aws

Amazon Web Services

Lambda

Contents

- ✓ AWS Lambda & Serverless Computing
- ✓ AWS Lambda Concepts
- ✓ Working with Lambda – Basic Steps
- ✓ Demo 1: SNS – Lambda – DynamoDB
- ✓ Demo 2: API Gateway – Lambda – SES
- ✓ Demo 3: S3 – Lambda – DynamoDB & CloudWatch

AWS Lambda

➢ Lambda is a serverless compute service from AWS.

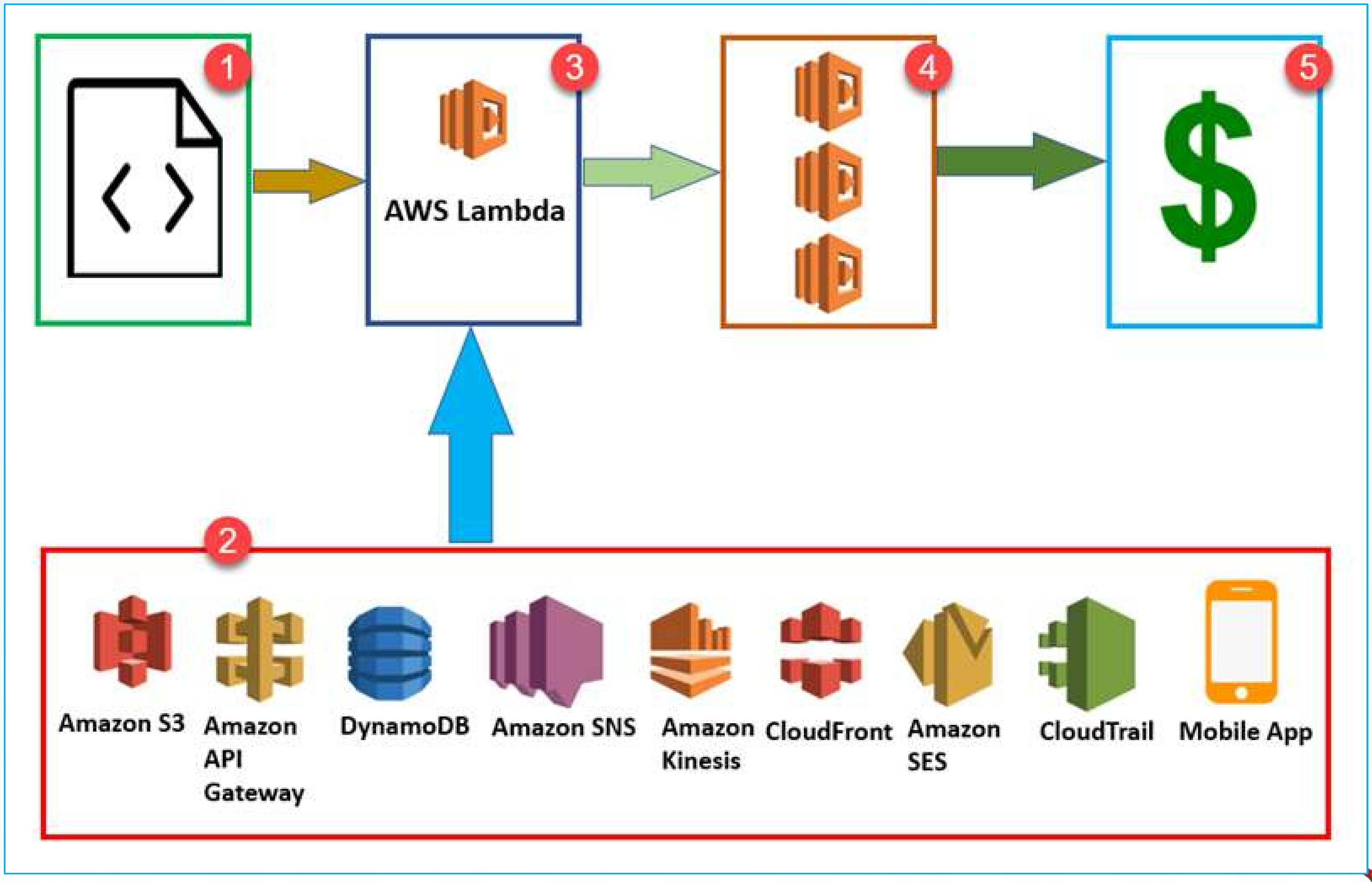
- AWS Lambda is a compute service that lets you run code without provisioning or managing servers.
- AWS Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second.
- You pay only for the compute time you consume - there is no charge when your code is not running

AWS Lambda

➢ Lambda is a serverless compute service from AWS.

- To get working with **AWS Lambda**, we just have to push the code in AWS Lambda service. All other tasks and resources such as infrastructure, operating system, maintenance of server, code monitoring, logs and security is taken care by AWS.
- The code is executed based on the response of events in AWS services such as adding/removing files in S3 bucket, updating Amazon DynamoDB tables, sending SNS notifications etc.
- AWS Lambda supports languages such as Java, Node.js, Python, C#, Go, Ruby

AWS Lambda


1
3
AWS Lambda
4
5
2
Amazon S3
Amazon API Gateway
DynamoDB
Amazon SNS
Amazon Kinesis
CloudFront
Amazon SES
CloudTrail
Mobile App

AWS Lambda

1. Create an AWS Lambda function in any of languages AWS lambda supports – Node.js, Java, Python, C# and Go.

2. AWS Lambda code is triggered when some event happens from any of the several AWS services such as S3, API Gateway, DynamoDB, SNS, Kinesis, CludFront etc.

3. Pricing: AWS charges for the duration and for the resources consumed for executing the code.

AWS Lambda - Advantages

- Ease of deploying Code
- Log Provisioning
- Billing based on Usage
- Multi Language Support

AWS Lambda - Limitations

- It is not suitable for small projects.
- You need to carefully analyze your code and decide the memory and timeout.
    - Incase if your function needs more time than what is allocated, it will get terminated as per the timeout specified on it and the code will not be fully executed.
- Since AWS Lambda relies completely on AWS for the infrastructure, you cannot install anything additional software if your code demands it.

AWS Lambda - Event Triggers

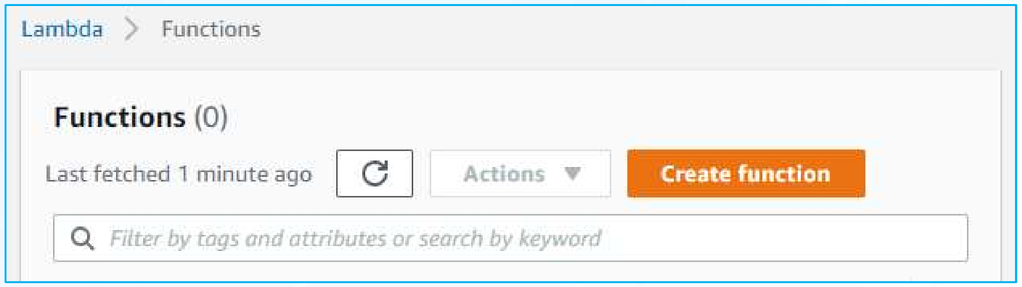
- Entry into a S3 object
- Insertion, updation and deletion of data in Dynamo DB table
- Push notifications from SNS
- GET/POST calls to API Gateway
- Log entries in AWS Kinesis data stream
- Log history in CloudTrail

and more..

Working with Lambda

AWS Lambda - Getting Started

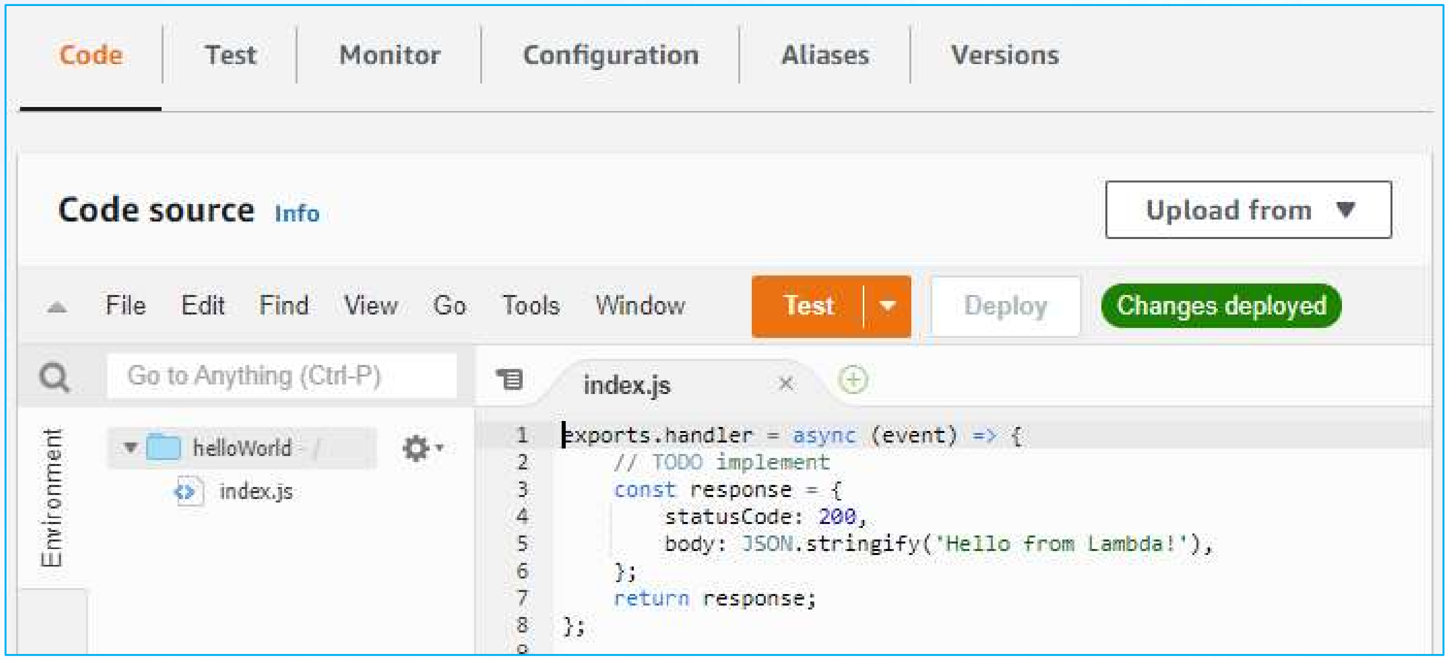
- Open AWS Lambda Service and click on **Create function** button to create a new Lambda function


Lambda > Functions
Functions (0)
Last fetched 1 minute ago
Actions
Create function
Filter by tags and attributes or search by keyword

- Fill the following details:
  - Function name
  - Runtime
  - Change default execution role
    - You may choose to create a new role or use an existing role
  - Click on **Create function** button to create the function

AWS Lambda - Write the function

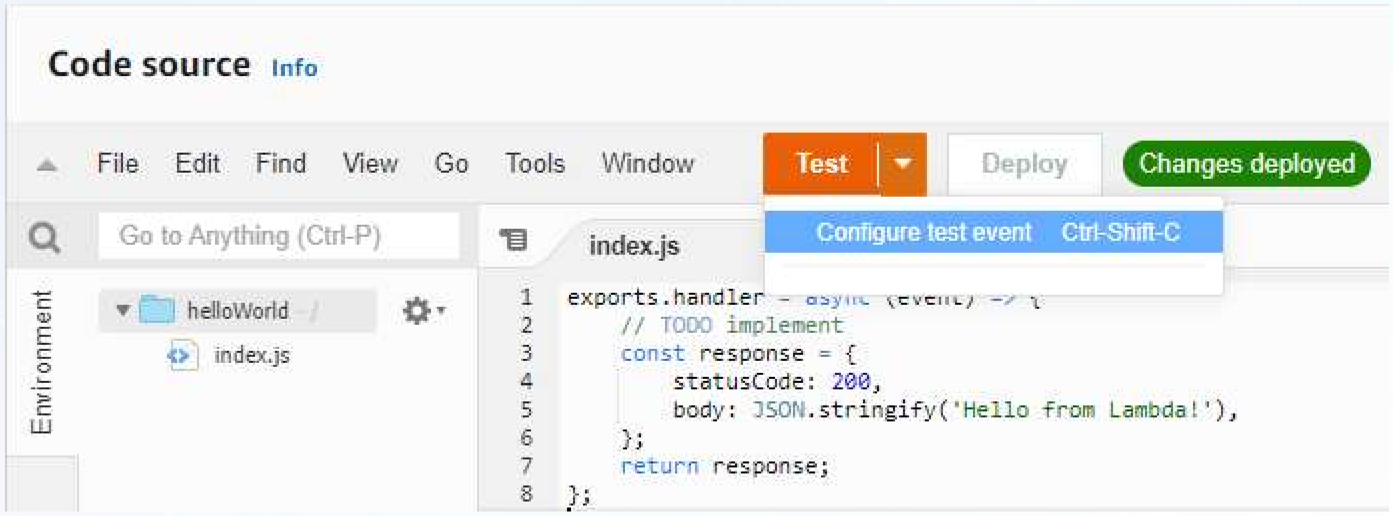
- Open the function and go to the **Code** tab. Write your code the in **Code Source** window.



```
exports.handler = async (event) => {
    // TODO implement
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

AWS Lambda - Test the function

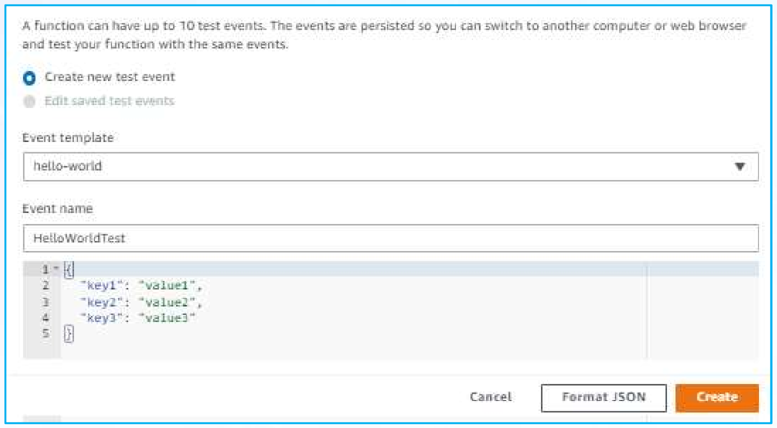
- Click on **Configure test event** option from **Test** menu



```
exports.handler = async (event) => {
    // TODO implement
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

AWS Lambda - Test the function

- Fill the following details and click on **Create** button. After the event is created, click on test button in the **Code Source** window to create the function.



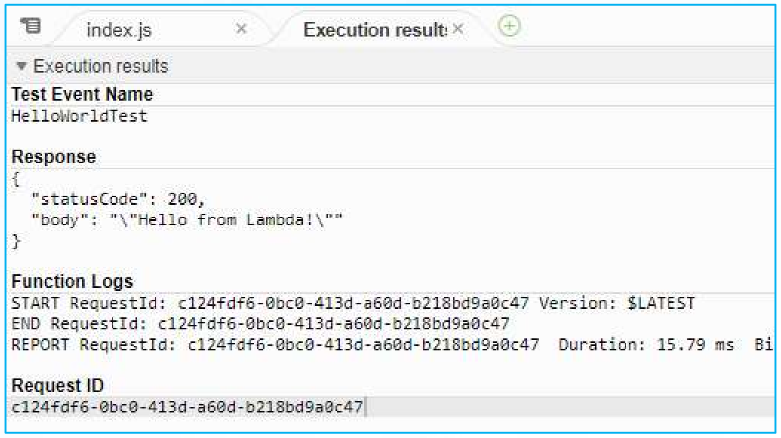
A function can have up to 10 test events. The events are persisted so you can switch to another computer or web browser and test your function with the same events.

[✓ Create new test event] Edit saved test events

Event template: hello-world

Event name: HelloWorldTest

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```



Test Event Name
HelloWorldTest

Response

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

Function Logs

```
START RequestId: c124fdf6-0bc0-413d-a60d-b218bd9a0c47 Version: $LATEST
END RequestId: c124fdf6-0bc0-413d-a60d-b218bd9a0c47
REPORT RequestId: c124fdf6-0bc0-413d-a60d-b218bd9a0c47  Duration: 15.79 ms  Bi
```

Request ID
c124fdf6-0bc0-413d-a60d-b218bd9a0c47

Demo 1: SNS - Lambda - DynamoDB

1. Create an SNS topic - Select **Standard** for **type.**

2. Create a new Lambda function
   - In the permission options, keep ***Create a new role with basic Lambda permissions***

3. Add the script for the function.
   - Refer to the script in the next slide.
   - Make sure your region matches with the one mentioned in the script

4. Create a table called "***message***" in DynamoDB with partition-key as "***messageid***".

5. Add the **SNS topic** as a trigger for your lambda function.
   - select the SNS topic created in step 1

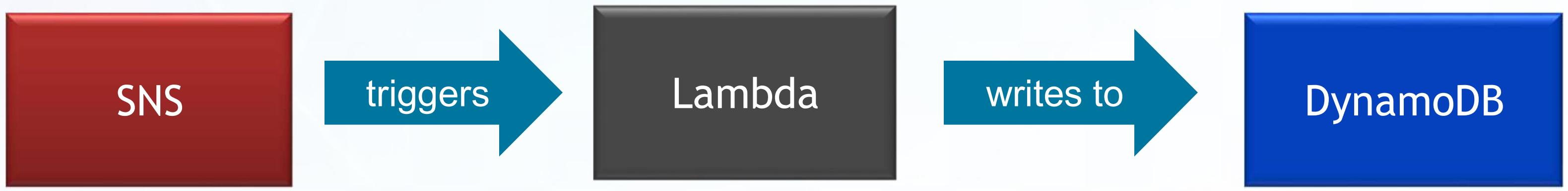
6. Edit the default role created for the lambda function.
   - Click on **Edit policy** option and add **DymanoDB** using add **additional permissions** option
   - Select All actions and all Resources (to keep it simple though not recommended).

7. Create a SNS topic notification test event to test your function.
   - Click on **configure test event** in the Test option.
   - Select **SNS Topic Notification** as template.
   - Modify the data as needed (like giving a MessageId, Subject, Message etc.)

Demo 1: SNS - Lambda - DynamoDB

8. Test the function using Test event created in step 7 mentioned before.
   - Select the Test event and click on Test
   - This should add an item in the messages table in DynamoDB.

9. Go to SNS Topic and publish a few messages
   - The messages you publish in SNS topic should trigger the lambda function that adds these messages as items in the DynamoDB table.



Demo 1: SNS - Lambda - DynamoDB - Script

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient({region: 'us-east-1'});

exports.handler = (event, context, callback) => {
        var params = {
            Item: {
                messageid:event.Records[0].Sns.MessageId,
                message:event.Records[0].Sns.Message,
                timestamp:event.Records[0].Sns.Timestamp
            },
            TableName: 'Message'
        };
        docClient.put(params, function(err, data) {
            if (err) {
               callback(err, null);
            }
            else{
               callback(null, data);
            }
        });
}
```

Demo 2: API Gateway – Lambda – SES

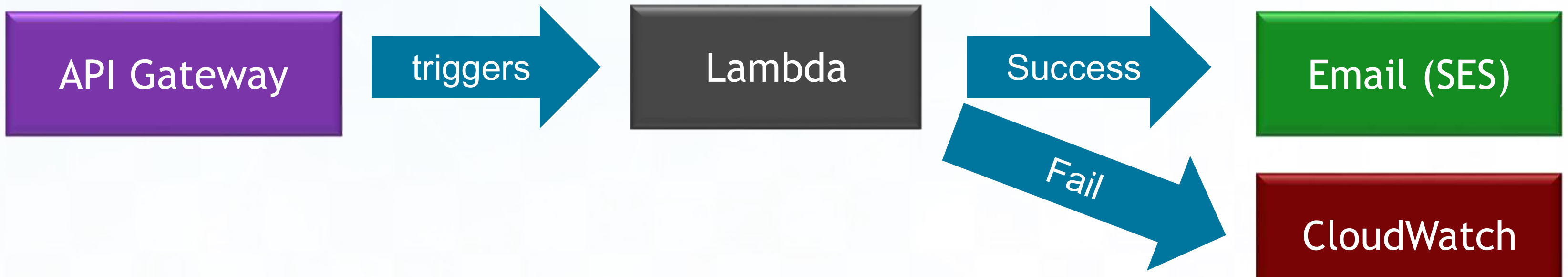
1. Create a new Lambda function
   - In the permission options, keep ***Create a new role with basic Lambda permissions***
2. Add the script for the function.
   - Refer to the script in the next slide.
3. Go to SES (Simple Email Service) and create an identity
   - Click on create an identity button
   - Select Email address option and provide your email.
   - A verification link will be sent to this address. Click on the link to verify your email address
4. Add API Gateway as a trigger to your lambda function
   - Select **Create an API** option, Select type as **HTTP** and security as **Open** and click on **Add**
5. Attach a new policy (SES - SendEmail) to your role
   - **Open the Role** from IAM, Click on **Attach policies** and click on **Add policy**
   - Select Service: **SES**; Actions: **Access Level -> Write -> SendEmail**; Resources: All resources
   - Open the IAM role gain, click on Attach policies and attach the policy that you just created.

Demo 2: API Gateway - Lambda - SES

6. Click on API Gateway trigger and copy the **API endpoint** URL of your API Gateway.

7. Open Postman tool and do the following
   - Select **POST** from the dropdown and past the **API endpoint** URL in the Textbox
   - Select **Raw** option under **Body** and paste the following code.
     ```
     {
           "to": "myemail@mydomain.com",
           "subject": "test message ...!!"
     }
     ```
   - Click on Send

8. This API request sent to API Gateway triggers the Lambda and sends an email to the designated email address



Demo 2: API Gateway - Lambda - SES Script

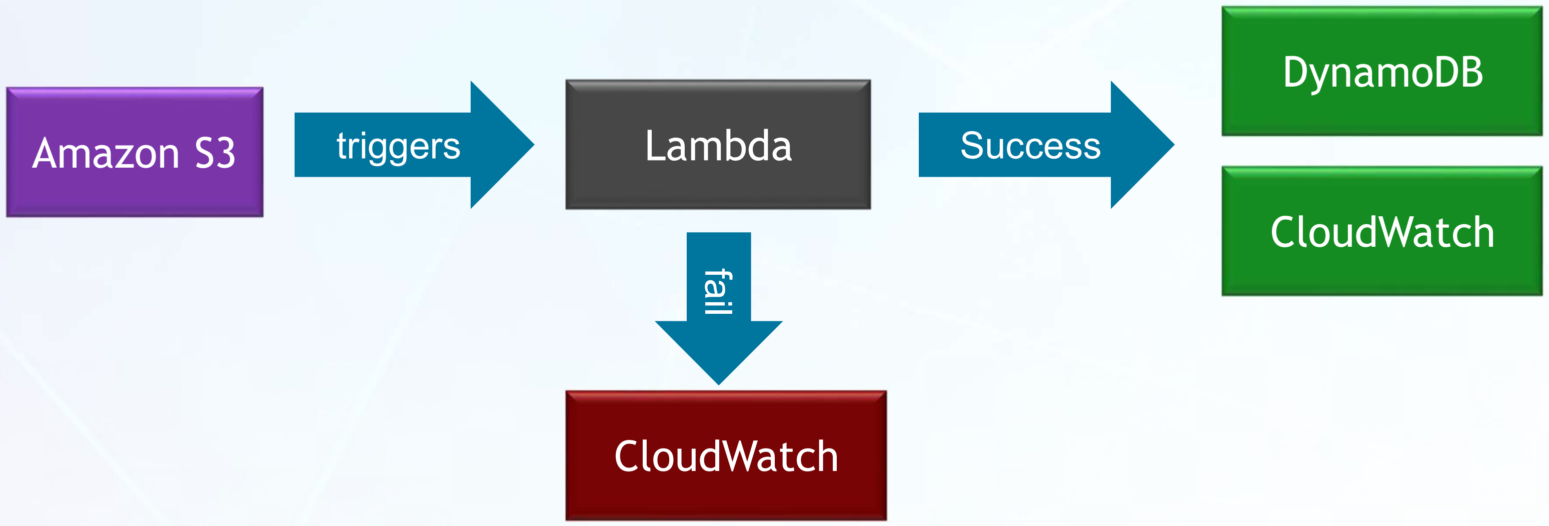
```
const AWS = require('aws-sdk');
const SES = new AWS.SES();
const fs = require('fs');

exports.handler = async (event) => {
    const { to, subject } = JSON.parse(event.body);
    const body = "<html><body>TEST </body></html>";
    const params = {
        Destination: { ToAddresses: [to], },
        Message: {
            Body: { Html: { Data: body } },
            Subject: { Data: subject },
        },
        Source: `myemail@gmail.com'
    };
    try {
        await SES.sendEmail(params).promise();
        return { statusCode: 200, body: 'Email sent!' };
    } catch (e) {
        console.error(e);
        return { statusCode: 400, body: 'Sending failed' };
    }
};
```

Demo 3: S3 - Lambda - DynamoDB (& CloudWatch)

1. Create a new Lambda function
   - In the permission options, keep ***Create a new role with basic Lambda permissions***
2. Add the script for the function.
   - Refer to the script in the next slide.
3. Attach the following policies to the role:
   - AmazonS3FullAccess, CloudWatchFullAccess, AmazonDynamoDBFullAccess
4. Create an S3 bucket
5. Attach S3 bucket as trigger
   - Select the bucket created above and select **all object create events** as event type.
6. Create a DynamoDB table as per script
   - Table Name:**S3Objects**, Partition-key: **id**
7. Now add objects to the S3 bucket.
   - Notice that the events are logged in the CloudWatch as well as added to the DynamoDB table.

Demo 3: S3 - Lambda - DynamoDB (& CloudWatch)



Demo 3: S3 - Lambda - DynamoDB (& CloudWatch)

```
const AWS = require('aws-sdk');
const docClient = new AWS.DynamoDB.DocumentClient({region: 'us-east-1'});

exports.handler = function(event, context, callback) {
   console.log("Incoming Event: ", event);
   const bucket = event.Records[0].s3.bucket.name;
   const filename = decodeURIComponent(event.Records[0].s3.object.key;
   const message = `File is uploaded in - ${bucket} -> ${filename}`;

   var params = {
               Item: {
                         id:event.Records[0].eventTime,
                         bucket:event.Records[0].s3.bucket.name,
                         arn:event.Records[0].s3.bucket.arn,
                         file:event.Records[0].s3.object.key
               },
               TableName: 'S3Objects'
           };
           docClient.put(params, function(err, data) {
               if (err) {
                  console.log("ERROR: File not uploaded");
                  callback(err, null);
               }
               else{
                  console.log(message);
                callback(null, data);
               }
           });
};
```

THANK YOU