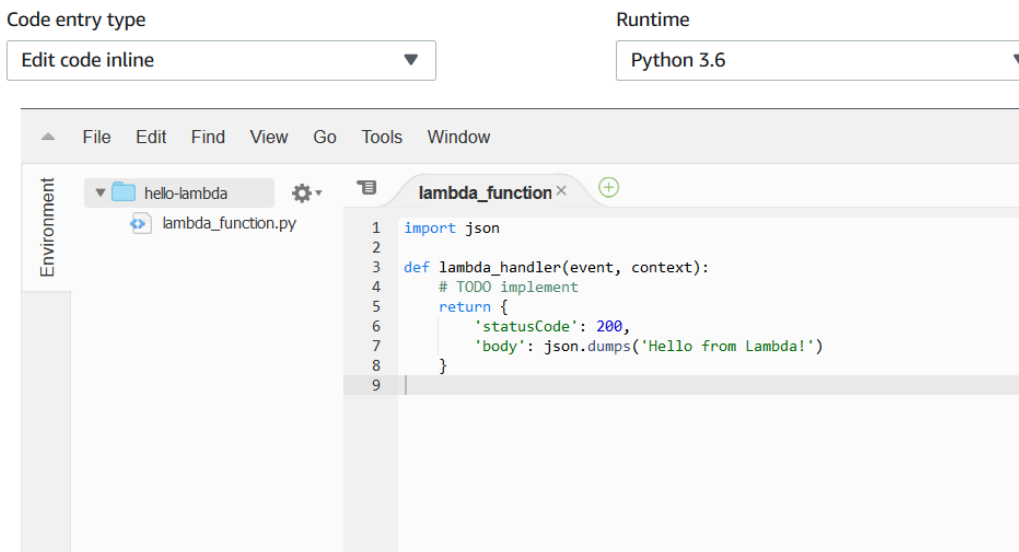


Figure 4 Python Code

5. To test the function, we can use the “Test” functionality included in the Lambda console. A test event is used in lieu of an actual event for triggering. Scroll to the top of the console and select “Test”. Then select “Create new test event”. Use the default Hello World option. Enter an Event name such as TestHello. Scroll down and select “Create” to continue. See figure 5.

Configure test event

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.

☒ Create new test event

☐ Edit saved test events

Event template

Hello World

Event name

TestHello

1 {

2 "key1": "value1",

3 "key2": "value2",

4 "key3": "value3"

5 }

Figure 5 Creating a Test Event

- From the Lambda console, click “Test” to use the TestHello event as shown in figure 6.

Throttle

Qualifiers ▼

Actions ▼

TestHello ▼

Test

Save

Figure 6 Select Test

- Upon successful execution of the Lambda function, you will see a results area that is green. Expanding the results will show the specific output as shown in figure 7.

hello-lambda

Throttle

Qualifiers ▼

✓ Execution result: succeeded (logs)

▼ Details

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

```
{
  "statusCode": 200,
  "body": "\"Hello from Lambda!\""
}
```

Figure 7 Successful Execution of Hello-Lambda

- Clicking on the “Logs” link will provide more details that was sent to the CloudWatch log files. Information such as time and resources used are provided by default. See figure 8.

Time (UTC +00:00)	Message
2019-09-28	
	No older events found at the moment. Retry .
▶ 15:36:56	START RequestId: 64c336ed-2785-4ca7-ae39-074c8bd3f378 Version: \$LATEST
▶ 15:36:56	END RequestId: 64c336ed-2785-4ca7-ae39-074c8bd3f378
▼ 15:36:56	REPORT RequestId: 64c336ed-2785-4ca7-ae39-074c8bd3f378 Duration: 0.73 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory
REPORT RequestId: 64c336ed-2785-4ca7-ae39-074c8bd3f378 Duration: 0.73 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 50 MB Init Duration: 1.05 ms	
	No newer events found at the moment. Retry .

Figure 8 Viewing the CloudWatch Log

Lambda Function Code Details

You can create Lambda functions in multiple languages (Java, C#, Node.js, Go, Python and others). For this course, we will use Python 3.x. The process and primary components needed will be the same for all languages and include:

- **Lambda Function Handler** - AWS Lambda calls this function to start execution of your code. Event data is passed to the handler as the first parameter. The handler should be coded to process the incoming event data. Additional functions can and should be called from the handler in accordance with programming best practices where the Lambda handler should be separated from core logic.
- **Context Object** - AWS Lambda passes a context object as the second parameter of the handler function. The context object provides interact with AWS Lambda context such as remaining execution time before time out, memory limit, function name and others.
- **Logging** - AWS Lambda writes these logs to CloudWatch Logs.
- **Function Errors** - Communicates the result of the function execution to AWS Lambda.

The following code demonstrate how to use the event parameter and the context parameter to build messages, return that message and log context information. The context parameters can be quite useful in determining the size and time needed to execute your Lambda function. Lambda functions should be analyzed for memory and time usage to minimize the cost associated with each call. The longer a Lambda function executes and the more memory it uses, the higher the cost to the user.

```
def lambda_handler(event, context):
    # This takes the event, calls functions

    # Call getMessage
    message = getMessage(event, context)

    # Call the Print to Log function.
    printtoLog(context)

    # Return the Welcome Message
    return {
        'message' : message
    }
```

```
def getMessage(myevent, mycontext):
    nameoffunction = mycontext.function_name
    message = 'Hello {} {} {} from function: {}'.format(
        myevent['title'],
        myevent['first_name'],
        myevent['last_name'],
        nameoffunction)
    return message

def printtoLog(mycontext):
    # The context object can be used as well for time and other
    data.
    print("Log stream name:", mycontext.log_stream_name)
    print("Log group name:", mycontext.log_group_name)
    print("Request ID is:", mycontext.aws_request_id)
    print("Millis :", mycontext.get_remaining_time_in_millis())
```

When reviewing this code, several critical points should be discussed:

1. Parameter passing in Python (and Lambda) work just like any other programming language. Variables can be passed and renamed as needed. Note that `event` and `context` were renamed `myevent` and `mycontext`.
2. Event data is referenced based on key->value pairs coming into the function. In this example events with keys of `title`, `first_name` and `last_name` are used. This directly maps to the TestHello Configure Test event as shown in figure 9.

Configure test event

A function can have up to 10 test events. The events are persisted so you can switch to another and test your function with the same events.

- ☐ Create new test event
☒ Edit saved test events

Saved test event

TestHello ▼

```

1 {
2   "first_name": "Jimmy",
3   "last_name": "Robertson",
4   "title": "Dr."
5 }
```

Figure 9 Test Event Used as Input

- Best programming practices were used where Lambda handler is separated from core logic by providing two additional functions (`getMessage()` and `printtoLog()`) that are called from the handler.
- Upon successful execution, the Hello message is returned in the Lambda Console as shown in figure 10.

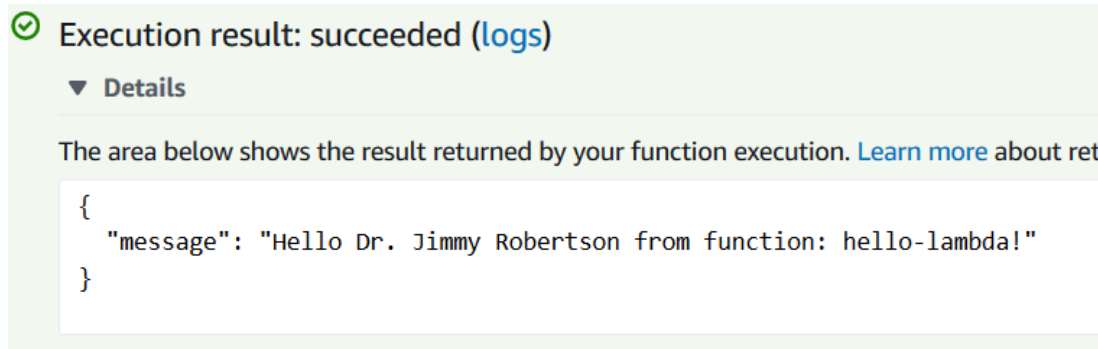


Figure 10 Successful Execution

- The CloudWatch log will provide the results of the print statement. CloudWatch logs are excellent resources for debugging more complex code through the use of strategically placed `print()` statements. (See figure 11.)

Time (UTC +00:00)	Message
2019-09-29	No older events found at the moment. Retry .
17:05:23	START RequestId: f4d1da09-98e6-4412-b2b8-3971378ba237 Version: \$LATEST
17:05:23	Log stream name: 2019/09/29/[\$LATEST]804e2f2e2a743fb5a0b6ca3ec091f7
17:05:23	Log group name: /aws/lambda/hello-lambda
17:05:23	Request ID is: f4d1da09-98e6-4412-b2b8-3971378ba237
17:05:23	Milliseconds remaining: 2999
17:05:23	END RequestId: f4d1da09-98e6-4412-b2b8-3971378ba237
17:05:23	REPORT RequestId: f4d1da09-98e6-4412-b2b8-3971378ba237 Duration: 0.32 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 50 MB Init Duration: 0.00 ms
	No newer events found at the moment. Retry .

Figure 11 CloudWatch Logs for Debug and print() display

Multiple methods and properties exist for the context object including:

- `get_remaining_time_in_millis()` - Returns the number of milliseconds left before the execution times out.
- `function_name` - The name of the function.
- `function_version` - The version of the function.
- `invoked_function_arn` - The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memory_limit_in_mb` - The amount of memory that's allocated for the function.
- `aws_request_id` - The identifier of the invocation request.
- `log_group_name` - The log group for the function.
- `log_stream_name` - The log stream for the function instance.

Watching S3

The examples so far have used the Lambda Console Test Configuration for sample input. Although useful for debugging code, real-world implementation using other Cloud services to trigger the Lambda function. For example, the following example monitors a specific S3 bucket to determine if a specific file has been uploaded.

```
import boto3

def lambda_handler(event, context):
    message = processRecords(event)

    return {
        'message' : message
    }

def processRecords(myevent):
    region=""
    mytime=""
    vers = ""
    filenamed = ""
    foundIt='false'
    # Loop through event Records
    # Assign variable
    for record in myevent['Records']:
        region=record['awsRegion']
        mytime=record['eventTime']
        vers= record['s3']['s3SchemaVersion']
        filenamed = record['s3']['object']['key']

    if filenamed == 'Tester.dat':
        print('Tester.dat was uploaded')
        foundIt='true'

    # Prepare the Message
    message = ' Data: {} {} {} found file is {} '.format(
        filenamed,
        region,
        mytime,
        foundIt
    )

    return message
```

This example monitors an S3 bucket to log when a specific file (Tester.dat) has been uploaded. The test functionality within the Lambda console can still be used to debug the code. In this case, instead of using the helloworld test case, we can use the existing Amazon S3 Put. When creating a new test configuration, use the drop down menu to select Amazon S3 Put. As shown in figure 12, the resulting

test configuration is much more complicated but we can focus in on the items we want to monitor. The object key is where the filename would be stored. We modified this to list the object key as "Tester.dat"

```
14     },
15     "responseElements": {
16       "x-amz-request-id": "EXAMPLE123456789",
17       "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/mnopqrs
18     },
19     "s3": {
20       "s3SchemaVersion": "1.0",
21       "configurationId": "testConfigRule",
22       "bucket": {
23         "name": "example-bucket",
24         "ownerIdentity": {
25           "principalId": "EXAMPLE"
26         },
27         "arn": "arn:aws:s3:::example-bucket"
28       },
29       "object": {
30         "key": "Tester.dat",
31         "size": 1024,
32         "eTag": "0123456789abcdef0123456789abcdef",
33         "sequencer": "0A1B2C3D4E5F678901"
34       }
35     }
36   }
37 }
```

Figure 12 S3 Test Configuration

After selecting the Test button, we can see the logic works as the Tester.dat file was identified as being uploaded. (See figure 13).

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

```
{
  "message": " Data: Tester.dat us-east-1 1970-01-01T00:00:00.000Z found file is true "
}
```

Figure 13 Tester.dat file upload test

You can test the other branch of the logic as well by changing the test configuration. Renaming the key from Tester.dat to Tester1.dat, results in the Lambda function output shown in Figure 14.

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

```
{
  "message": " Data: Tester1.dat us-east-1 1970-01-01T00:00:00.000Z found file is false "
}
```

Figure 14 Testing an additional Data file

Moving this out of the test configuration environment requires configuring a specific trigger event. To do this, click on the add trigger option in the Lambda console for the function. (See figure 15.)

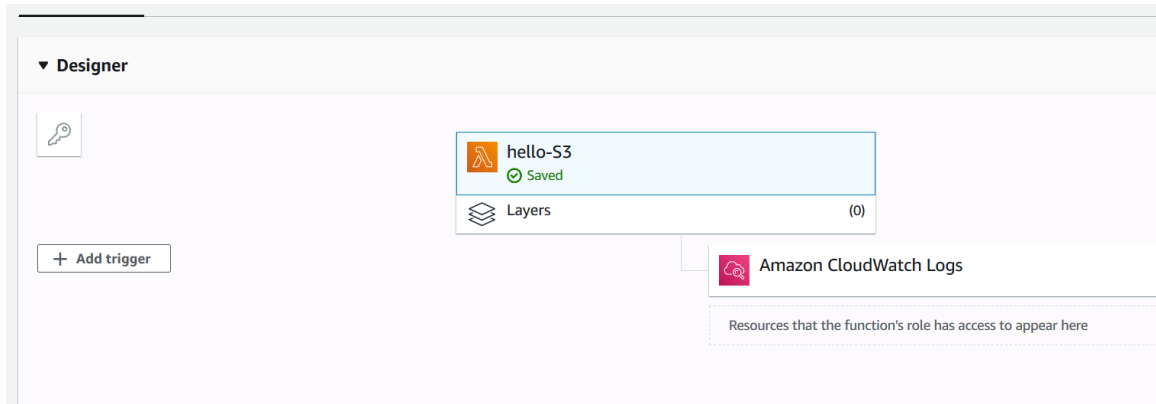



Figure 15 Adding a trigger

As shown in figure 16, using the Lambda console, select the S3 trigger, an existing bucket and the PUT event type. The Prefix and Suffix can be left blank. Scroll down and Click Add to continue.

Trigger configuration

 **S3**
aws storage

Bucket
Please select the S3 bucket that serves as the event source. The bucket must be in the same region as the function.

Event type
Select the events that you want to have trigger the Lambda function. You can optionally set up a prefix or suffix for an event. However, for each bucket, individual events cannot have multiple configurations with overlapping prefixes or suffixes that could match the same object key.

Prefix
Enter a single optional prefix to limit the notifications to objects with keys that start with matching characters.

Suffix
Enter a single optional suffix to limit the notifications to objects with keys that end with matching characters.

Lambda will add the necessary permissions for Amazon S3 to invoke your Lambda function from this trigger. [Learn more](#) about the Lambda permissions model.

☒ **Enable trigger**
Enable the trigger now, or create it in a disabled state for testing (recommended).

Figure 16 Selecting a Put S3 Trigger

This results in the S3 graphical component being added to the graphical display in Lambda console as shown in figure 17.

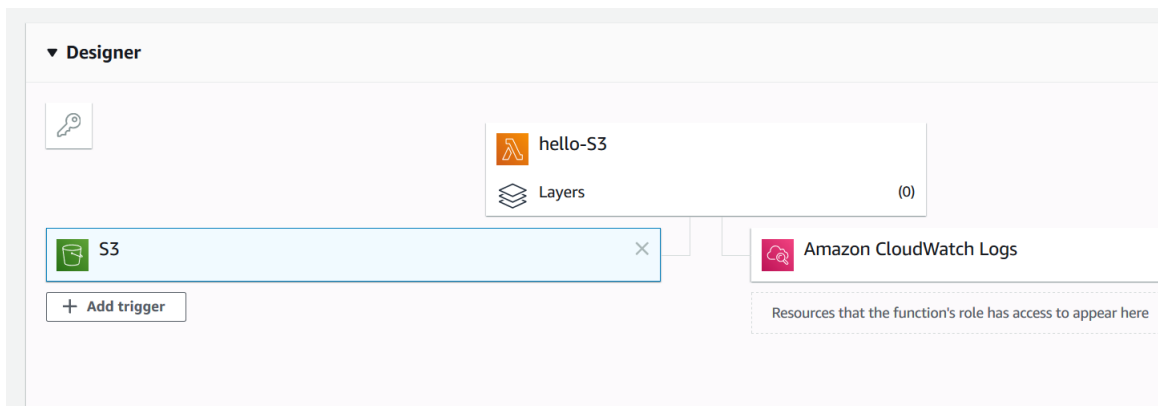


Figure 17 S3 trigger added

With AWS, no permissions are provided by default. To enable permissions for the Lambda function to be able to read the S3 bucket an additional role must be added. Clicking on the Amazon CloudWatch Logs, and scroll down you can add additional roles.

To begin the process of adding the permissions, select the “Manage These Permissions” link as shown in figure 18. Note that right-mouse clicking allows you to open the page in a new tab which is useful for multi-tasking.



Figure 18 Managing Permissions for Lambda

When the new IAM policy page appears, select the Attach policies as shown in figure 19.

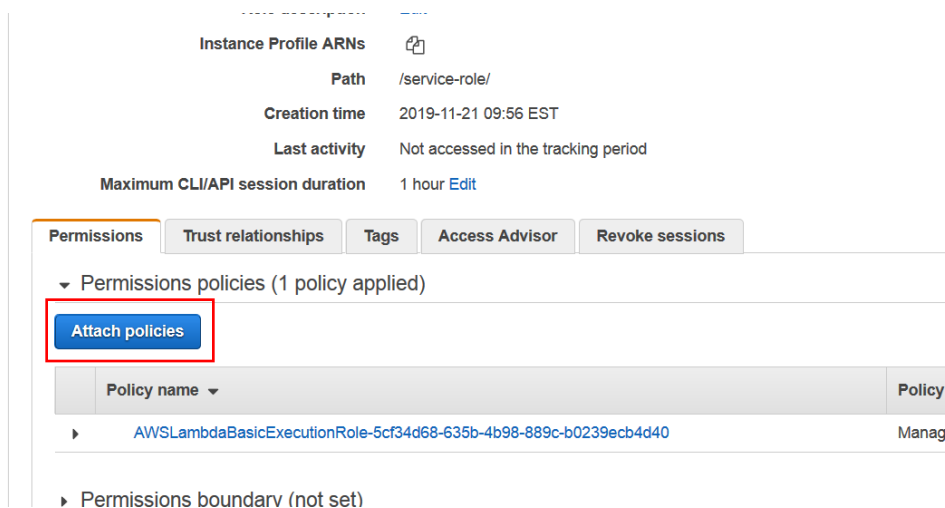


Figure 19 Attaching a Policy

As shown in figure 20, enter **S3** in the Filter policies textarea and then select AmazonS3ReadOnlyAccess. Amazon has multiple preexisting policies to select from associated with most of their services. In this case the S3 ReadOnly provides a user (or assumed role in this case), to have the permissions they need to move forward.

Add permissions to hello-lambda-role-melju11d

Attach Permissions

Create policy

Filter policies

	Policy name
<input type="checkbox"/>	AmazonDMSRedshiftS3Role
<input type="checkbox"/>	AmazonS3FullAccess
<input checked="" type="checkbox"/>	AmazonS3ReadOnlyAccess
<input type="checkbox"/>	QuickSightAccessForS3StorageManagementAnalyticsReadOnly

Figure 20 Selecting the S3 Read Only Policy

To attach the policy, scroll down and select “Attach Policy”. As shown in figure 21, the role now include the AmazonS3ReadOnlyAccess policy.

Role ARN: [arn:aws:iam::064494135434:role/service-role/hello-lambda-role-melju11d](#)

Role description: [Edit](#)

Instance Profile ARNs: [+](#)

Path: /service-role/

Creation time: 2019-11-21 09:56 EST

Last activity: Not accessed in the tracking period

Maximum CLI/API session duration: 1 hour [Edit](#)

Permissions | Trust relationships | Tags | Access Advisor | Revoke sessions

▼ Permissions policies (2 policies applied)

[Attach policies](#)

Policy name	Policy type
AmazonS3ReadOnlyAccess	AWS managed policy
AWSLambdaBasicExecutionRole-5cf34d68-635b-4b98-889c-b0239ecb4d40	Managed policy

Figure 21 The S3 Policy is now Attached

When you go back to the Lambda function page and refresh it, you will see the new S3 roles added as shown in figure 22.

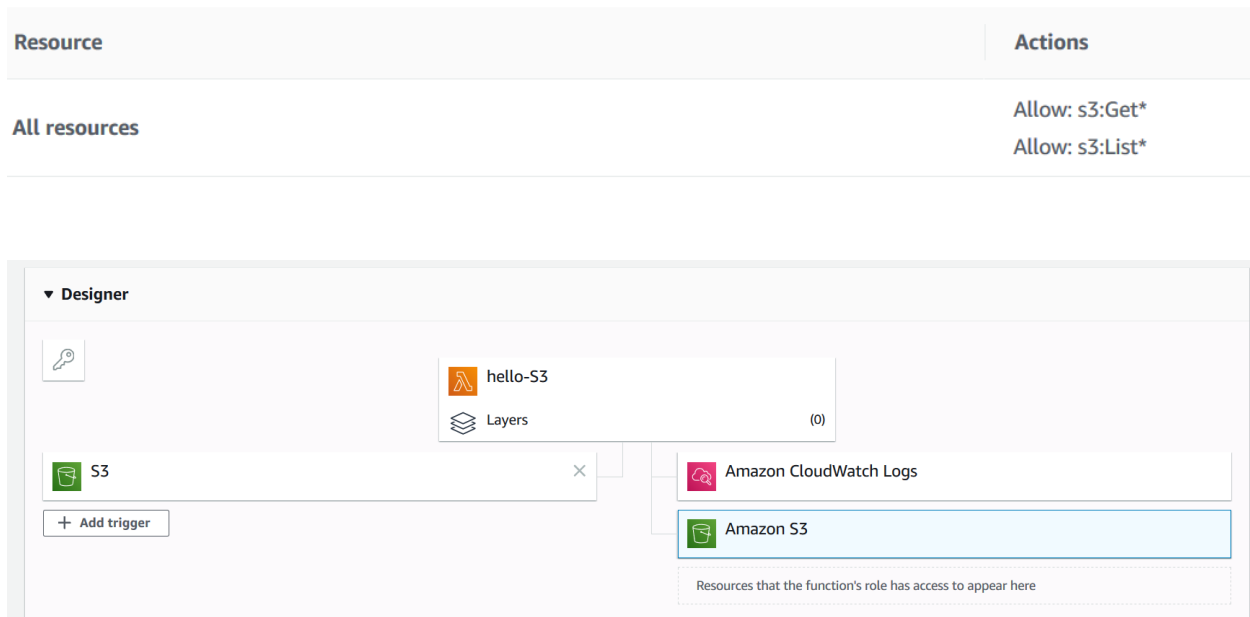


Figure 22 Adding the S3 role

Notice the S3 icon is added under the CloudWatch Logs as well.

Once the role is added (and this function will not work without it), you can upload Tester.dat file to the S3 bucket you aligned with this function. Figure 23 shows the log results after the Tester.dat file is uploaded.

Time (UTC +00:00)	Message
2019-09-29	No older events found at the moment. Re
19:30:34	START RequestId: 328e86c0-1bee-4133-94cf-6c8afb649c86 Version: \$LATEST
19:30:34	Tester.dat was uploaded
19:30:34	Data: Tester.dat us-east-1 2019-09-29T19:30:33.426Z found file is true
19:30:34	END RequestId: 328e86c0-1bee-4133-94cf-6c8afb649c86
19:30:34	REPORT RequestId: 328e86c0-1bee-4133-94cf-6c8afb649c86 Duration: 0.40 ms Billed
	No newer events found at the moment. Re

Figure 23 CloudWatch Logs for S3 trigger

Similarly, if a file that we are not interested, results are logged but with the appropriate “false” notation. (see figure 20).

19:45:13	START RequestId: 3476f8dc-6ef0-4417-8d8c-2ce4de611063 Version: \$LATEST
19:45:13	Data: Testit.dat us-east-1 2019-09-29T19:45:13.246Z found file is false
19:45:13	END RequestId: 3476f8dc-6ef0-4417-8d8c-2ce4de611063
19:45:13	REPORT RequestId: 3476f8dc-6ef0-4417-8d8c-2ce4de611063 Duration: 0.46 ms Billed Duration: 100 ms Memory Size: 1
	No newer events found at the moment. Retrv.

Figure 24 CloudWatch Logs for Other File submission

Each AWS service has test file configurations. For example, Amazon DynamoDB can tested for updates using the Amazon DynamoDB update configuration. Figure 21 shows the test configuration for this option.

```

{
  "Records": [
    {
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "NewImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "awsRegion": "us-east-1",
      "eventName": "INSERT",
      "eventSourceARN": "arn:aws:dynamodb:us-east-1:account-
id:table/ExampleTableWithStream/stream/2015-06-27T00:48:05.899",
      "eventSource": "aws:dynamodb"
    },
    {
      "eventID": "2",
      "eventVersion": "1.0",
      "dynamodb": {
        "OldImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "SequenceNumber": "222",
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "SizeBytes": 59,
        "NewImage": {
          "Message": {
            "S": "This item has changed"
          },

```

```

        "Id": {
            "N": "101"
        }
    },
    "StreamViewType": "NEW_AND_OLD_IMAGES"
},
"awsRegion": "us-east-1",
"eventName": "MODIFY",
"eventSourceARN": "arn:aws:dynamodb:us-east-1:account-
id:table/ExampleTableWithStream/stream/2015-06-27T00:48:05.899",
"eventSource": "aws:dynamodb"
},
{
    "eventID": "3",
    "eventVersion": "1.0",
    "dynamodb": {
        "Keys": {
            "Id": {
                "N": "101"
            }
        }
    },
    "SizeBytes": 38,
    "SequenceNumber": "333",
    "OldImage": {
        "Message": {
            "S": "This item has changed"
        },
        "Id": {
            "N": "101"
        }
    }
},
    "StreamViewType": "NEW_AND_OLD_IMAGES"
},
"awsRegion": "us-east-1",
"eventName": "REMOVE",
"eventSourceARN": "arn:aws:dynamodb:us-east-1:account-
id:table/ExampleTableWithStream/stream/2015-06-27T00:48:05.899",
"eventSource": "aws:dynamodb"
}
]
}

```

Note there are many options for extra data. However; only a few may be of interest for your application. For example, the following Python code looks for the ID and Type for the event records associated with the DynamoDB updates:

```

import json
import boto3

def lambda_handler(event, context):
    # TODO implement
    print('DynamoDB updates')

```

```
    for record in event['Records']:
        print(record['eventID'])
        print(record['eventName'])
        print("DynamoDB Record: " + json.dumps(record['dynamodb'],
indent=2))
    return 'Successfully processed {}
records.'.format(len(event['Records']))
```

Your next assignment will work with Lambda functions associated with S3 buckets, DynamoDB and API Gateway services. Experiment with the examples and variations on a theme to become comfortable with test configurations within the Lambda console for these services.