

**GEORGIA INSTITUTE OF TECHNOLOGY**  
**SCHOOL of ELECTRICAL and COMPUTER ENGINEERING**

ECE 4813A Spring 2021

**Lab: Serverless Photo Gallery Application using AWS Lambda, API Gateway,  
S3, DynamoDB and Cognito**

---

**References:**

- [1] A. Bahga, V. Madisetti, "Cloud Computing Solutions Architect: A Hands-On Approach", ISBN: 978-0996025591
- [2] <https://aws.amazon.com/documentation/>

**Due Date:**

The lab report will be **due on February 17, 2021.**

---

This lab is about creating a serverless Photo Gallery application. **\*\*\*Make sure your AWS region is set for us-east-1 or N. Virginia (located at the top right of the navigation bar)**

The application uses the following:

1. A web interface implemented in HTML & JS, which can be served through S3 static website hosting
2. AWS API Gateway endpoints
3. Lambda functions
4. Cognito for user pool management
5. DynamoDB for storing records of photos
6. S3 for storing photos

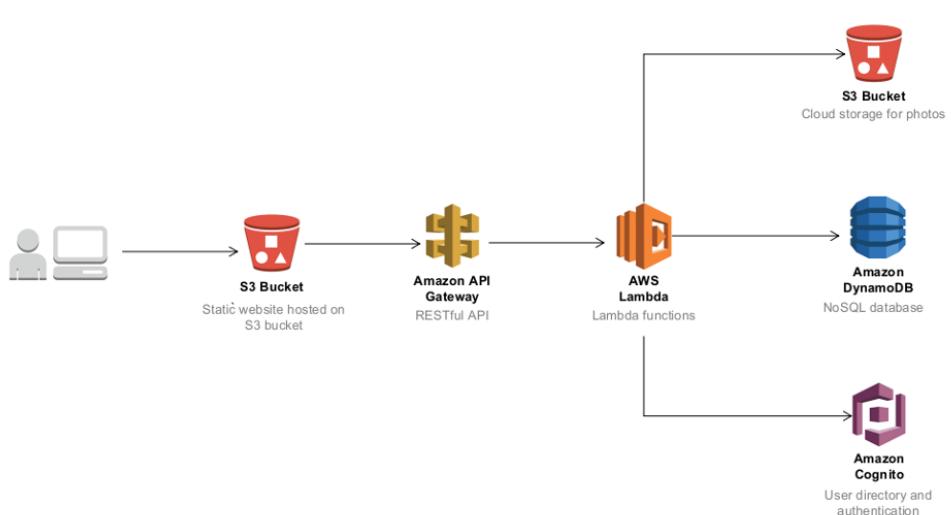


Fig. Architecture diagram of the Photo Gallery application showing AWS services used

Follow the steps below to set up the Photo Gallery application in your AWS account.

## 1. Create an S3 Bucket for hosting the static website for the application

- Create a new S3 bucket for hosting the static website and enable static website hosting for the bucket. **Uncheck Block all public access under Bucket setting for Block Public Access section**
- The bucket name **must be unique**. [Click here to know the rules](#). For the name of the bucket, use **staticwebsite-lastname-year-courseNumber** (e.g., **staticwebsite-corporan-2021-4813**)
- Add a bucket policy below to enable public access to the photos uploaded. Replace '**mybucketname**' with the name of the S3 bucket created.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "PublicReadGetObject",  
            "Effect": "Allow",  
            "Principal": "*",  
            "Action": "s3:GetObject",  
            "Resource": "arn:aws:s3:::mybucketname/*"  
        }  
    ]  
}
```

## 2. Create an S3 Bucket for storing photos

- Create a new S3 bucket for storing photos. **Uncheck Block all public access under Bucket setting for Block Public Access section**
- For the name of the bucket, use **photobucket-lastname-year-courseNumber** (e.g., **photobucket-corporan-2021-4813**)
- Create a folder named 'photos' in this bucket.
- Add a bucket policy below to enable public access to the photos uploaded. Replace '**mybucketname**' with the name of the S3 bucket created.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "PublicReadGetObject",  
            "Effect": "Allow",  
            "Principal": "*",  
            "Action": "s3:GetObject",  
            "Resource": "arn:aws:s3:::mybucketname/photos/*"  
        }  
    ]  
}
```

### 3. Create a DynamoDB table for storing records of photos

- Create a DynamoDB table named **PhotoGallery** with a partition key and a sort key named **PhotoID** and **CreationTime**, respectively, as shown below.

The screenshot shows the 'Create DynamoDB table' wizard. At the top, there's a banner about Kinesis Data Streams for DynamoDB being available. The main form has a 'Table name\*' field containing 'PhotoGallery'. Under 'Primary key\*', there's a 'Partition key' field with 'PhotoID' and a 'String' type dropdown. Below it, a checked checkbox says 'Add sort key' with a 'CreationTime' field and a 'Number' type dropdown. A 'Tutorial' button and a help icon are at the top right.

### 4. Create Cognito User Pool

- Navigate to the AWS Cognito console
- Create a new **User Pool** called **PhotoGalleryUserPool** as shown below. Click "Review defaults"

The screenshot shows the 'Create a user pool' wizard. On the left, a sidebar lists options like Name, Attributes, Policies, MFA and verifications, Message customizations, Tags, Devices, App clients, Triggers, and Review. The 'Name' option is highlighted. The main area has a title 'What do you want to name your user pool?' and a note: 'Give your user pool a descriptive name so you can easily identify it in the future.' A 'Pool name' field contains 'PhotoGalleryUserPool'. Below it, a title 'How do you want to create your user pool?' leads to a box titled 'Review defaults' with the sub-instruction: 'Start by reviewing the defaults and then customize as desired'. A 'Cancel' button is at the top right.

Click “add app client” under App Clients, as shown below

The screenshot shows the AWS User Pools 'Create a user pool' interface. On the left, a sidebar lists navigation options: Name, Attributes, Policies, MFA and verifications, Message customizations, Tags, Devices, App clients (which is currently selected), Triggers, and Review. The main area is titled 'Create a user pool' and contains several configuration sections:

- Pool name:** PhotoGalleryUserPool
- Required attributes:** email
- Alias attributes:** Choose alias attributes...
- Username attributes:** Choose username attributes...
- Enable case insensitivity?**: Yes
- Custom attributes:** Choose custom attributes...
- Minimum password length:** 8
- Password policy:** uppercase letters, lowercase letters, special characters, numbers
- User sign ups allowed?**: Users can sign themselves up
- FROM email address:** Default
- Email Delivery through Amazon SES:** Yes
- MFA:** Enable MFA...
- Verifications:** Email
- Tags:** Choose tags for your user pool
- App clients:** Add app client... (This section is circled in red)
- Triggers:** Add triggers...

A blue 'Create pool' button is located at the bottom right of the configuration area.

Click “Add an app client” and name it **MyCloudApp**. Uncheck “Generate Client Secret”, enable all the authentication checkbox, and click “Create app client” as shown below

Screenshot of the AWS User Pools "Create a user pool" configuration page.

The left sidebar shows navigation options: Name, Attributes, Policies, MFA and verifications, Message customizations, Tags, Devices, **App clients** (selected), Triggers, and Review.

The main area is titled "Which app clients will have access to this user pool?" with a sub-instruction: "The app clients that you add below will be given a unique ID and an optional secret key to access this user pool."

**App client name:** MyCloudApp

**Refresh token expiration:** 30 days and 0 minutes (must be between 60 minutes and 3650 days)

**Access token expiration:** 0 days and 60 minutes (must be between 5 minutes and 1 day, cannot be greater than refresh token expiration)

**ID token expiration:** 0 days and 60 minutes (must be between 5 minutes and 1 day, cannot be greater than refresh token expiration)

**Generate client secret:**  (This checkbox is circled in red.)

**Auth Flows Configuration:**

- Enable username password auth for admin APIs for authentication (ALLOW\_ADMIN\_USER\_PASSWORD\_AUTH) [Learn more.](#)
- Enable lambda trigger based custom authentication (ALLOW\_CUSTOM\_AUTH) [Learn more.](#)
- Enable username password based authentication (ALLOW\_USER\_PASSWORD\_AUTH) [Learn more.](#)
- Enable SRP (secure remote password) protocol based authentication (ALLOW\_USER\_SRP\_AUTH) [Learn more.](#)
- Enable refresh token based authentication (ALLOW\_REFRESH\_TOKEN\_AUTH) [Learn more.](#)

**Security configuration:**

Prevent User Existence Errors [Learn more.](#)

Legacy  
 Enabled (Recommended)

**Set attribute read and write permissions:**

[Cancel](#) [Create app client](#)

[Return to pool details](#)

After creating the app client, return to the pool details by clicking “**Return to pool details**”. The new app client should be included int eh **App Client** section, as shown below

The screenshot shows the 'Create a user pool' review step in the AWS User Pools console. The left sidebar lists navigation options: Name, Attributes, Policies, MFA and verifications, Message customizations, Tags, Devices, App clients (which is selected), Triggers, and Review (which is highlighted). The main area displays various configuration settings:

- Pool name:** PhotoGalleryUserPool
- Required attributes:** email
- Alias attributes:** Choose alias attributes...
- Username attributes:** Choose username attributes...
- Enable case insensitivity?**: Yes
- Custom attributes:** Choose custom attributes...
- Minimum password length:** 8
- Password policy:** uppercase letters, lowercase letters, special characters, numbers
- User sign ups allowed?**: Users can sign themselves up
- FROM email address:** Default
- Email Delivery through Amazon SES:** Yes
- MFA:** Enable MFA...
- Verifications:** Email
- Tags:** Choose tags for your user pool
- App clients:** MyCloudApp (This section is circled in red)
- Triggers:** Add triggers...

A blue 'Create pool' button is located at the bottom right of the review step.

Create the User Pool by clicking “**Create Pool**”.

After creating the pool, save the **Pool Id** for later; we will use this identifier for our serverless function.

The screenshot shows the AWS Cognito User Pools console with the following details:

- General settings:**
  - Pool Id: us-east-1\_Ku6tG6Hqo (highlighted by a red oval)
  - Pool ARN: arn:aws:cognito-idp:us-east-1:797770618644:userpool/us-east-1\_Ku6tG6Hqo
  - Estimated number of users: 0
  - Required attributes: email
  - Alias attributes: none
  - Username attributes: none
  - Enable case insensitivity? Yes
  - Custom attributes: Choose custom attributes...
- Sign-in:**
  - Minimum password length: 8
  - Password policy: uppercase letters, lowercase letters, special characters, numbers
  - User sign ups allowed? Users can sign themselves up
- Email:**
  - FROM email address: Default
  - Email Delivery through Amazon SES: No
    - Note: You have chosen to have Cognito send emails on your behalf. Best practices suggest that customers send emails through Amazon SES for production User Pools due to a daily email limit. Learn more about email best practices.
- MFA:**
  - MFA: Enable MFA...
  - Verifications: Email
- Advanced security:** Enable advanced security...
- Tags:** Choose tags for your user pool
- App clients:** MyCloudApp
- Triggers:** Add triggers...

From the left column, click **App Clients**. Under the new app client (**MyCloudApp**), copy and save the **App client id**; we will use this identifier for our serverless function.

The screenshot shows the AWS Cognito User Pools console. On the left, there's a sidebar with various options like General settings, Users and groups, Attributes, Policies, MFA and verifications, Advanced security, Message customizations, Tags, Devices, App clients (which is highlighted in orange), Triggers, Analytics, and App integration. Below the sidebar, there's a link to 'App client settings'. The main area has a heading 'Which app clients will have access to this user pool?'. It says, 'The app clients that you add below will be given a unique ID and an optional secret key to access this user pool.' A modal window is open, showing a table with one row. The first column contains 'MyCloudApp', and the second column contains 'App client id' with the value '5gpqla43t691ujeain71aiibo'. A large red oval highlights the 'App client id' value.

## 5. Create IAM Roles

- In the AWS IAM console, create a new IAM role called '`lambda_photogallery_role`' for Lambda service, as shown below. Attach policy called `AmazonDynamoDBFullAccess` to this role. Then attach policy called `AmazonCognitoPowerUser` to this role.

The screenshot shows the AWS IAM 'Create role' wizard. Step 1: 'Select type of trusted entity'. It shows four options: 'AWS service' (selected), 'Another AWS account', 'Web identity', and 'SAML 2.0 federation'. Step 2: 'Choose a use case'. It shows 'Common use cases' with 'EC2' and 'Lambda' selected. 'Lambda' is circled in red. Step 3: 'Or select a service to view its use cases'. It lists various services like API Gateway, CloudWatch Events, EKS, IoT Things Graph, Redshift, AWS Backup, CodeBuild, EMR, KMS, Rekognition, AWS Chatbot, CodeDeploy, ElastiCache, Kinesis, and RoboMaker. 'Lambda' is the active tab.

## Create role

1 2 3 4

### ▼ Attach permissions policies

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

[Create policy](#)



Filter policies ▾		Showing 1 result
	Policy name ▾	Used as
<input checked="" type="checkbox"/>	▶  AmazonCognitoPowerUser	None

## Create role

1 2 3 4

### Review

Provide the required information below and review this role before you create it.

**Role name\***

Use alphanumeric and '+=-,@-\_' characters. Maximum 64 characters.

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

Maximum 1000 characters. Use alphanumeric and '+=-,@-\_' characters.

**Trusted entities** AWS service: lambda.amazonaws.com

**Policies** [AmazonDynamoDBFullAccess](#)   
[AmazonCognitoPowerUser](#)

**Permissions boundary** Permissions boundary is not set

No tags were added.

## Create role

1 2 3 4

### ▼ Attach permissions policies

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

[Create policy](#)



Filter policies ▾		Showing 2 results
	Policy name ▾	Used as
<input checked="" type="checkbox"/>	▶  AmazonDynamoDBFullAccess	Permissions policy (1)
<input type="checkbox"/>	▶  AmazonDynamoDBFullAccesswithDataPipeline	None

- Create another IAM role called 'apiS3PutGet-Role' to upload the photos to the S3 Bucket service, as shown below. Attach policy called **AmazonAPIGatewayPushToCloudWatchLogs** to this role. Then attach policy called **AmazonS3FullAccess** to this role.

**Create role**

Select type of trusted entity

<b>AWS service</b> EC2, Lambda and others	<b>Another AWS account</b> Belonging to you or 3rd party	<b>Web identity</b> Cognito or any OpenID provider	<b>SAML 2.0 federation</b> Your corporate directory
--	---	---	--

Allows AWS services to perform actions on your behalf. [Learn more](#)

Choose a use case

**Common use cases**

- EC2**  
Allows EC2 instances to call AWS services on your behalf.
- Lambda**  
Allows Lambda functions to call AWS services on your behalf.
- Or select a service to view its use cases
 

<b>API Gateway</b>	CloudWatch Events	EKS	IoT Things Graph	Redshift
<b>AWS Backup</b>	CodeBuild	EMR	KMS	Rekognition
<b>AWS Chatbot</b>	CodeDeploy	ElastiCache	Kinesis	RoboMaker

**Create role**

**Review**

Provide the required information below and review this role before you create it.

<b>Role name*</b>	apiS3PutGet-Role
Use alphanumeric and '+,-,@-_.' characters. Maximum 64 characters.	
<b>Role description</b>	Allows API Gateway to push logs to CloudWatch Logs.
Maximum 1000 characters. Use alphanumeric and '+,-,@-_.' characters.	

**Trusted entities** AWS service: apigateway.amazonaws.com

**Policies** [AmazonAPIGatewayPushToCloudWatchLogs](#)

**Permissions boundary** Permissions boundary is not set

No tags were added.

**Identity and Access Management (IAM)**

**Dashboard**

**Access management**

- Groups
- Users
- Roles**
- Policies
- Identity providers
- Account settings

**Access reports**

- Archive rules
- Analyzers
- Settings

Credential report

Organization activity

Service control policies (SCPs)

**Create role**

The role apiS3PutGet-Role has been created.

Role name	Trusted entities	Last activity
apiS3PutGet-Role	AWS service: apigateway apiS3PutGet-Role	None
AWSServiceRole...	AWS service: ops.apigateway (Service-Linked role)	286 days
AWSServiceRole...	AWS service: dynamodb.application-autoscale...	Today
AWSServiceRole...	AWS service: autoscaling (Service-Linked role)	289 days
AWSServiceRole...	AWS service: elasticloadbalancing (Service-...	289 days
AWSServiceRole...	AWS service: support (Service-Linked role)	91 days
AWSServiceRole...	AWS service: trustedadvisor (Service-Linked ...)	None
AWSServiceRole...	AWS service: lambda lambda_photogal...	None

AWS Services Search for services, features, marketplace products, and docs [Option+S] VKMGT Global Support

**Identity and Access Management (IAM)**

- Dashboard
- Access management
  - Groups
  - Users
  - Roles**
    - Policies
    - Identity providers
    - Account settings
  - Access reports
    - Access analyzer
    - Archive rules
    - Analyzers
    - Settings
  - Credential report
  - Organization activity
  - Service control policies (SCPs)

Search IAM

**Summary**

Role ARN: arn:aws:iam::797770618644:role/apiS3PutGet-Role

Role description: Allows API Gateway to push logs to CloudWatch Logs. | Edit

Instance Profile ARNs: /

Path: /

Creation time: 2021-01-21 23:47 EST

Last activity: Not accessed in the tracking period

Maximum session duration: 1 hour | Edit

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

▼ Permissions policies (1 policy applied)

**Attach policies** **Add inline policy**

Policy name	Policy type
AmazonAPIGatewayPushTo...	AWS managed policy

▶ Permissions boundary (not set)

AWS Services Search for services, features, marketplace products, and docs [Option+S] VKMGT Global Support

### Add permissions to apiS3PutGet-Role

Attach Permissions

Create policy

Filter policies ▾ Q S3 Showing 6 results

Policy name	Type	Used as
AmazonDMSRedshiftS3Role	AWS managed	None
<b>AmazonS3FullAccess</b>	AWS managed	None
AmazonS3OutpostsFullAccess	AWS managed	None
AmazonS3OutpostsReadOnlyAccess	AWS managed	None
AmazonS3ReadOnlyAccess	AWS managed	None
QuickSightAccessForS3StorageManagementAnalyticsR...	AWS managed	None

AWS Services Search for services, features, marketplace products, and docs [Option+S] VKMGT Global Support

**Identity and Access Management (IAM)**

- Dashboard
- Access management
  - Groups
  - Users
  - Roles**
    - Policies
    - Identity providers
    - Account settings
  - Access reports
    - Access analyzer
    - Archive rules
    - Analyzers
    - Settings
  - Credential report
  - Organization activity
  - Service control policies (SCPs)

Search IAM

**Summary**

Policy AmazonS3FullAccess has been attached for the apiS3PutGet-Role.

Role ARN: arn:aws:iam::797770618644:role/apiS3PutGet-Role

Role description: Allows API Gateway to push logs to CloudWatch Logs. | Edit

Instance Profile ARNs: /

Path: /

Creation time: 2021-01-21 23:47 EST

Last activity: Not accessed in the tracking period

Maximum session duration: 1 hour | Edit

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

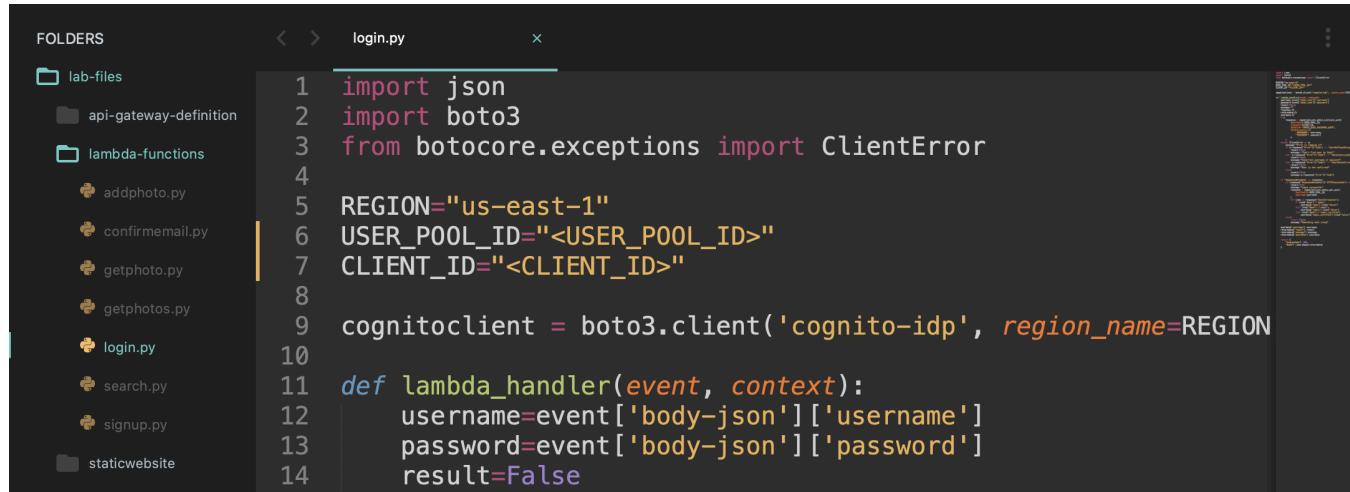
▼ Permissions policies (2 policies applied)

**Attach policies** **Add inline policy**

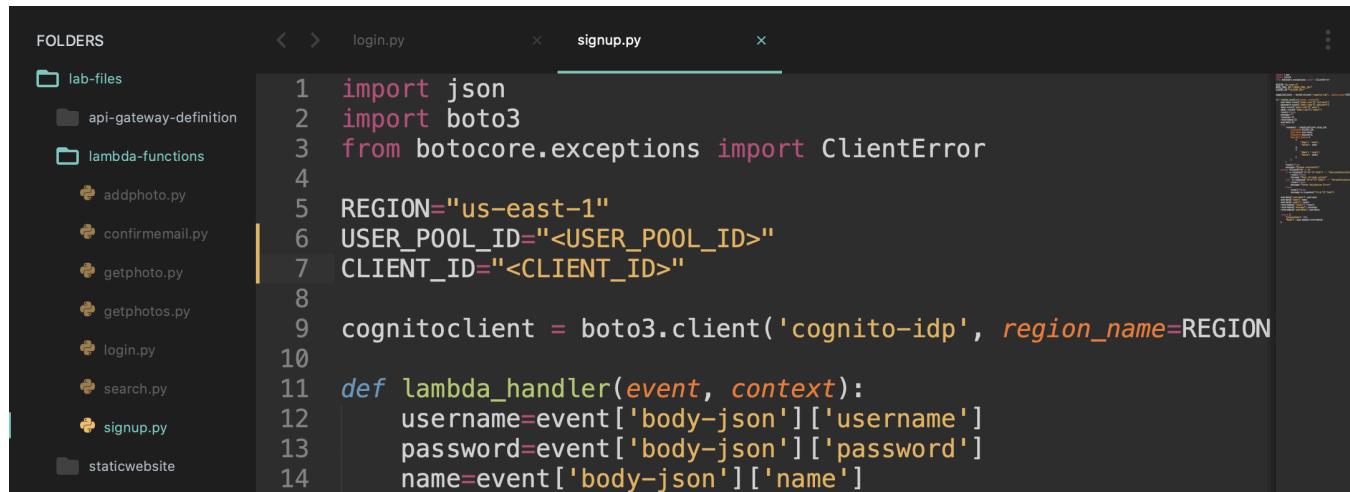
Policy name	Policy type
AmazonS3FullAccess	AWS managed policy
AmazonAPIGatewayPushTo...	AWS managed policy

## 6. Review Code

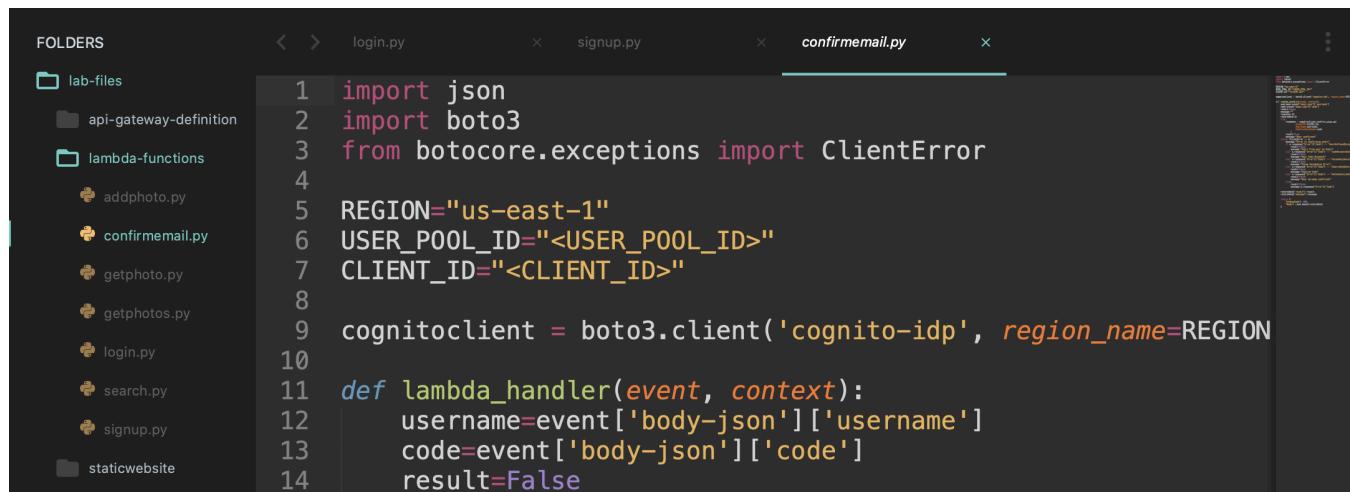
Under the **lambda-functions** directory, open **login.py**, **signup.py**, and **confirmemail.py**; update the <USER\_POOL\_ID> and the <CLIENT\_ID>, in lines 6 and 7, respectively, with the ones you just created.



```
1 import json
2 import boto3
3 from botocore.exceptions import ClientError
4
5 REGION="us-east-1"
6 USER_POOL_ID=<USER_POOL_ID>
7 CLIENT_ID=<CLIENT_ID>
8
9 cognitoclient = boto3.client('cognito-idp', region_name=REGION)
10
11 def lambda_handler(event, context):
12     username=event['body-json']['username']
13     password=event['body-json']['password']
14     result=False
```



```
1 import json
2 import boto3
3 from botocore.exceptions import ClientError
4
5 REGION="us-east-1"
6 USER_POOL_ID=<USER_POOL_ID>
7 CLIENT_ID=<CLIENT_ID>
8
9 cognitoclient = boto3.client('cognito-idp', region_name=REGION)
10
11 def lambda_handler(event, context):
12     username=event['body-json']['username']
13     password=event['body-json']['password']
14     name=event['body-json']['name']
```



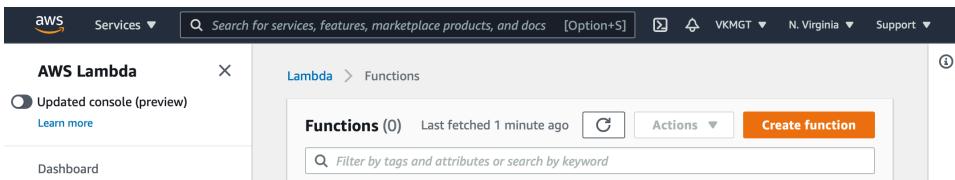
```
1 import json
2 import boto3
3 from botocore.exceptions import ClientError
4
5 REGION="us-east-1"
6 USER_POOL_ID=<USER_POOL_ID>
7 CLIENT_ID=<CLIENT_ID>
8
9 cognitoclient = boto3.client('cognito-idp', region_name=REGION)
10
11 def lambda_handler(event, context):
12     username=event['body-json']['username']
13     code=event['body-json']['code']
14     result=False
```

## 7. Create Lambda functions

We are going to create seven serverless functions under the AWS Lambda console. Each function follow the same process to deploy them. Please, follow each of the screenshots below.

- A. User signup (using source code in `signup.py` file provided):

- Under AWS Lambda console, create a new function



- Give this function a name (**photogallery\_signup**), attached the role previously created (**lambda\_photogallery\_role**) to this function, and then click "**Create function**"

The screenshot shows the 'Create function' wizard. At the top, there are four options: 'Author from scratch' (selected), 'Use a blueprint', 'Container image', and 'Browse serverless app repository'. The main form is titled 'Basic information'.

**Function name:** photogallery\_signup

**Runtime:** Python 2.7

**Permissions:** By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

**Execution role:** Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

Create a new role with basic Lambda permissions  
 Use an existing role  
 Create a new role from AWS policy templates

**Existing role:** Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

lambda\_photogallery\_role

[View the lambda\\_photogallery\\_role role](#) on the IAM console.

**Advanced settings**

Cancel **Create function**

- After creating the function, under “**Function Code**” copy and paste the code provided.

The screenshot shows the AWS Lambda console interface. At the top, there are three tabs: "photogallery\_confirmemail - Lambda", "User Pools - Amazon Cognito", and "photogallery\_signup - Lambda". The middle tab is active. The main area shows the function configuration for "photogallery\_signup". The "Designer" tab is selected, displaying a single function icon labeled "photogallery\_signup". Below the icon are buttons for "+ Add trigger" and "+ Add destination". To the right of the icon, it says "(0)" for layers. The "Function code" tab is also visible, showing the Python code for the lambda function.

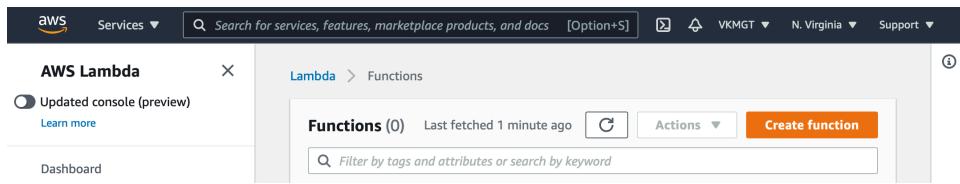
```

import json
import boto3
from botocore.exceptions import ClientError
REGION="us-east-1"
USER_POOL_ID="us-east-1_Ku6tG6Hqo"
CLIENT_ID="5gpqla43t691ujeain7laiibo"
cognitoClient = boto3.client('cognito-idp', region_name=REGION)
def lambda_handler(event, context):
    username=event['body-json']['username']
    password=event['body-json']['password']
    name=event['body-json']['name']
    email=event['body-json']['email']
    result=False
    message=""
    response={}
    returndata={}
    userdata={}
    try:
        response = cognitoclient.sign_up(
            ClientId=CLIENT_ID,
            Username=username,
            Password=password,
            UserAttributes=[
                {
                    'Name': 'name',
                    'Value': name
                },
                {
                    'Name': 'email',
                    'Value': email
                }
            ]
    
```

- Click “**Deploy**” to release the function

B. Confirming user email after signup (using source code in confirmemail.py file provided):

- Under AWS Lambda console, create a new function



- Give this function a name (**photogallery\_confirmemail**), attached the role previously created (**lambda\_photogallery\_role**) to this function, and then click "**Create function**"

The screenshot shows the 'Create function' wizard. Step 1: Choose a blueprint. It includes four options: 'Author from scratch' (selected), 'Use a blueprint', 'Container image', and 'Browse serverless app repository'. The 'Basic information' step is shown below:

**Basic information**

Function name: photogallery\_confirmemail

Runtime: Python 2.7

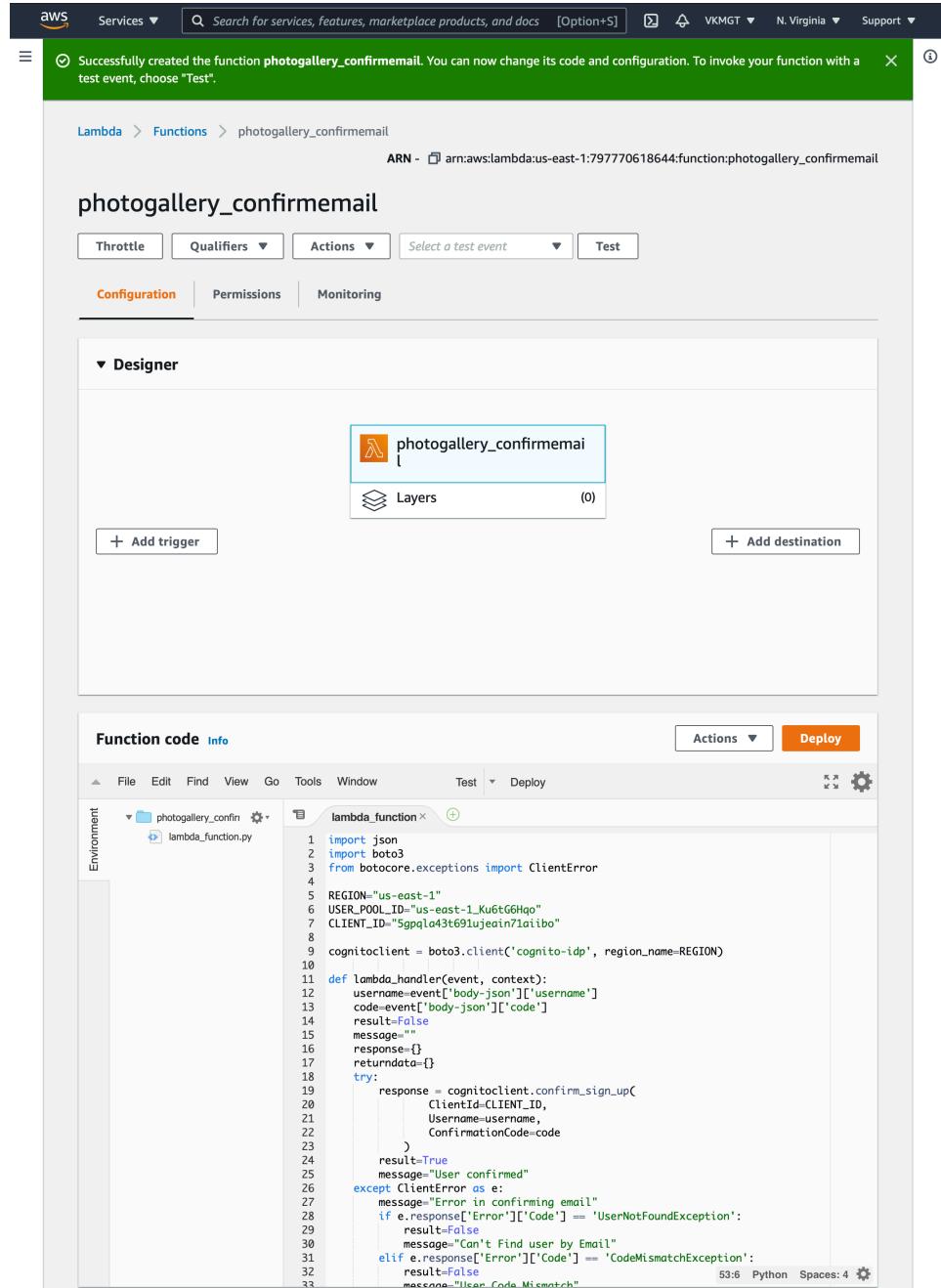
**Permissions**

Execution role: lambda\_photogallery\_role

**Advanced settings**

Cancel Create function

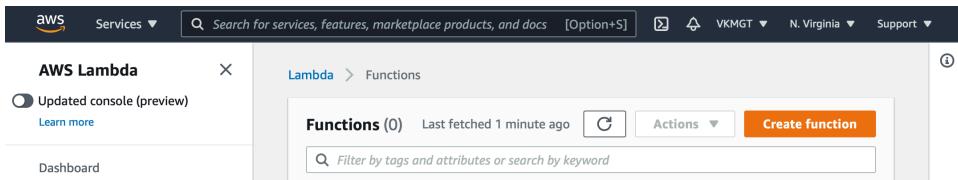
- After creating the function, under “**Function Code**” copy and paste the code provided.



- Click “**Deploy**” to release the function

C. User login (using source code in login.py file provided):

- Under AWS Lambda console, create a new function



- Give this function a name (**photogallery\_login**), attached the role previously created (**lambda\_photogallery\_role**) to this function, and then click "**Create function**"

The screenshot shows the 'Create function' wizard. At the top, there are four options: 'Author from scratch' (selected), 'Use a blueprint', 'Container image', and 'Browse serverless app repository'. The 'Basic information' step is currently active. It includes fields for 'Function name' (set to 'photogallery\_login'), 'Runtime' (set to 'Python 2.7'), and 'Permissions' (set to 'lambda\_photogallery\_role'). The 'Advanced settings' step is partially visible at the bottom.

**Basic information**

**Function name**  
Enter a name that describes the purpose of your function.  
  
Use only letters, numbers, hyphens, or underscores with no spaces.

**Runtime** [Info](#)  
Choose the language to use to write your function.

**Permissions** [Info](#)  
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

**Change default execution role**

**Execution role**  
Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

Create a new role with basic Lambda permissions  
 Use an existing role  
 Create a new role from AWS policy templates

**Existing role**  
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

**Advanced settings**

- After creating the function, under “**Function Code**” copy and paste the code provided.

The screenshot shows the AWS Lambda console interface. At the top, a green banner indicates: "Successfully created the function photogallery\_login. You can now change its code and configuration. To invoke your function with a test event, choose 'Test'." Below this, the function name "photogallery\_login" is displayed, along with its ARN: arn:aws:lambda:us-east-1:797770618644:function:photogallery\_login. Navigation tabs include Throttle, Qualifiers, Actions, Select a test event, and Test. The Configuration tab is selected. In the Designer section, there is a box labeled "photogallery\_login" containing a "Layers" section with "(0)". Buttons for "+ Add trigger" and "+ Add destination" are visible. Below the Designer is the "Function code" editor, which contains the following Python code:

```

 37     message="User is not confirmed"
 38 else:
 39     result=False
 40     message=e.response['Error']['Code']
 41
 42 if 'ResponseMetadata' in response:
 43     if response['ResponseMetadata']['HTTPStatusCode']==200:
 44         result=True
 45         message="Login successful"
 46         response = cognitoclient.admin_get_user(
 47             UserPoolId=USER_POOL_ID,
 48             Username=username
 49         )
 50         for item in response['UserAttributes']:
 51             if item['Name']=='name':
 52                 userdata['name']=item['Value']
 53             elif item['Name']=='email':
 54                 userdata['email']=item['Value']
 55             elif item['Name']=='email_verified':
 56                 userdata['email_verified']=item['Value']
 57             else:
 58                 return False
 59             message="Something went wrong"
 60
 61             userdata['username']=username
 62             returndata['result']=result
 63             returndata['message']=message
 64             returndata['userdata']=userdata
 65
 66             return {
 67                 "statusCode": 200,
 68                 "body": json.dumps(returndata)
 69             }
 70         ]
 71     ]

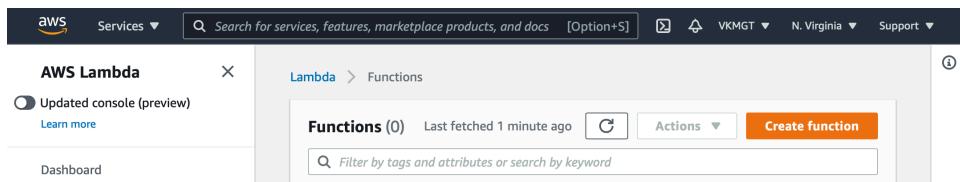
```

The code handles user login and retrieves user attributes from AWS Cognito. It returns a success message with user data or an error message if the user is not confirmed or if there's a problem with the login response.

- Click “**Deploy**” to release the function

D. Getting details of all photos (using source code in getphotos.py file provided):

- Under AWS Lambda console, create a new function



- Give this function a name (**photogallery\_getphotos**), attached the role previously created (**lambda\_photogallery\_role**) to this function, and then click "**Create function**"

**Basic information**

Function name  
Enter a name that describes the purpose of your function.

Runtime [Info](#)  
Choose the language to use to write your function.

**Permissions** [Info](#)  
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

**Change default execution role**

Execution role  
Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

Create a new role with basic Lambda permissions  
 Use an existing role  
 Create a new role from AWS policy templates

Existing role  
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

[View the lambda\\_photogallery\\_role role](#) on the IAM console.

**Advanced settings**

- After creating the function, under “**Function Code**” copy and paste the code provided.

The screenshot shows the AWS Lambda console interface. At the top, the navigation bar includes 'Services ▾', a search bar ('Search for services, features, marketplace products, and docs [Option+S]'), and account information ('ARN - arn:aws:lambda:us-east-1:797770618644:function:photogallery\_getphotos', 'VKMGT', 'N. Virginia', 'Support'). Below the navigation, the path 'Lambda > Functions > photogallery\_getphotos' is shown, along with the ARN. The main area is titled 'photogallery\_getphotos'. It features tabs for 'Throttle', 'Qualifiers', 'Actions', 'Select a test event', 'Test', 'Configuration' (which is selected), 'Permissions', and 'Monitoring'. Under the 'Configuration' tab, there's a 'Designer' section with a box labeled 'photogallery\_getphotos' containing a 'Layers' section with '(0)'. Buttons for '+ Add trigger' and '+ Add destination' are visible. Below this is a large empty space. The bottom half of the screen shows the 'Function code' editor. The title bar says 'Function code Info' with 'Actions ▾'. The menu bar includes 'File', 'Edit', 'Find', 'View', 'Go', 'Tools', 'Window', 'Test', 'Deploy' (which is highlighted in orange), and 'Changes not deployed'. The code editor window has a sidebar for 'Environment' and a search bar 'Go to Anything (% P)'. The code itself is in a Python file named 'lambda\_function.py':

```

1 import json
2 import boto3
3
4 REGION="us-east-1"
5 dynamodb = boto3.resource('dynamodb', region_name=REGION)
6 table = dynamodb.Table('PhotoGallery')
7
8 def lambda_handler(event, context):
9     response = table.scan()
10    items = response['Items']
11    #return items
12    return {
13        "statusCode": 200,
14        "body": items
15    }
16

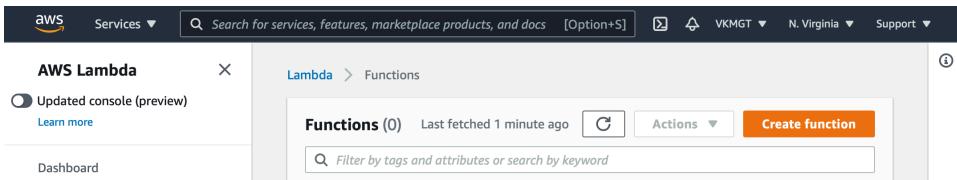
```

The status bar at the bottom right indicates '16:1 Python Spaces: 4'.

- Click “**Deploy**” to release the function

E. Getting details of a specific photo (using source code in getphoto.py file provided):

- Under AWS Lambda console, create a new function



- Give this function a name (**photogallery\_getphoto**), attached the role previously created (**lambda\_photogallery\_role**) to this function, and then click "**Create function**"

**Basic information**

**Function name**  
Enter a name that describes the purpose of your function.

**Runtime** [Info](#)  
Choose the language to use to write your function.

**Permissions** [Info](#)  
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

**Change default execution role**

**Execution role**  
Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

Create a new role with basic Lambda permissions  
 Use an existing role  
 Create a new role from AWS policy templates

**Existing role**  
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

[View the lambda\\_photogallery\\_role role](#) on the IAM console.

**Advanced settings**

[Cancel](#) [Create function](#)

- After creating the function, under “**Function Code**” copy and paste the code provided.

The screenshot shows the AWS Lambda console interface. At the top, a green banner indicates that the function 'photogallery\_getphoto' has been successfully created. Below this, the 'photogallery\_getphoto' function page is displayed. The 'Configuration' tab is selected, showing the 'Designer' section where triggers and destinations can be managed. The 'Function code' tab is also visible, displaying the Python code for the lambda function.

```

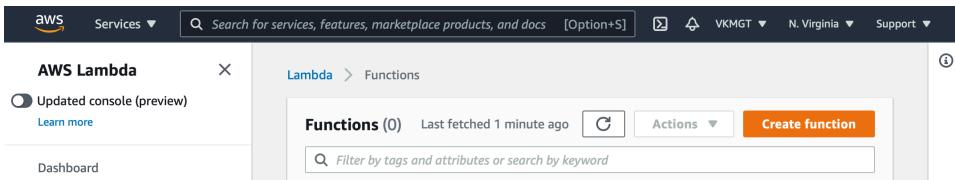
import json
import boto3
from boto3.dynamodb.conditions import Key, Attr
REGION="us-east-1"
dynamodb = boto3.resource('dynamodb',region_name=REGION)
table = dynamodb.Table('PhotoGallery')
def lambda_handler(event, context):
    photoID=event['params']['path'][1]['id']
    print(photoID)
    response = table.scan(
        FilterExpression=Attr('PhotoID').eq(str(photoID))
    )
    print(response)
    items = response['Items']
    #print(items)
    returnres= {
        "statusCode": 200,
        "body": items,
        "headers": {"Access-Control-Allow-Origin": "*"}
    }
    print(returnres)
    return returnres

```

- Click “**Deploy**” to release the function

F. Adding a photo (using source code in addphoto.py file provided):

- Under AWS Lambda console, create a new function



- Give this function a name (**photogallery\_addphoto**), attached the role previously created (**lambda\_photogallery\_role**) to this function, and then click "**Create function**"

Author from scratch  
Start with a simple Hello World example.

Use a blueprint  
Build a Lambda application from sample code and configuration presets for common use cases.

Container image  
Select a container image to deploy for your function.

Browse serverless app repository  
Deploy a sample Lambda application from the AWS Serverless Application Repository.

---

#### Basic information

Function name  
Enter a name that describes the purpose of your function.

Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime [Info](#)  
Choose the language to use to write your function.

Permissions [Info](#)  
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

▼ Change default execution role

Execution role  
Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

Create a new role with basic Lambda permissions  
 Use an existing role  
 Create a new role from AWS policy templates

Existing role  
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

[View the lambda\\_photogallery\\_role role on the IAM console](#).

► Advanced settings

Cancel Create function

- After creating the function, under “**Function Code**” copy and paste the code provided.

The screenshot shows the AWS Lambda console interface. At the top, a green success message states: "Successfully created the function photogallery\_addphoto. You can now change its code and configuration. To invoke your function with a test event, choose 'Test'." Below this, the function name "photogallery\_addphoto" is displayed, along with ARN and configuration tabs. The "Designer" tab is selected, showing a single trigger named "photogallery\_addphoto". The "Function code" tab is open, displaying the Python code for the lambda\_handler function:

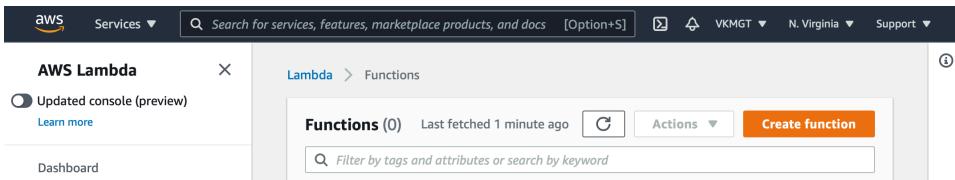
```

1 import json
2 import boto3
3 import time
4 import datetime
5
6 REGION="us-east-1"
7 dynamodb = boto3.resource('dynamodb',region_name=REGION)
8 table = dynamodb.Table('PhotoGallery')
9
10 def lambda_handler(event, context):
11     username=event['body-json']['username']
12     title=event['body-json']['title']
13     description=event['body-json']['description']
14     tags=event['body-json']['tags']
15     uploadedFileURL=event['body-json']['uploadedFileURL']
16     #ExifData=event['body-json']['ExifData']
17     ts=time.time()
18     #timestamp = datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d %H:%M:%S')
19     timestamp=int(ts*1000)
20     photoID=str(timestamp)
21
22     table.put_item(
23         Item={
24             "PhotoID": photoID,
25             "Username": username,
26             "CreationTime": timestamp,
27             "Title": title,
28             "Description": description,
29             "Tags": tags,
30             "URL": uploadedFileURL
31             "#ExifData": ExifData
32         }
33     )
    
```

- Click “**Deploy**” to release the function

## G. Searching photos (using source code in search.py file provided):

- Under AWS Lambda console, create a new function



- Give this function a name (**photogallery\_search**), attached the role previously created (**lambda\_photogallery\_role**) to this function, and then click "**Create function**"

**Basic information**

**Function name**  
Enter a name that describes the purpose of your function.

**Runtime** [Info](#)  
Choose the language to use to write your function.

**Permissions** [Info](#)  
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

**Change default execution role**

**Execution role**  
Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

Create a new role with basic Lambda permissions  
 Use an existing role  
 Create a new role from AWS policy templates

**Existing role**  
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

[View the lambda\\_photogallery\\_role role on the IAM console.](#)

**Advanced settings**

[Cancel](#) [Create function](#)

- After creating the function, under “**Function Code**” copy and paste the code provided.

The screenshot shows the AWS Lambda console interface. At the top, a green banner indicates that the function 'photogallery\_search' has been successfully created. Below this, the 'photogallery\_search' function details are shown, including its ARN and configuration options like Throttle, Qualifiers, Actions, and Test. The 'Configuration' tab is selected. In the 'Designer' section, there is a single trigger named 'photogallery\_search'. Below it, there are buttons for '+ Add trigger' and '+ Add destination'. The 'Function code' tab is open, showing the Python code for the lambda function:

```

1 import json
2 import boto3
3 import time
4 from boto3.dynamodb.conditions import Key, Attr
5 REGION="us-east-1"
6 dynamodb = boto3.resource('dynamodb',region_name=REGION)
7 table = dynamodb.Table('PhotoGallery')
8
9 def lambda_handler(event, context):
10     query = event['body-json']['query']
11
12     response = table.scan(
13         FilterExpression=Attr('Title').contains(str(query)) | Attr('Description').contains(str(query)))
14
15     items = response['Items']
16
17     return {
18         "statusCode": 200,
19         "body": items
20     }
21
22
23
24
25
26

```

The code uses the Boto3 library to interact with a DynamoDB table named 'PhotoGallery'. It defines a lambda handler that takes an event with a 'body-json' key containing a 'query' parameter. It then performs a scan operation on the table, filtering items where either the 'Title' or 'Description' attribute contains the query string. The resulting items are returned as a JSON response with a status code of 200.

- Click “**Deploy**” to release the function

## 8. Create a new API from API Gateway

Go to the AWS API Gateway Console. Once you enter the console, it will provide you with an option to create an Example API with some resources and method; disregard that option and select “**New API**”. Create a new API called **PhotoGalleryAPI** with the resources, methods, and integration shown below.

Create new API

In Amazon API Gateway, a REST API refers to a collection of resources and methods that can be invoked through HTTPS endpoints.

New API    Import from Swagger or Open API 3    Example API

**Settings**

Choose a friendly name and description for your API.

API name\*

Description

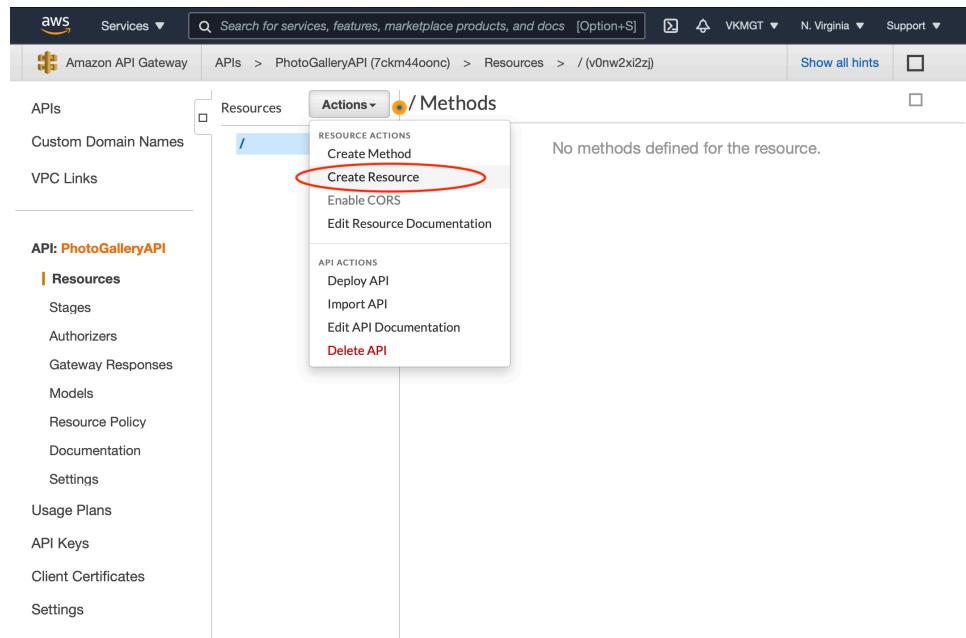
Endpoint Type

\* Required

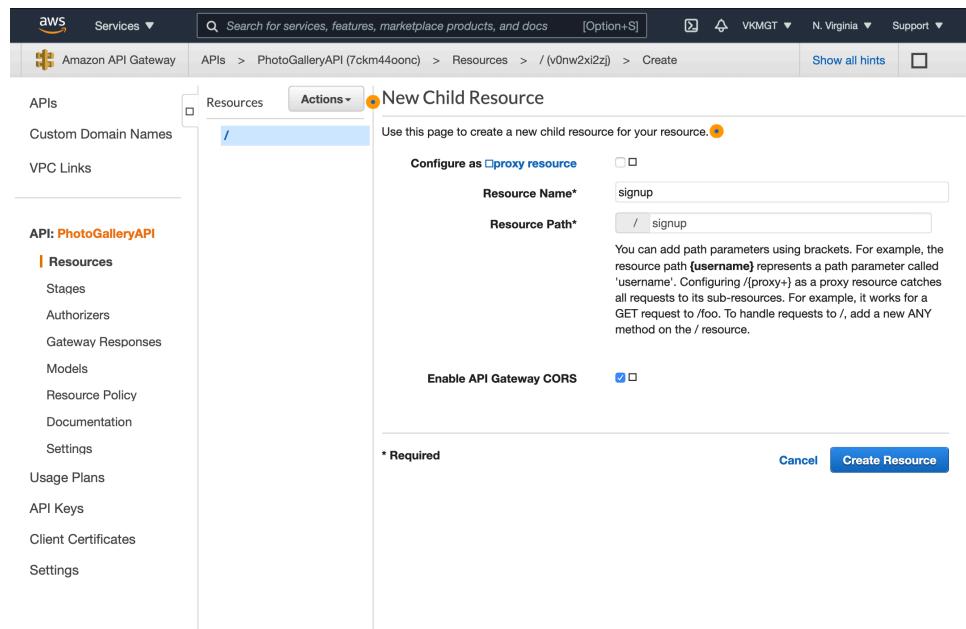
Resource	Method	Integration Request
/signup	POST	Lambda Function: photogallery_signup
/confirmemail	POST	Lambda Function: photogallery_confirmemail
/login	POST	Lambda Function: photogallery_login
/photos	POST	Lambda Function: photogallery_addphoto
/photos	GET	Lambda Function: photogallery_getphotos
/photos/{id}	GET	Lambda Function: photogallery_getphoto
/uploadphoto	PUT	S3 Bucket: photobucket-corporan-2021-4813
/search	POST	Lambda Function: photogallery_search

For **/signup**, a POST request a lambda function:

- Create a new **Resource** under the **Actions** dropdown button



- Give a name to the resource and make sure that "Enable API Gateway CORS" is checked



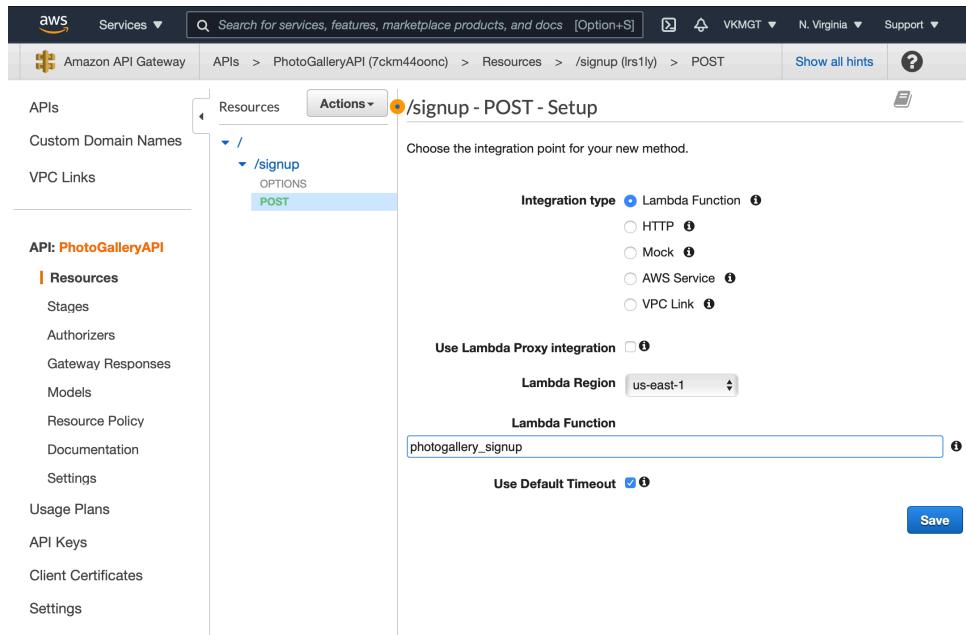
- Once the resource is created, select the resource and create a new **Method** under that resource.

The screenshot shows the AWS API Gateway console. On the left, the navigation pane is visible with sections like APIs, Custom Domain Names, VPC Links, and API: PhotoGalleryAPI (Resources, Stages, Authorizers, etc.). In the main area, the path is APIs > PhotoGalleryAPI > Resources > /signup (rvyp48). A context menu is open over the /signup resource, with the 'Actions' dropdown expanded. The 'Create Method' option is highlighted with a red circle. The menu also includes options like Create Resource, Enable CORS, Edit Resource Documentation, Delete Resource, API ACTIONS (Deploy API, Import API, Edit API Documentation), and Delete API.

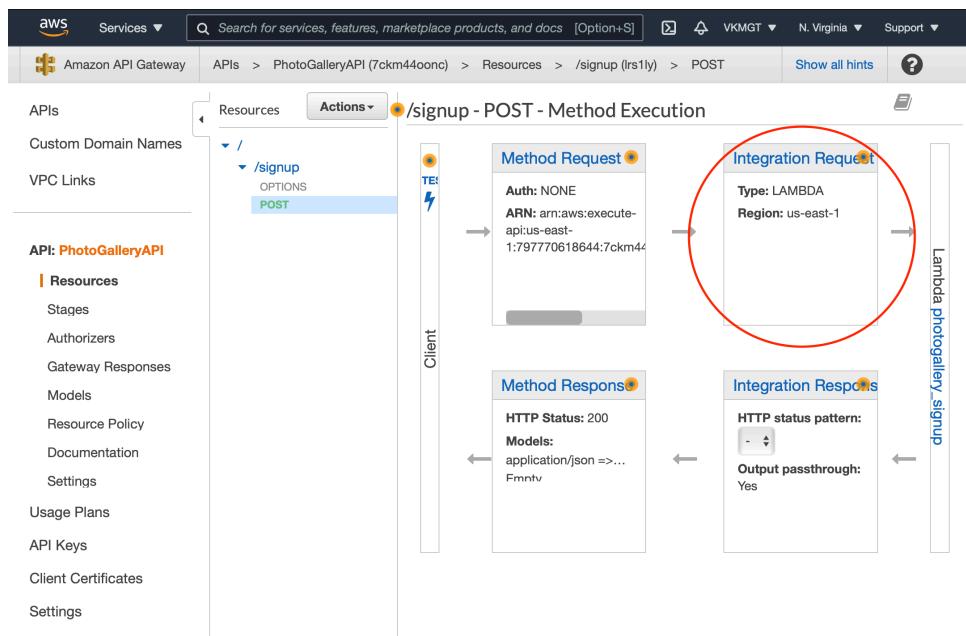
- Select **POST** as the method for this resource

The screenshot shows the AWS API Gateway console with the same navigation and path as the previous screenshot. The context menu is now open over the /signup resource, specifically over the 'OPTIONS' method. The 'POST' method is selected from a dropdown menu. The right panel displays the 'OPTIONS' method configuration, showing 'Mock Endpoint' and 'Authorization' set to 'None'. The 'API ACTIONS' section is also visible.

- Select the method created and attach the lambda function as the integration type.



- After saving the integration type, select “**Integration Request**” under the method. Under the “**Mapping Template**”, choose “**When there are no templates defined (recommended)**” under the Request body passthrough. Under the **Content-Type**, click “**Add mapping**” and add “**application/json**” to the mapping. Finally, Under the **Generate template** dropdown, select “**Method Request passthrough**” and click “**Save**”.



The screenshot shows the AWS API Gateway console. On the left, the navigation pane lists the API: PhotoGalleryAPI and its resources, stages, authorizers, and other settings. The main area shows the configuration for the /signup resource, specifically the POST method. The 'Actions' dropdown is open, and the 'POST' method is selected. The configuration details include:

- Integration type:** Lambda Function (selected)
- Lambda Region:** us-east-1
- Lambda Function:** photogallery\_signup
- Execution role:** (not explicitly named)
- Invoke with caller credentials:** (unchecked)
- Credentials cache:** Do not add caller credentials to cache key (unchecked)
- Use Default Timeout:** (checked)
- URL Path Parameters:** (empty)
- URL Query String Parameters:** (empty)
- HTTP Headers:** (empty)
- Mapping Templates:** (selected)
  - Request body passthrough:** When there are no templates defined (recommended) (selected)
  - Content-Type:** application/json
  - Add mapping:** (button)
- Generate template:** (button)
- Template code:**

```

1 ## See http://docs.aws.amazon.com
   /apigateway/latest/developerguide
   /api-gateway-mapping-template
   -reference.html
2 ## This template will pass through
   all parameters including path,
   querystring, header, stage
   variables, and context through to
   the integration endpoint via the
   body/payload
3 #set($allParams = $input.params())
4 {
5   "body-json" : $input.json('$'),
6   "params" : {
7     #foreach($type in $allParams.keySet())
8       #set($param = $allParams.get(
9         $type))
10      "$type" : {
11        #foreach($paramName in $params)
12          #foreach($paramValue in $param)
13            "$paramName" : $paramValue
14          #end
15        #end
16      }
17    }
18  }
  
```
- Cancel** and **Save** buttons.

- To create another resource, click "/" under the Resources column

The screenshot shows the AWS API Gateway console with a new dialog box for creating a resource. The URL path is /Methods. The dialog displays the message: "No methods defined for the resource." There is a red circle highlighting the slash character (/) in the URL path field, indicating where to click to create a new resource.

\*\*\*Repeat the same process for `/confirmemail`, `/login`, and `/search`\*\*\*

For `/photos`, a GET request a lambda function:

- Create a new **Resource** under the **Actions** dropdown button. Give a name to the resource and make sure that "Enable API Gateway CORS" is checked. Set the mapping template for all the method created below.

The screenshot shows the AWS Lambda function configuration page for the `/photos` resource. The left sidebar shows the API structure: `/` > `/confirmemail` > `/login` > `/search` > `/photos`. The right panel is titled "New Child Resource" and contains fields for "Resource Name" (set to "photos") and "Resource Path" (set to `/ photos`). A note explains that you can add path parameters using brackets. Below these fields is a checkbox for "Enable API Gateway CORS" which is checked. At the bottom are "Cancel" and "Create Resource" buttons.

- Once the resource is created, select the resource and create a new **Method** under that resource.

The screenshot shows the AWS Lambda function configuration page for the `/photos` resource. The left sidebar shows the API structure: `/` > `/confirmemail` > `/login` > `/search` > `/photos`. The right panel is titled "Actions" and shows a dropdown menu for "Methods". The "Create Method" option is highlighted. Other options in the dropdown include "Create Resource", "Enable CORS", "Edit Resource Documentation", and "Delete Resource". Below the dropdown, there is a note: "None Not required".

- Select **GET** as the method for this resource. Select the method created and attach the lambda function as the integration type.

The screenshot shows the AWS API Gateway console. On the left, there's a navigation sidebar with options like APIs, Custom Domain Names, VPC Links, and others. Under the API section, 'PhotoGalleryAPI' is selected. In the main area, the path 'Resources > /photos - GET - Setup' is shown. A dropdown menu on the left lists methods: /confirmemail (OPTIONS, POST), /login (OPTIONS, POST), /photos (GET, OPTIONS), /search (OPTIONS, POST), and /signup (OPTIONS, POST). The 'GET' option under '/photos' is highlighted. On the right, there's a configuration panel for the 'GET' method. It says 'Choose the integration point for your new method.' Below that, 'Integration type' is set to 'Lambda Function' (radio button selected). Other options include 'HTTP', 'Mock', 'AWS Service', and 'VPC Link'. There's also a 'Use Lambda Proxy Integration' checkbox, which is unchecked. The 'Lambda Region' is set to 'us-east-1'. Under 'Lambda Function', the name 'photogallery\_getphotos' is entered. At the bottom right is a blue 'Save' button.

- Under the /photos resource, create a new **POST** method. Select the method created and attach the lambda function as the integration type.

This screenshot shows the same AWS API Gateway interface as the previous one, but with a different focus. The path 'Resources > /photos Methods' is shown. A context menu is open over the '/photos' resource, with the 'Create Method' option highlighted and circled in red. Other options in the menu include 'Create Resource', 'Enable CORS', 'Edit Resource Documentation', and 'Delete Resource'. Below the menu, there are sections for 'API ACTIONS' (Deploy API, Import API, Edit API Documentation) and 'Delete API'. To the right, a detailed view of the new POST method is shown, including the API ID 'us-east-1:797770618644:function...', the stage 'None', and the note 'Not required'. At the bottom right of the main area, there's an 'API Key' section with the note 'Not required'.

The screenshot shows the AWS API Gateway console. On the left, the navigation pane is open with the 'APIs' section expanded, showing the 'PhotoGalleryAPI'. Under 'Resources', the '/photos' resource is selected. In the center, the 'Actions' dropdown is open, and the 'POST' option is highlighted. To the right, two method configurations are displayed: a 'GET' method and an 'OPTIONS' method. Both methods have 'arn:aws:lambda:us-east-1:797770618644:function...' as the ARN, 'None' for Authorization, and 'Not required' for API Key.

The screenshot shows the 'POST' method setup page for the '/photos' resource. The 'Integration type' is set to 'Lambda Function'. The 'Lambda Region' is 'us-east-1'. The Lambda function name is 'photogallery\_addphoto'. The 'Save' button is visible at the bottom right.

- After saving the integration type, select “**Integration Request**” under both get and post methods. Under the “**Mapping Template**”, choose “**When there are no templates defined (recommended)**” under the Request body passthrough. Under the **Content-Type**, click “**Add mapping**” and add “**application/json**” to the mapping. Finally, Under the **Generate template** dropdown, select “**Method Request passthrough**” and click “**Save**”.

- Under the /photos resource, create a new resource for /photos/{id}, as shown below. Under the new created resource, create a new **GET** method. Select the method created and attach the lambda function as the integration type.

The screenshots illustrate the step-by-step process of creating a child resource under the /photos resource and configuring its methods:

- Screenshot 1: Creating a Child Resource**  
Shows the 'Resources' section with the /photos resource selected. A new child resource named '/{id}' is being created under it. The 'Resource Path' is set to '/photos/{id}'.
- Screenshot 2: Defining Methods for the Child Resource**  
Shows the '/photos/{id}' resource selected. A new 'ANY' method is being created. The 'OPTIONS' method is also listed. The 'OPTIONS' method is selected, showing its configuration options.
- Screenshot 3: Setting Up the GET Method Integration**  
Shows the '/photos/{id}' - GET - Setup screen. The 'Integration type' is set to 'Lambda Function'. The 'Lambda Region' is 'us-east-1'. The 'Lambda Function' name is 'photogallery\_getphoto'. The 'Save' button is visible at the bottom right.

- After saving the integration type, select “**Integration Request**” under the method. Under the “**Mapping Template**”, choose “**When there are no templates defined (recommended)**” under the Request body passthrough. Under the **Content-Type**, click “**Add mapping**” and add “**application/json**” to the mapping. Finally, Under the **Generate template** dropdown, select “**Method Request passthrough**” and click “**Save**”.

- Finally, Create a new **Resource** at the root (“/”) to upload the photos to the S3 bucket. Give a name to the resource and make sure that “**Enable API Gateway CORS**” is checked

The screenshot shows the AWS API Gateway console. In the left sidebar, under the API named "PhotoGalleryAPI", the "Resources" section is selected. On the main page, the "Actions" dropdown for the root resource "/" is open, showing options like "Create Method", "Create Resource", "Enable CORS", and "Edit Resource Documentation". Below this, the "API ACTIONS" section includes "Deploy API", "Import API", "Edit API Documentation", and "Delete API".

In the second part of the screenshot, the "Create" dialog for a new child resource is displayed. The "Resource Name" field is set to "uploadphoto" and the "Resource Path" field is set to "/uploadphoto". The "Enable API Gateway CORS" checkbox is checked. The "Create Resource" button is visible at the bottom right.

- Within the same resource, create another resource as shown below.

This screenshot shows the continuation of the API creation process. The "Actions" dropdown for the "/uploadphoto" resource is open, and the "Create Resource" button is highlighted. The "Resource Name" field is set to "uploadphoto/{item}" and the "Resource Path" field is set to "/uploadphoto/{item}". The "Enable API Gateway CORS" checkbox is checked. The "Create Resource" button is visible at the bottom right.

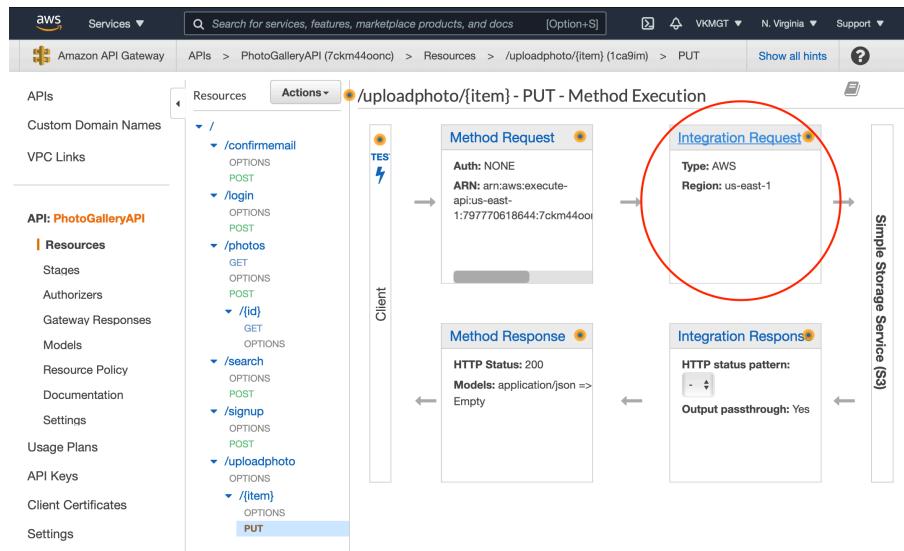
- Under the new created resource, create a new **PUT** method.

The screenshot shows the AWS API Gateway console. On the left, the navigation pane is open with the API named "PhotoGalleryAPI". Under the "Resources" section, there is a tree view of API endpoints. One endpoint, "/uploadphoto/{item}", has its "Actions" dropdown menu open, showing various HTTP methods: ANY, DELETE, GET, HEAD, PATCH, POST, and PUT. The "PUT" method is highlighted with a blue selection bar. To the right of the tree view, a modal window titled "OPTIONS" is displayed, showing the "Mock Endpoint" configuration with "Authorization: None" and "API Key: Not required". The top navigation bar includes the AWS logo, services dropdown, search bar, and account information for "N. Virginia".

- Select the method created and select the **AWS Region** where the S3 bucket was created (in this case, is **us-east-1** or **N. Virginia**). Under **AWS Service**, choose **Simple Storage Service (S3)**. Under **Action Type**, select "**Use path override**" and add the path override; in this case, it is the path where the photos are going to be stored in the bucket (**photobucket-corporan-2021-4813/photos/{object}**). Attach the execution role previously for the S3 bucket. Click "**Save**" to set the configuration.

The screenshot shows the "PUT - Setup" configuration page for the "/uploadphoto/{item}" method. The left sidebar lists the API resources. The main panel shows the configuration options for the "Integration type" (set to "AWS Service"), "AWS Region" (set to "us-east-1"), "AWS Service" (set to "Simple Storage Service (S3)"), "AWS Subdomain" (empty), "HTTP method" (set to "PUT"), "Action Type" (set to "Use path override"), "Path override (optional)" (set to "photobucket-corporan-2021-4813/photos/{object}"), "Execution role" (set to "arn:aws:iam::797770618644:role/apiS3PutGet-Role"), "Content Handling" (set to "Passthrough"), and "Use Default Timeout" (checkbox checked). A "Save" button is at the bottom right. The top navigation bar is identical to the previous screenshot.

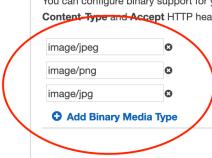
- Once the method is created, select the **Integration Request** under the method and create a mapping configuration as shown below.



The screenshot shows the 'Integration Request' configuration for the same PUT method. The 'Integration type' is set to 'AWS Service' (Simple Storage Service). The 'AWS Region' is 'us-east-1'. The 'AWS Service' is 'Simple Storage Service (S3)'. The 'AWS Subdomain' is empty. The 'HTTP method' is 'PUT'. The 'Path override' is 'photobucket-corporan-2021-4813/photos/{object}'. The 'Execution role' is 'arn:aws:iam::797770618644:role/apiS3PutGet-Role'. The 'Credentials cache' is set to 'Do not add caller credentials to cache key'. The 'Content Handling' is 'Passthrough'. The 'Use Default Timeout' checkbox is checked. The 'URL Path Parameters' section is expanded, showing a table with one row: 'Name' (object), 'Mapped from' (method.request.path.item), and 'Caching' (unchecked). A red circle highlights this table. Other sections like 'URL Query String Parameters', 'HTTP Headers', and 'Mapping Templates' are also visible.

Name	Mapped from	Caching
object	method.request.path.item	<input type="checkbox"/>

- To allow images to be uploaded to the S3 bucket, we need to add the binary media types. For this case, we are going to allow **JPEG**, **JPG**, and **PNG** to be uploaded to the bucket. To do that, select **Setting** in the left column and add the media types under **Binary Media Types**, as shown below



The screenshot shows two screenshots of the AWS API Gateway interface. The top screenshot shows the 'Resources' page for the 'PhotoGalleryAPI'. The bottom screenshot shows the 'Settings' page for the same API. A red circle highlights the 'Binary Media Types' section on the Settings page.

**Top Screenshot: Resources Page**

- APIs > PhotoGalleryAPI > Resources > / (v0nw2x12z)
- Actions: / Methods
- No methods defined for the resource.
- API: PhotoGalleryAPI
  - Resources
  - Stages
  - Authorizers
  - Gateway Responses
  - Models
  - Resource Policy
  - Documentation
  - Settings
  - Usage Plans
  - API Keys
  - Client Certificates
  - Settings

**Bottom Screenshot: Settings Page**

- APIs > PhotoGalleryAPI > Settings
- Actions: / Settings
- Configure settings for your API deployments. Configure Tags
- General Settings**
  - Update the name and description for your API.
  - Name: PhotoGalleryAPI
  - Description: [Text area]
- Endpoint Configuration**
  - Specify the endpoint type for your API. For Private APIs, you can associate one or more VPC endpoints with your API and API Gateway will generate new Route 53 Alias records which you can use to invoke your API.
  - Endpoint Type: Regional
- Default Endpoint**
  - The default execute-api endpoint generated by API Gateway: <https://7ckm44oconc.execute-api.us-east-1.amazonaws.com>. To allow clients to access your API only by using a custom domain name, disable the default endpoint. This is an API level setting and affects all stages of your API. After you enable or disable the default endpoint, deploy your API to any one stage for the update to take effect. [Learn more](#).
  - Default endpoint:  Enabled  Disabled
- API Key Source**
  - Choose the source of your API Keys from incoming requests. Configure deployments to receive API keys from the x-api-key header or from a Lambda Authorizer.
  - API Key Source: HEADER
- Content Encoding**
  - Allow compression of response bodies based on client's Accept-Encoding header. Compression is triggered when response body size is greater than or equal to your configured threshold. The maximum body size threshold is 10 MB (10,485,760 Bytes). The following compression types are supported: gzip, deflate, and identity.
  - Content Encoding enabled:
- Binary Media Types**
  - You can configure binary support for your API by specifying which media types should be treated as binary types. API Gateway will look at the Content-Type and Accept HTTP headers to decide how to handle the body.
  - image/jpeg
  - image/png
  - image/jpg
  - Add Binary Media Type**

**Save Changes**

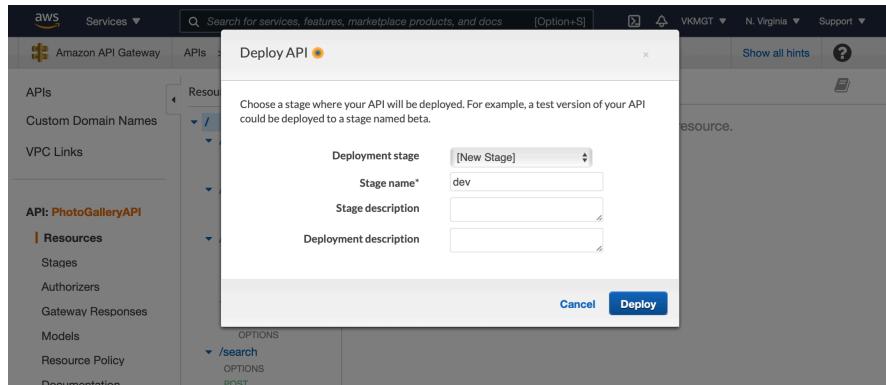
- Make sure to **enable CORS** for the resources. To do that, select the resources and click the **Actions** dropdown and select “**Enable CORS**”.

The screenshot shows the AWS Lambda console interface. On the left, there's a sidebar with options like APIs, Custom Domain Names, VPC Links, and API: PhotoGalleryAPI. Under API: PhotoGalleryAPI, there are sections for Resources, Stages, Authorizers, Gateway Responses, Models, Resource Policy, Documentation, Dashboard, Settings, Usage Plans, API Keys, Client Certificates, and Settings. The main area shows a tree structure of resources: /, /confirm (with OPTIONS, POST methods), /login (with OPTIONS, POST methods), /photos (with GET, OPTIONS, POST methods), /{id} (with GET, OPTIONS methods), /search (with OPTIONS, POST methods), /signup (with OPTIONS, POST methods), and /uploadphoto (with OPTIONS, POST methods). A context menu is open over the /confirmresource, showing actions: Create Method, Create Resource, Enable CORS, Edit Resource Documentation, and Delete Resource. The 'Enable CORS' option is highlighted. Below the menu, it says 'None' under 'API ACTIONS' and 'Not required' under 'API Key'. The URL in the browser bar is: APIs > PhotoGalleryAPI (7ckm44oconc) > Resources > /confirmemail (eof9k).

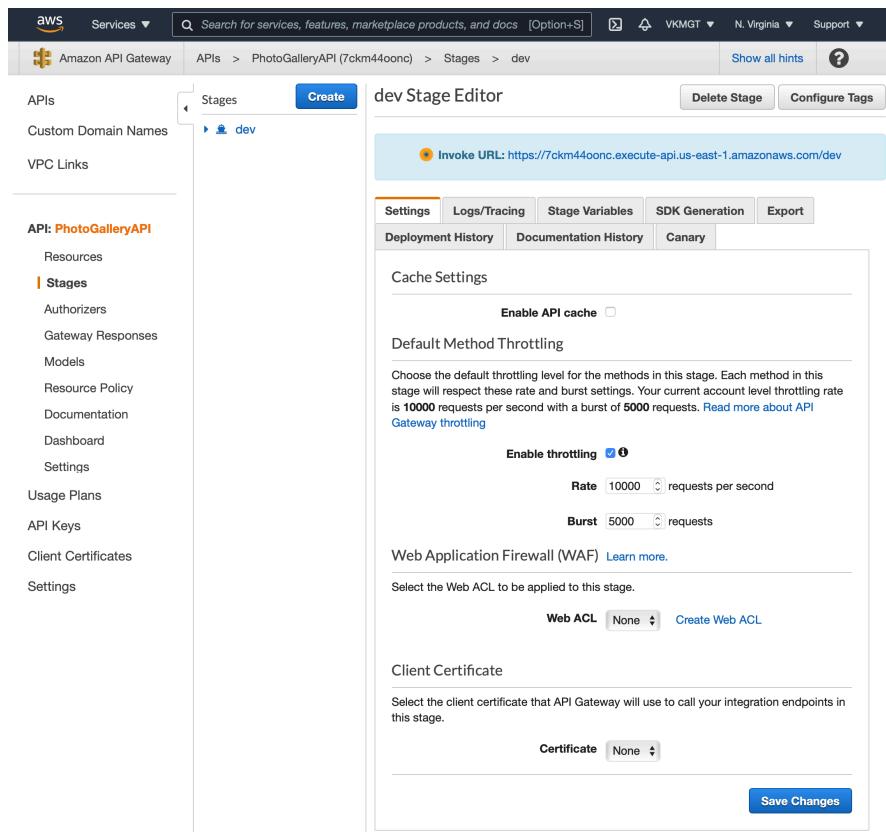
- To deploy the API, click “**Deploy API**”, as shown below

This screenshot shows the AWS Lambda console interface, similar to the previous one but for the root resource. The sidebar and API list are identical. The main area shows the same tree structure of resources. A context menu is open over the /resource, showing actions: Create Method, Create Resource, Enable CORS, and Edit Resource Documentation. The 'Enable CORS' option is highlighted. Below the menu, it says 'None' under 'RESOURCE ACTIONS' and 'Not required' under 'API Key'. The URL in the browser bar is: APIs > PhotoGalleryAPI (7ckm44oconc) > Resources > / (v0nw2xi2z).

- Create a new stage for deployment and called it “**dev**” as shown below

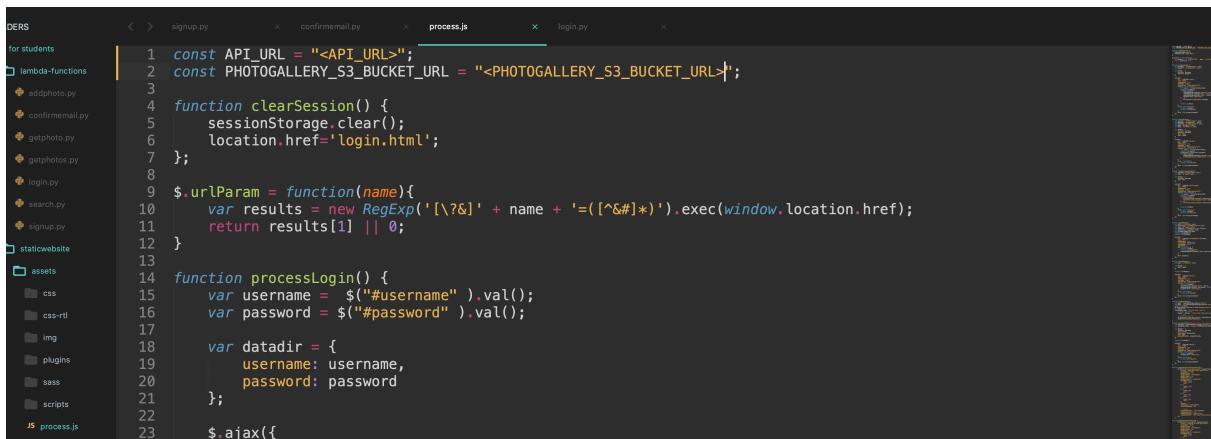


- Copy and save the API URL; will use this URL to access the resources from the website.



## 9. Updating the website

Add some resources to script used for the website. Update the **process.js** file under the staticwebsite/assets directory. Add the **<API\_URL>** and **<PHOTOGALLERY\_S3\_BUCKET\_NAME>** in lines 1 and 2, respectively, as shown below



```
for students
  for lambda-functions
    addphoto.py
    confirmemail.py
    getphoto.py
    getphotos.py
    login.py
    search.py
    signus.py
  staticwebsite
    assets
      css
      css-rtl
      img
      plugins
      sass
      scripts
    process.js
```

```
1 const API_URL = "<API_URL>";
2 const PHOTOGALLERY_S3_BUCKET_URL = "<PHOTOGALLERY_S3_BUCKET_URL>";
3
4 function clearSession() {
5   sessionStorage.clear();
6   location.href='login.html';
7 };
8
9 $.urlParam = function(name){
10   var results = new RegExp('(\?&)'+ name + '=([^&#]*)').exec(window.location.href);
11   return results[1] || 0;
12 }
13
14 function processLogin() {
15   var username = $("#username").val();
16   var password = $("#password").val();
17
18   var datadir = {
19     username: username,
20     password: password
21   };
22
23   $.ajax({
```

After making the updates, upload the files from the 'staticwebsite' folder to the S3 bucket for the website.

## 10. Access the Photo Gallery application in a browser

- Access the URL of the static website for photo gallery application hosted on S3.
- Goto signup page and create a new user.
- Confirm the user's email.
- Goto login page and login using the user created above.
- Goto add photos page and add a new photo.
- Add a more photos and try browsing and searching for photos.

## Challenges:

1. Competition of the main functionalities of the website as described in the instructions **(80 points)**
2. Allow users to delete a picture in their own gallery **(10 Points)**
3. Allow users to update the title, description and tags of the pictures after creation **(10 Points)**

## Deliverables:

A video that shows the functions of your website, including user signup/login/logout, upload a picture, view pictures under the same tag. If you have completed the challenge tasks, also be sure to include the following example: a deletion of a picture, search for a picture, update of a picture's information, etc.. You can also show the database to prove that the picture has been deleted or modified.

## Hints on Debugging Process

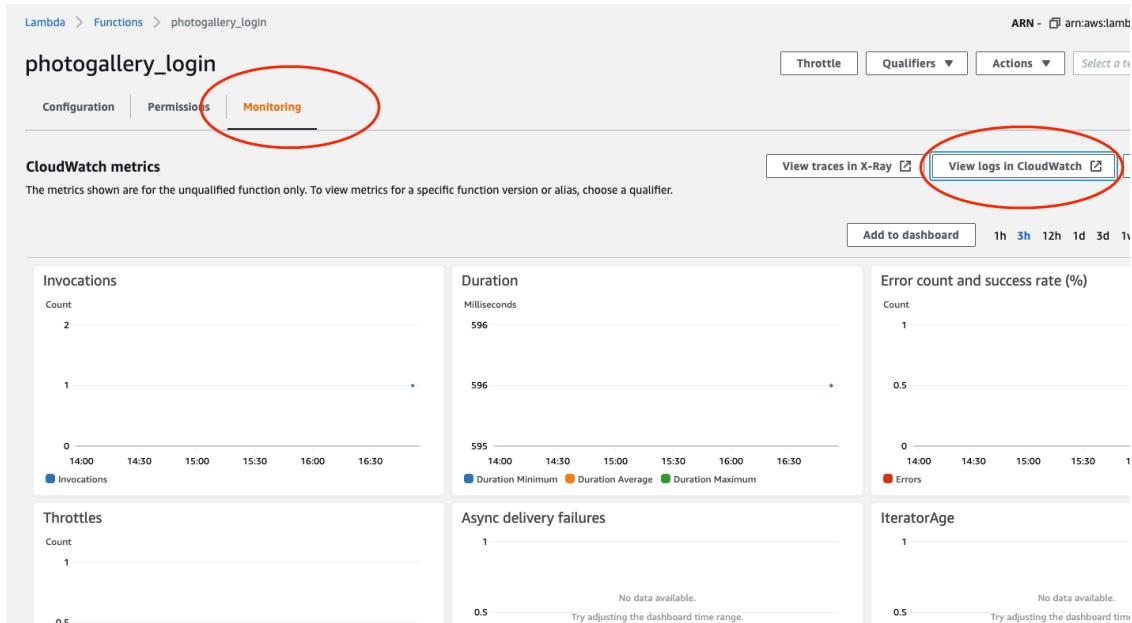
1. Test your code on localhost before uploading to cloud

```
[11:20] (/>w</) ~ $ cd /Users/jolinchen/Desktop/Spring_2021/TA_ECE_4150/Lab1/staticwebsite
[11:20] (/>w</) ~/Desktop/Spring_2021/TA_ECE_4150/Lab1/staticwebsite $ http-server
[Starting up http-server, serving ./
Available on:
  http://127.0.0.1:8080
  http://10.0.0.12:8080
Hit CRTL-C to stop the server
```

Go to the root folder of the staticwebsite, type in command line `http-server` to set up HTTP server for testing. You can find the installation information of `http-server` at: <https://www.npmjs.com/package/http-server>. Then open the url as indicated in your terminal.

2. Use CloudWatch to monitor data flow

This is to locate problems occurring in lambda functions. Go to your lambda function list and choose any lambda function, go to “**Monitoring**” → “**view logs in CloudWatch**”.



You have to create a log group in order to get the logs. To so do, in your newly opened CloudWatch Management Console page, copy the link above (in this case: group: `/aws/lambda/photogallery_login`):

Go back to Log groups, choose “**Create log group**”, in Log group name, paste the link you just copied then create:

**☒ Log group does not exist**  
The specific log group: `/aws/lambda/photogallery_login` does not exist in this account or region.

CloudWatch > CloudWatch Logs > Log groups > Create log group

### Create log group

**Log group details**

Log group name: /aws/lambda/photogallery\_login

Retention setting: Never expire

KMS key ARN - optional:

**Create**

In your newly created log group, click on the dropdown button of “**Log group details**” and copy the ARN in it:

CloudWatch > CloudWatch Logs > Log groups > /aws/lambda/photogallery\_login

Log group details			
Retention Never expire	Creation time 26 minutes ago	Stored bytes -	ARN <code>arn:aws:logs:us-east-1:610172127508:log-group:/aws/lambda/photogallery_login:*</code>
KMS key ID -	Metric filters 0	Subscription filters 0	Contributor Insights rules -

Now we need to grant write access to the log group. Go to IAM console and open roles. Go to **lambda\_photogallery\_role** you created before and choose “Attach policies”:

Permissions   Trust relationships   Tags   Access Advisor   Revoke sessions

▼ Permissions policies (3 policies applied)

**Attach policies**   **Add inline policy**

Policy name	Policy type	Actions
▶  AmazonDynamoDBFullAccess	AWS managed policy	✖
▶  AmazonCognitoPowerUser	AWS managed policy	✖

Choose “**Create policy**”. In Service, choose “**CloudWatch Logs**”. In Actions - Write, click on the dropdown list and choose as indicated below.

**Access level**

- ▶  List
- ▶  Read
- ▼  Write (3 selected)

<input type="checkbox"/> AssociateKmsKey	<input type="checkbox"/> DeleteLogStream	<input type="checkbox"/> PutMetricFilter
<input type="checkbox"/> CancelExportTask	<input type="checkbox"/> DeleteMetricFilter	<input type="checkbox"/> PutResourcePolicy
<input type="checkbox"/> CreateExportTask	<input type="checkbox"/> DeleteResourcePolicy	<input type="checkbox"/> PutRetentionPolicy
<input type="checkbox"/> CreateLogDelivery	<input type="checkbox"/> DeleteRetentionPolicy	<input type="checkbox"/> PutSubscriptionFilter
<input checked="" type="checkbox"/> CreateLogGroup	<input type="checkbox"/> DeleteSubscriptionFilter	<input type="checkbox"/> TagLogGroup
<input checked="" type="checkbox"/> CreateLogStream	<input type="checkbox"/> DisassociateKmsKey	<input type="checkbox"/> UntagLogGroup
<input type="checkbox"/> DeleteDestination	<input type="checkbox"/> PutDestination	<input type="checkbox"/> UpdateLogDelivery
<input type="checkbox"/> DeleteLogDelivery	<input type="checkbox"/> PutDestinationPolicy	
<input type="checkbox"/> DeleteLogGroup	<input checked="" type="checkbox"/> PutLogEvents	

[Expand all](#) | [Collapse all](#)

In Resources, choose “**Specific**”, click on “**Add ARN**” in log-group and paste your copied ARN into it. After finishing everything, check on your json file, which should look like the screenshot below. Click on “**Review Policy**”, name your new policy as you like and create. Attach the policy to your role after creation.

The screenshot shows the 'Add ARN(s)' dialog box from the AWS Lambda console. The 'Specify ARN for log-group' input field is highlighted with a red arrow. Below the dialog, a JSON policy document is displayed:

```

1  {
2      "Version": "2012-10-17",
3      "Statement": [
4          {
5              "Sid": "VisualEditor0",
6              "Effect": "Allow",
7              "Action": [
8                  "logs:CreateLogStream",
9                  "logs:CreateLogGroup",
10                 "logs:PutLogEvents"
11             ],
12             "Resource": "arn:aws:logs:us-east-1:610172127508:log-group:/aws/lambda/photogallery_login
13             ":"*"
14         }
15     ]
}

```

Now as you enter your log group of chosen lambda function, you should be able to monitor and examine all the activities.

/aws/lambda/photogallery\_login

Actions ▾ View in Logs Insights Search log group

► Log group details

Log streams Metric filters Subscription filters Contributor Insights

Log streams (1)

Filter log streams or try prefix search

Log stream Last event time

2021/01/25/[\$LATEST]7d58d95e577b46c18c418c01a75b4246 2021-01-25 12:53:53 (UT...)

CloudWatch > CloudWatch Logs > Log groups > /aws/lambda/photogallery\_login > 2021/01/25/[\$LATEST]7d58d95e577b46c18c418c01a75b4246

Try CloudWatch Logs Insights Try Logs Insights X

CloudWatch Logs insights allows you to search and analyze your logs using a new, purpose-built query language. To learn more, read the AWS blog or visit our documentation.

Log events

You can use the filter bar below to search for and match terms, phrases, or values in your log events. Learn more about filter patterns

View as text Actions Create Metric Filter

Filter events Clear 1m 30m 1h 12h Custom

Timestamp	Message
No older events at this moment. <a href="#">Retry</a>	
2021-01-25T12:53:52.475-05:00	START RequestId: eb9ad9ca-18fe-47dc-9a5d-21f0493136f2 Version: \$LATEST
2021-01-25T12:53:53.052-05:00	END RequestId: eb9ad9ca-18fe-47dc-9a5d-21f0493136f2
2021-01-25T12:53:53.052-05:00	REPORT RequestId: eb9ad9ca-18fe-47dc-9a5d-21f0493136f2 Duration: 576.11 ms Billed Durat...
No newer events at this moment. <a href="#">Auto retry paused. Resume</a>	