

GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 4150-A Fall 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 October 1, 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

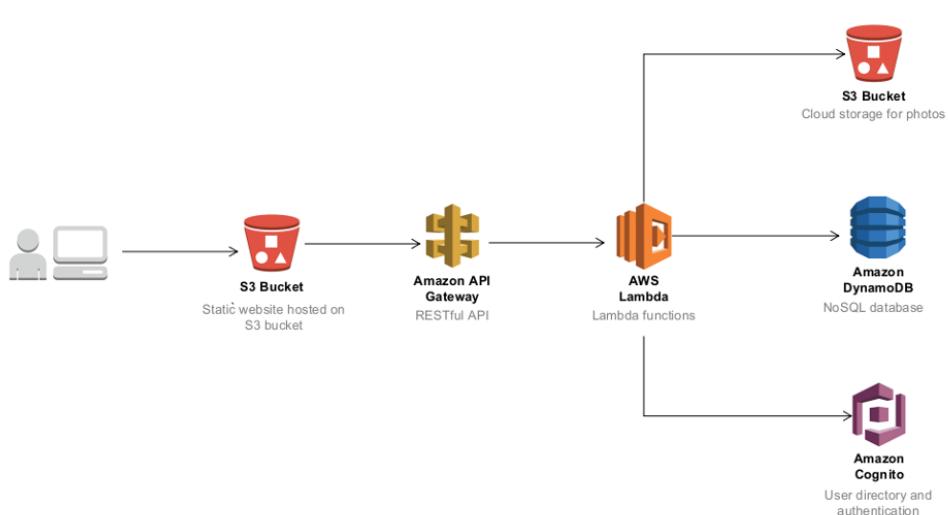


Fig. Architecture diagram of the Photo Gallery application showing AWS services used

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-2021-4150**)
- 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-2021-4150**)
- 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 message: 'Kinesis Data Streams for DynamoDB is now available. You now can capture item-level changes in your DynamoDB tables as a Kinesis data stream and start taking advantage of Kinesis services to build advanced streaming applications.' Below the message, the main form has a 'Table name*' field containing 'PhotoGallery'. Under 'Primary key*', there's a 'Partition key' section with a 'PhotoID' field set to 'String'. A checked checkbox 'Add sort key' is followed by a 'CreationTime' field set to 'Number'. On the right side of the form, there are 'Tutorial' and '?' buttons.

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: Name (highlighted in yellow), Attributes, Policies, MFA and verifications, Message customizations, Tags, Devices, App clients, Triggers, and Review. The main area has a heading 'What do you want to name your user pool?'. It asks for a descriptive name and shows 'PhotoGalleryUserPool' in the 'Pool name' field. Below it, a heading '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'. On the right, there's a 'Cancel' button.

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 content area is titled "Which app clients will have access to 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, Triggers, and Review. The 'Review' option is selected and highlighted with a yellow background. 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 section.

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. The top navigation bar includes the AWS logo, Services dropdown, search bar ('Search for services, features, marketplace products, and docs [Option+S]'), and account information ('VKMGT N. Virginia Support'). Below the navigation is the 'User Pools' section header ('Federated Identities') and the pool name 'PhotoGalleryUserPool'. On the left, a sidebar lists various settings: General settings, Users and groups, Attributes, Policies, MFA and verifications, Advanced security, Message customizations, Tags, Devices, App clients (which is selected and highlighted in orange), Triggers, Analytics, and App integration. The main content area is titled 'Which app clients will have access to this user pool?' and contains a note: '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 is 'MyCloudApp' and the second column is 'App client id' containing '5gpqla43t691ujeain71aiibo'. A red oval highlights the 'App client id' field. At the bottom of the modal is a 'Show Details' button.

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. The top navigation bar includes the AWS logo, Services dropdown, search bar ('Search for services, features, marketplace products, and docs [Option+S]'), and account information ('VKMGT Global Support'). The process is at 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' shows 'Common use cases': 'EC2' (selected) and 'Lambda'. A red oval highlights the 'Lambda' option. Step 3: 'Common use cases' shows 'EC2' (selected). Step 4: 'Or select a service to view its use cases' lists various services: API Gateway, CloudWatch Events, EKS, IoT Things Graph, Redshift; AWS Backup, CodeBuild, EMR, KMS, Rekognition; AWS Chatbot, CodeDeploy, ElastiCache, Kinesis, RoboMaker. At the bottom right are numbered circles 1, 2, 3, 4.

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

AWS service EC2, Lambda and others	Another AWS account Belonging to you or 3rd party	Web identity Cognito or any OpenID provider	SAML 2.0 federation 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

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

Create role

Review

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

Role name* apiS3PutGet-Role
Use alphanumeric and '+,-,@-_.' characters. Maximum 64 characters.

Role description 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 **Delete 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
AmazonS3FullAccess	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**

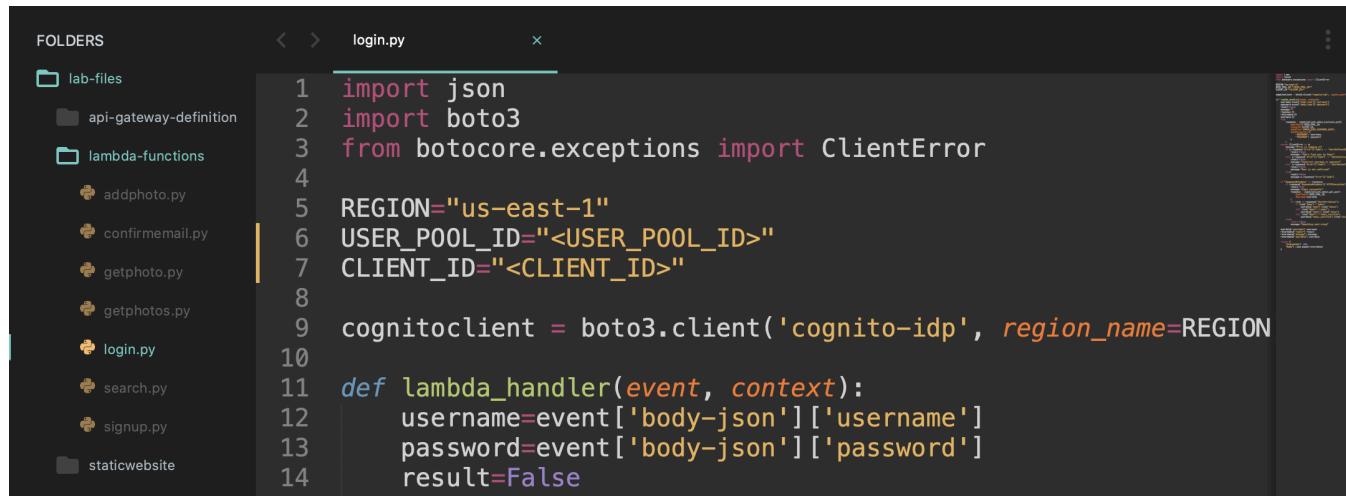
▼ Permissions policies (2 policies applied)

Attach policies **Add inline policy**

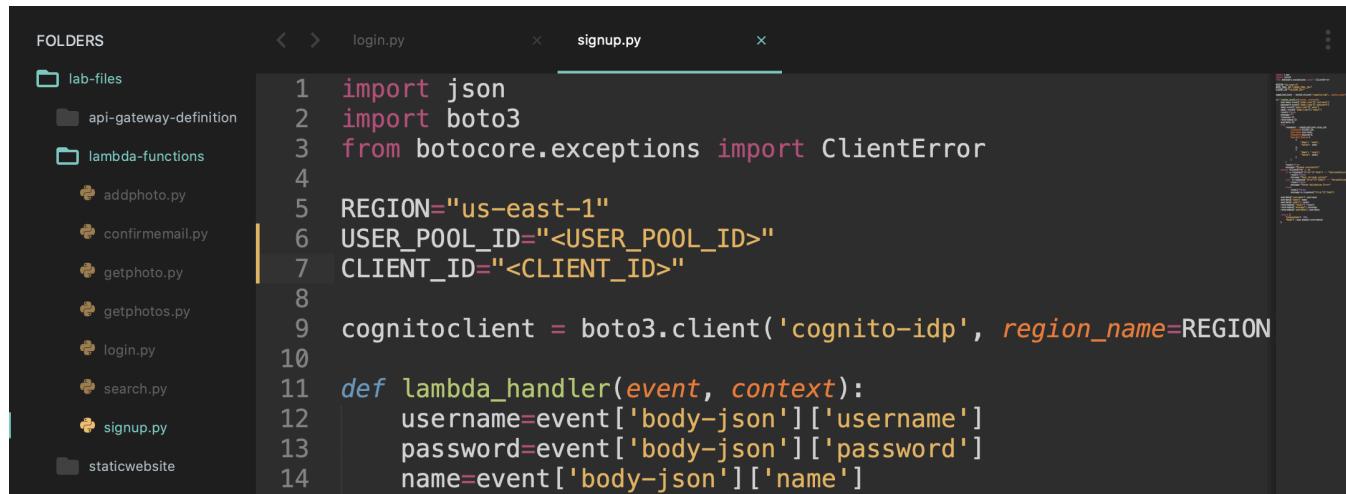
Policy name	Policy type
AmazonS3FullAccess	AWS managed policy
AmazonAPIGatewayPushTo...	AWS managed policy

6. Review Code

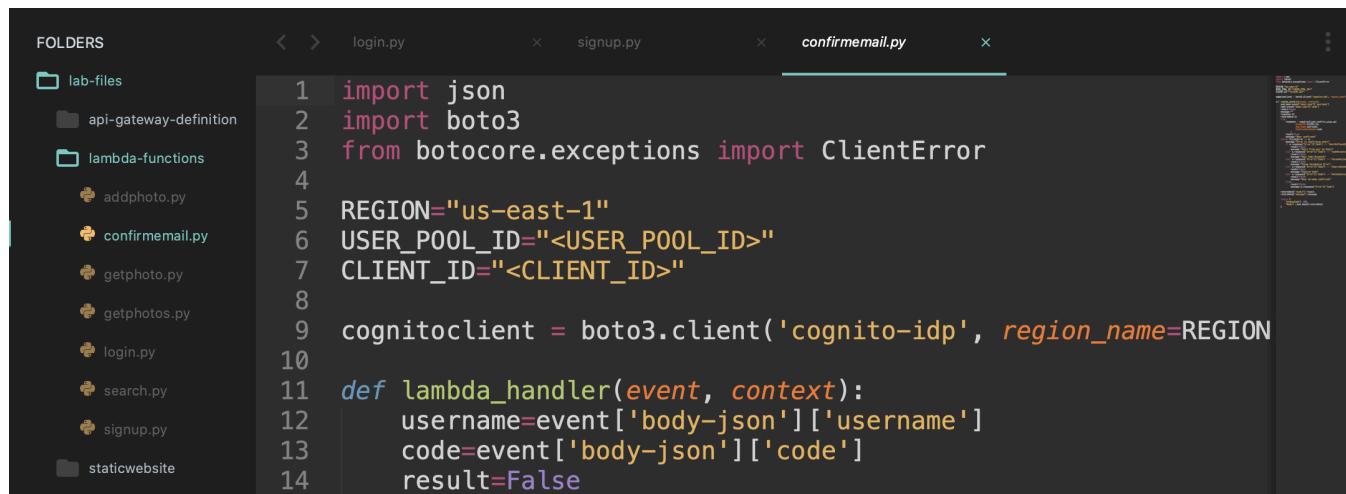
Under the **lambda-functions** directory, open **login.py**, **signup.py**, and **confirmemail.py**; update the <USER_POOL_ID> and the <CLIENT_ID>, in lines 6 and 7, respectively, with the ones you just created.



```
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']
```



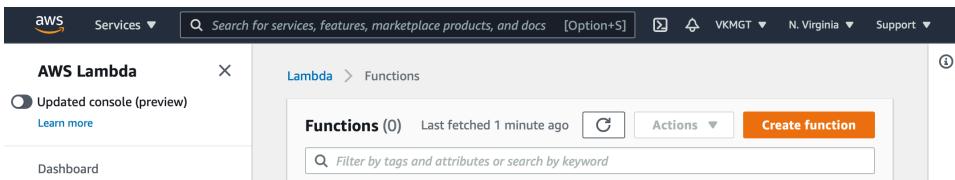
```
1 import json
2 import boto3
3 from botocore.exceptions import ClientError
4
5 REGION="us-east-1"
6 USER_POOL_ID=<USER_POOL_ID>
7 CLIENT_ID=<CLIENT_ID>
8
9 cognitoclient = boto3.client('cognito-idp', region_name=REGION)
10
11 def lambda_handler(event, context):
12     username=event['body-json']['username']
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. At the top, there are three tabs: "photogallery_confirmemail - Lambda", "User Pools - Amazon Cognito", and "photogallery_signup - Lambda". The middle tab is active. Below the tabs, the navigation bar shows "Lambda > Functions > photogallery_signup". The ARN is listed as "arn:aws:lambda:us-east-1:797770618644:function:photogallery_signup". The main area is titled "photogallery_signup" and contains a "Designer" section with a single function icon labeled "photogallery_signup". Below the Designer is a "Function code" editor. The code editor has tabs for "File", "Edit", "Find", "View", "Go", "Tools", "Window", "Test", and "Deploy". The "File" tab is selected. The code itself is a Python script named "lambda_function.py". The code defines a lambda_handler function that signs up a user in a Cognito pool using the boto3 library. It sets REGION to "us-east-1", USER_POOL_ID to "us-east-1_Ku6tG6Hqo", and CLIENT_ID to "5gpqla43t691ujeain7laiibo". It then creates a cognitoClient and calls sign_up with the provided event data. The response is returned as JSON.

```

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

**** Follow the same image examples as above.**

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**"
- 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**"
- After creating the function, under "**Function Code**" copy and paste the code provided.
- 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**"
- After creating the function, under "**Function Code**" copy and paste the code provided.
- 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**"
- After creating the function, under "**Function Code**" copy and paste the code provided.
- 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**"
- After creating the function, under "**Function Code**" copy and paste the code provided.
- 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**"
- After creating the function, under "**Function Code**" copy and paste the code provided.
- Click "**Deploy**" to release the function

8. Create a new API from API Gateway

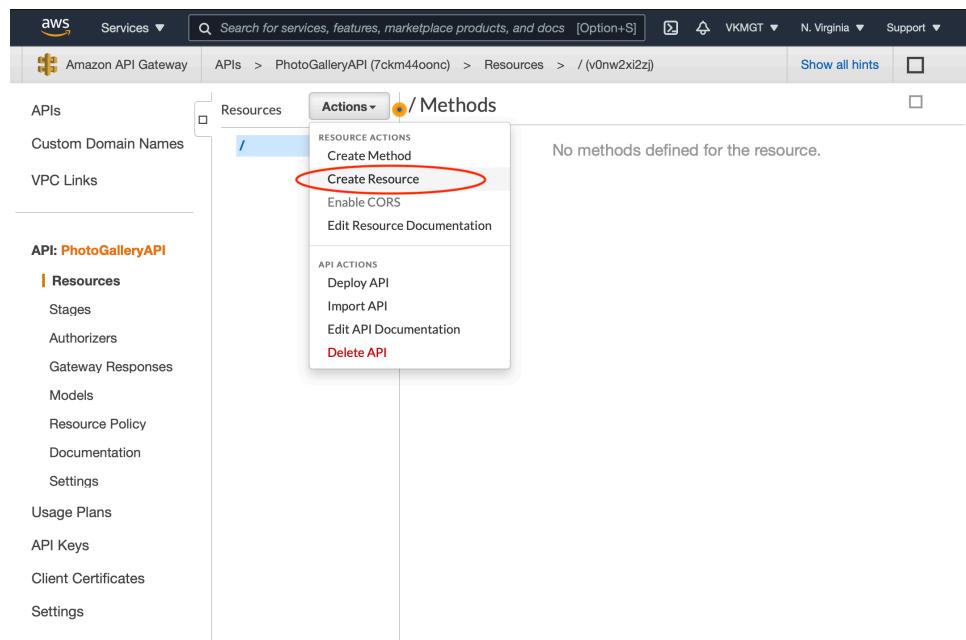
Go to the AWS API Gateway Console. Once you enter the console, it will provide you with an option to create an Example API with some resources and method; disregard that option and select “**New API**”. Create a new API called **PhotoGalleryAPI** with the resources, methods, and integration shown below.

The screenshot shows the 'Create new API' page in the AWS API Gateway console. At the top, there are navigation links for 'Services', a search bar, and account information. Below the header, the path 'Amazon API Gateway > APIs > Create' is shown. A radio button for 'New API' is selected. The 'Settings' section allows setting the API name (set to 'PhotoGalleryAPI'), description (empty), and endpoint type (set to 'Regional'). A note at the bottom left says '* Required'. On the right, a blue 'Create API' button is visible.

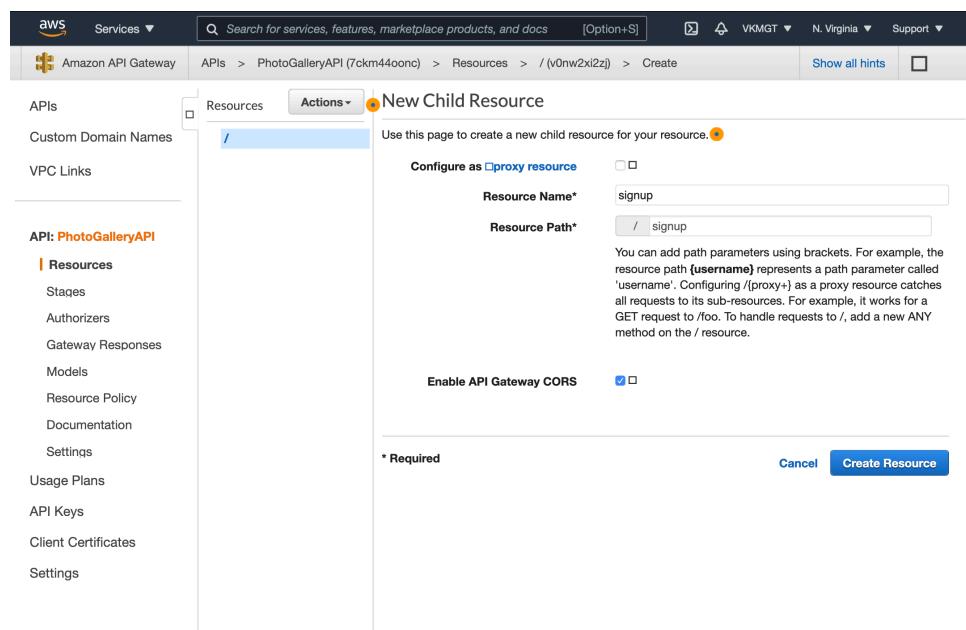
Resource	Method	Integration Request
/signup	POST	Lambda Function: photogallery_signup
/confirmemail	POST	Lambda Function: photogallery_confirmemail
/login	POST	Lambda Function: photogallery_login
/photos	POST	Lambda Function: photogallery_addphoto
/photos	GET	Lambda Function: photogallery_getphotos
/photos/{id}	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



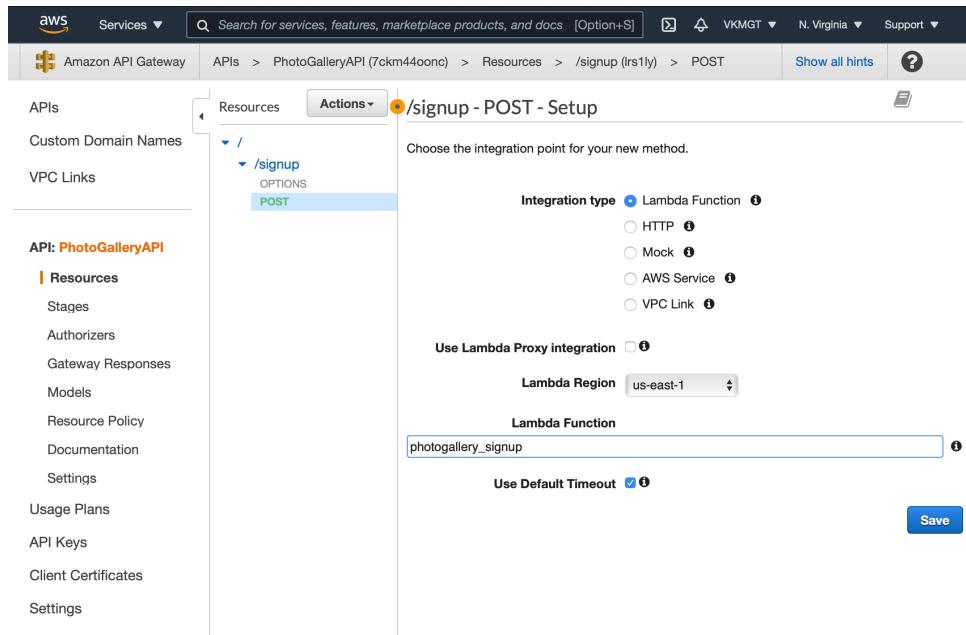
- Once the resource is created, select the resource and create a new **Method** under that resource.

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, a resource named '/signup' is selected. A context menu is open over this 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.

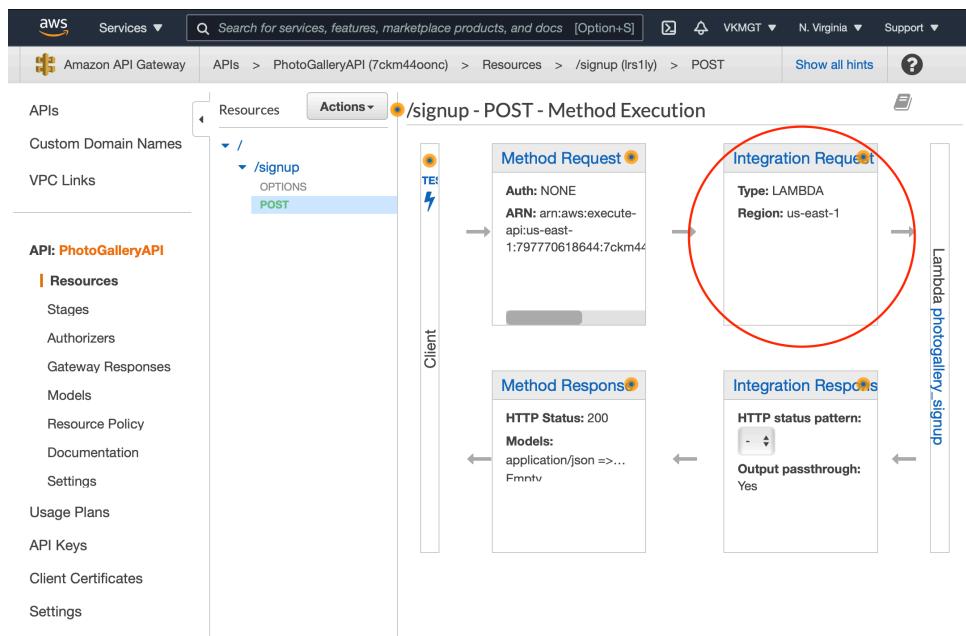
- Select **POST** as the method for this resource

The screenshot shows the AWS API Gateway console with the same interface as the previous one. The '/signup' resource is selected, and its methods are listed. The 'OPTIONS' method is currently selected. A sub-menu is open, showing the available HTTP methods: ANY, DELETE, GET, HEAD, PATCH, POST, and PUT. The 'POST' option is highlighted with a red circle. To the right, the configuration for the 'POST' method is shown, including 'Mock Endpoint' and 'Authorization' settings (None). The 'API Key' setting is noted as 'Not required'.

- 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 interface for configuring a POST method on the '/signup' resource of the 'PhotoGalleryAPI'. The 'Integration type' is set to 'Lambda Function'. The Lambda Region is 'us-east-1' and the Lambda Function is 'photogallery_signup'. The 'Content-Type' mapping is set to 'application/json'. A mapping template is shown in the code editor.

```

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 $param.keySet())
12          "#foreach($paramName in $param.keySet())
13            $paramName : $param.get($paramName)
14          "
15        "
16      }
17    }
18  }
19 }

```

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

The screenshot shows the AWS API Gateway interface for managing resources. The '/' resource has its Methods tab selected, showing that no methods are defined for it.

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 interface. On the left, there's a navigation sidebar with options like APIs, Custom Domain Names, VPC Links, and others. Under 'API: PhotoGalleryAPI', there's a 'Resources' section with links to Stages, Authorizers, Gateway Responses, Models, Resource Policy, Documentation, Settings, Usage Plans, API Keys, Client Certificates, and Settings. The main area shows a tree structure of resources: /, /confirmemail, /login, /photos (which is selected), /search, and /signup. Under /photos, there are methods: OPTIONS, POST, GET (selected), and OPTIONS. A modal window titled '/photos - GET - Setup' is open, showing the configuration for the GET method. It has a 'Choose the integration point for your new method.' section, an 'Integration type' section where 'Lambda Function' is selected, a 'Lambda Region' dropdown set to 'us-east-1', a 'Lambda Function' input field containing 'photogallery_getphotos', and a 'Save' button at the bottom right.

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

The screenshot shows the AWS API Gateway interface. The left sidebar is identical to the previous screenshot. The main area shows the same tree structure of resources. A context menu is open over the /photos resource node. The 'Create Method' option is highlighted and circled in red. Other options in the menu are: Create Resource, Enable CORS, Edit Resource Documentation, Delete Resource, Deploy API, Import API, Edit API Documentation, and Delete API. The menu also includes sections for RESOURCE ACTIONS and API ACTIONS.

The screenshot shows the AWS API Gateway console. On the left, the navigation pane is visible with various options like APIs, Stages, Authorizers, etc. The main area is titled "/photos Methods". It shows two methods: "GET" and "OPTIONS". The "GET" method is configured with "arn:aws:lambda:us-east-1:797770618644:function:..." as the ARN, "None" for Authorization, and "Not required" for API Key. The "OPTIONS" method is also configured with "None" for Authorization and "Not required" for API Key.

The screenshot shows the "POST" method setup page. The left sidebar lists other methods like GET, OPTIONS, and others. The right panel has a heading "Choose the integration point for your new method." It shows the "Integration type" section with "Lambda Function" selected (radio button is checked). Other options like "HTTP", "Mock", "AWS Service", and "VPC Link" are available but not selected. Below this, there's a "Use Lambda Proxy integration" checkbox which is unchecked. The "Lambda Region" dropdown is set to "us-east-1". The "Lambda Function" input field contains "photogallery_addphoto". At the bottom right, there is a "Save" button.

- 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 and configuring its methods in the AWS API Gateway:

- Screenshot 1: Creating a Child Resource**
Shows the "New Child Resource" page where a new resource is being created under the /photos resource. The "Resource Name" is set to "/photo/{id}" and the "Resource Path" is set to "/photos/{id}". A note explains that path parameters can be used to catch all requests to its sub-resources.
- Screenshot 2: Defining Methods for the Child Resource**
Shows the "/photos/{id} Methods" page. A new "GET" method is being defined under the "/{id}" resource. The "OPTIONS" method is also listed.
- Screenshot 3: Setting Up the GET Method Integration**
Shows the "/photos/{id} - GET - Setup" page. The "Integration type" is selected as "Lambda Function". The "Lambda Region" is set to "us-east-1" and 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.

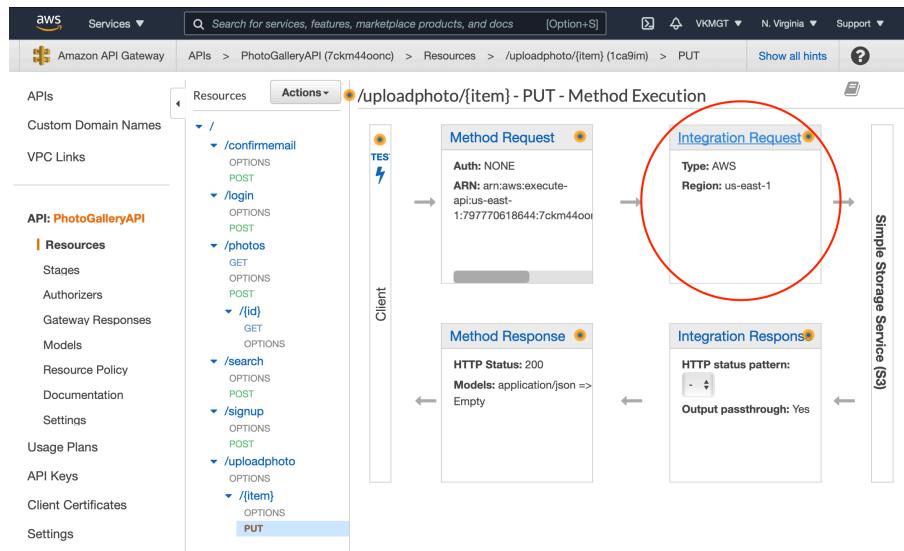
- Under the new created resource, create a new **PUT** method.

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 resources and methods. A context menu is open over the "PUT" method under the "/{item}" resource, with "PUT" selected. On the right, a modal window titled "OPTIONS" shows the "Mock Endpoint" configuration, which includes "Authorization: None" and "API Key: Not required".

- 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 various resources and methods. The main area shows the configuration for the PUT method. Under "Integration type", "AWS Service (Simple Storage Service (S3))" is selected. The "AWS Region" is set to "us-east-1". The "Action Type" is set to "Use path override", and the "Path override (optional)" field contains the value "photobucket-corporan-2021-4813/photos/{object}". Other settings include "Execution role" (set to "arn:aws:iam::797770618644:role/api/S3PutGet-Role"), "Content Handling" (set to "Passthrough"), and "Use Default Timeout" (checked). A "Save" button is at the bottom right.

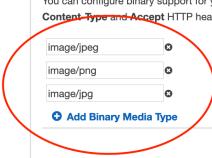
- Once the method is created, select the **Integration Request** under the method and create a mapping configuration as shown below.



The screenshot shows the 'Integration Request' configuration for the same PUT method. The 'Integration type' is set to 'AWS Service' (Simple Storage Service S3). The 'URL Path Parameters' section is highlighted with a red circle, showing a mapping for 'object' to 'method.request.path.item'. 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"/> <input checked="" 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**

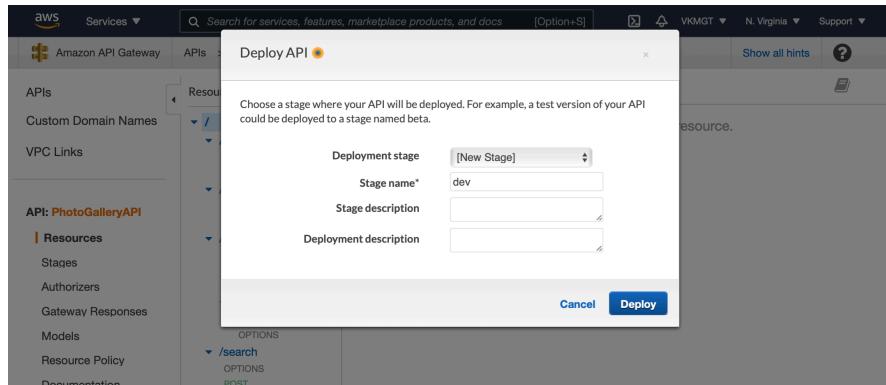
- Make sure to **enable CORS** for the resources. To do that, select the resources and click the **Actions** dropdown and select “**Enable CORS**”.

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 (OPTIONS, POST), /login (OPTIONS, POST), /photos (GET, OPTIONS, POST), /{id} (GET, OPTIONS), /search (OPTIONS, POST), /signup (OPTIONS, POST), /uploadphoto (OPTIONS, /{item} (OPTIONS, PUT)). A context menu is open over the /confirmemail method, specifically for the /confirmresource. The menu has two sections: RESOURCE ACTIONS (Create Method, Create Resource, Enable CORS, Edit Resource Documentation, Delete Resource) and API ACTIONS (Deploy API, Import API, Edit API Documentation, Delete API). The 'Enable CORS' option under RESOURCE ACTIONS is highlighted with a red box. Below the menu, it says 'None' under 'CORS' and 'Not required' under 'API Key'.

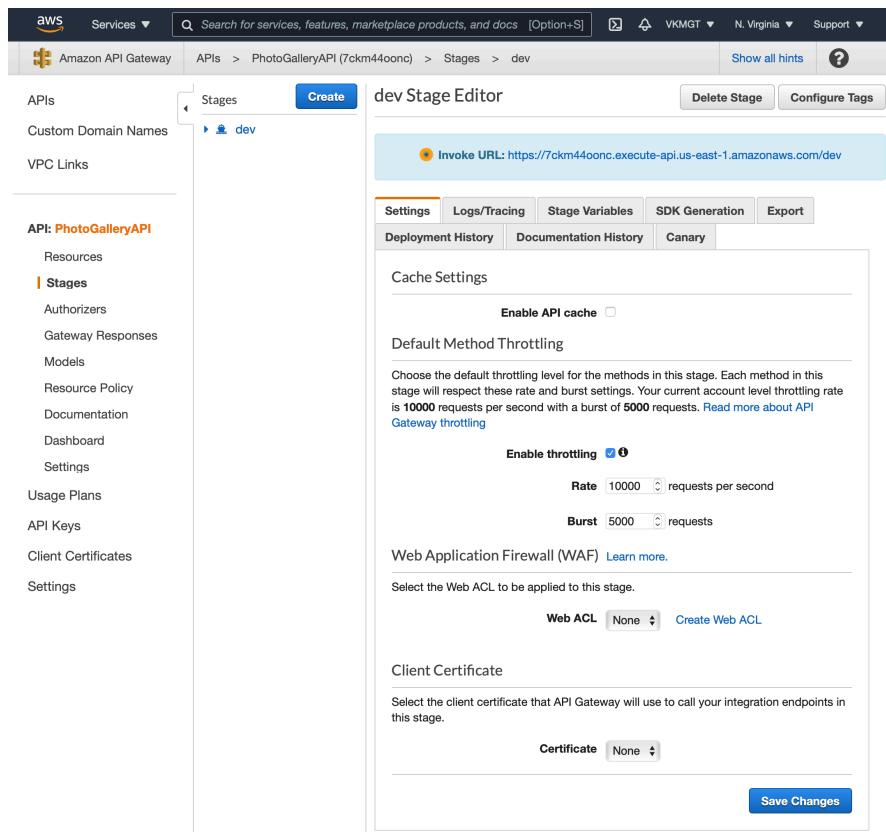
- To deploy the API, click “**Deploy API**”, as shown below

This screenshot shows the AWS Lambda console interface, similar to the previous one but for the root resource. The sidebar and resource tree are identical. A context menu is open over the root resource ('/'). The menu has two sections: RESOURCE ACTIONS (Create Method, Create Resource, Enable CORS, Edit Resource Documentation) and API ACTIONS (Deploy API, Import API, Edit API Documentation, Delete API). The 'Deploy API' option under API ACTIONS is highlighted with a red box. Below the menu, it says 'No methods defined for the resource.'

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

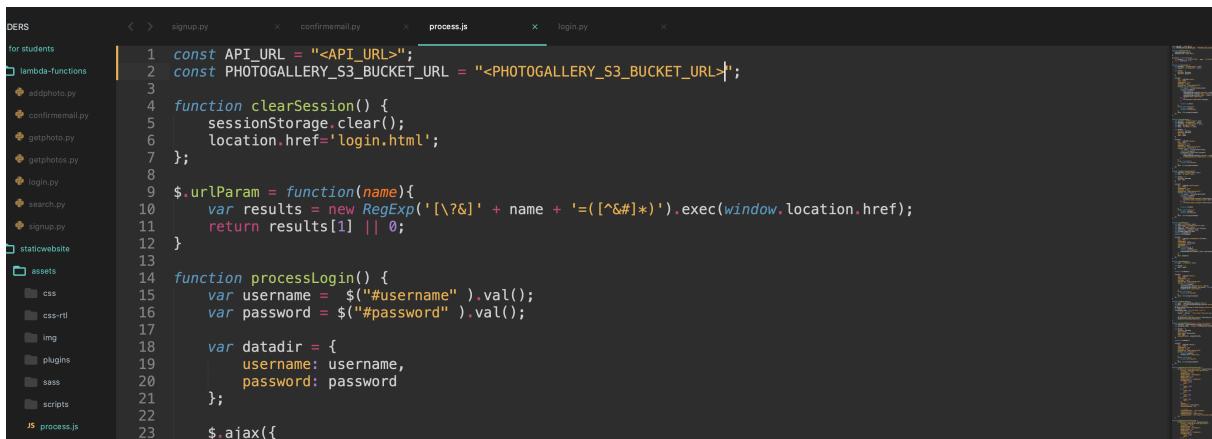


- Copy and save the API URL; will use this URL to access the resources from the website.



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

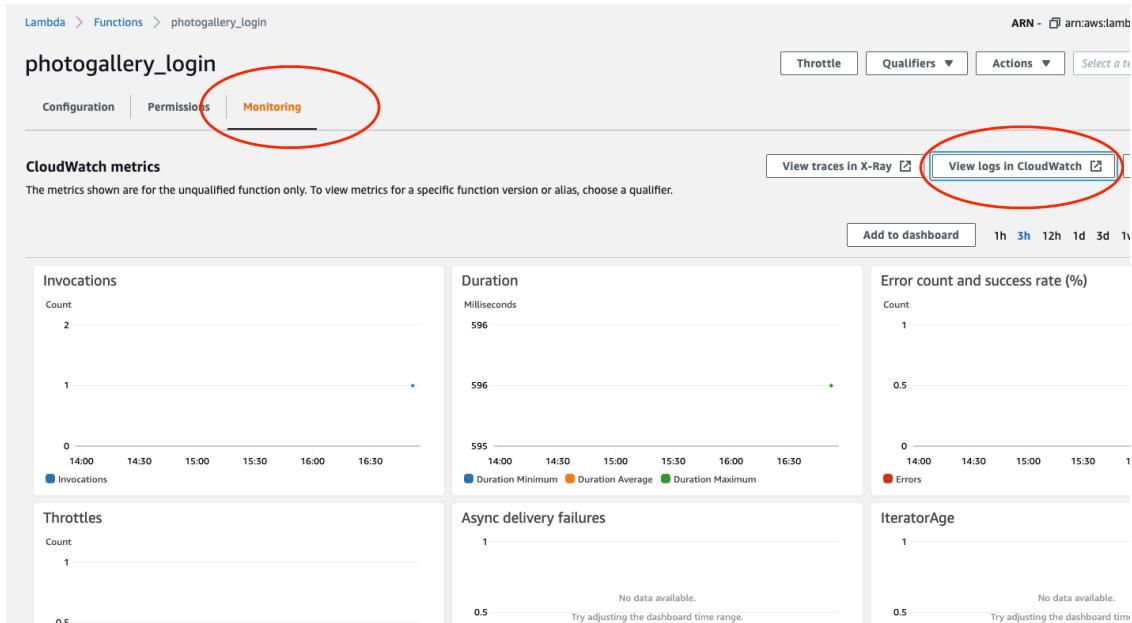
1. Test your code on localhost before uploading to cloud

```
[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

/aws/lambda/photogallery_login											
Actions ▾		View in Logs Insights									
Search log group											
▼ Log group details <table border="1"> <tr> <td>Retention Never expire</td> <td>Creation time 26 minutes ago</td> <td>Stored bytes -</td> <td>ARN <code>arn:aws:logs:us-east-1:610172127508:log-group:/aws/lambda/photogallery_login:*</code></td> </tr> <tr> <td>KMS key ID -</td> <td>Metric filters 0</td> <td>Subscription filters 0</td> <td>Contributor Insights rules -</td> </tr> </table>				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 -
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	X
▶ AmazonDynamoDBFullAccess	AWS managed policy	X
▶ AmazonCognitoPowerUser	AWS managed policy	X

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 two windows side-by-side. On the left is the 'Add ARN(s)' dialog, which has a text input field labeled 'Specify ARN for log-group' with a red arrow pointing to it. Below the input field are fields for 'Region', 'Account', and 'Log group name', each with a 'Any' checkbox. A message at the bottom says 'Entered ARN is invalid.' On the right is a JSON code editor showing a policy document:

```

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. Retry	
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. Auto retry paused. Resume	