SNS / SQS Fanout Pattern Tutorial

Overview

This tutorial implements a Fan Out pattern by creating an SNS Topic with 2 SQS Consumers. Each SQS consumer sends the message to its dedicated lambda function. One Lambda function will generate thumbnail images of any uploaded image files (png, jpg, jpeg) and store them to Thumbnails S3 bucket. The second Lambda function will store the metadata of the image file to a DynamoDB table.

The entire process will start by uploading an image file to the S3 "Upload" bucket that triggers the SNS topic.

Pre-Requisites

This tutorial assumes you have proficiency in S3, S3 triggers, Lambda, DynamoDB, CloudWatch, IAM, and Resource-based policies. It also assumes you are using the Management Console in the AWS Learner Lab account.

Steps

1. Create a an S3 bucket. That will be the destination of the thumbnail images that are generated in the Lambda function. Name it like "hw09-thumbnails".
2. Create a DynamoDB Table. Name it like "hw09-image-metadata".
   - PartitionKey = "FileName"
3. Create two lambda functions. These will be the "consumers" of the queues you will create in subsequent steps.
   - Lambda 1 - This will generate thumbnail image and write to the "Thumbnails" S3 Bucket created above. Name it like "hw09_thumbnail_generator.py".
     - Note: Be sure to select "Python3.9" for this function so that the thumbnail code will work correctly.
   - Lambda 2 - This will write image file metadata (e.g. file name, file size, modified date) to the DynamoDB Table you created above. Name it like "hw09_db_writer.py"
   - Each Lambda should log to CloudWatch by using print statements. Log the important values and information that your functions use and produce. (e.g. the names of the values being written to the database or the image names being resized.)
4. Create a "Standard" SNS Topic
   - For "Deliver status logging," choose "Amazon SQS."
     - You will need to get the ARN of LabRole from AWS IAM and enter it for both the "successful deliveries" and "failed deliveries" values.
5. Edit the newly created Topic
   - In the Access Policy section, edit the "Access Policy" so that the resource and condition sections look like this. Notice the default SourceARN section has been removed.

```
"Resource": "<SNS TOPIC ARN>",
                "Condition": {
                    "ArnLike": {
                      "AWS:SourceArn": "<S3 BUCKET ARN>"
                    }
                }
```

6. Create 2 "Standard" SQS queues, one for each Lambda function you created above
   - Receive Message Wait Time = 20 (which is long-polling)
7. Edit each of your 2 new queues
   - Under Access Policy, click "Advanced" and give your SNS Topic permission to write to this Queue. Use the following template to add an additional SID to your access policy (right after line 13).

```
,{
    "Sid": "_allow_sns",
    "Effect": "Allow",
    "Principal": {
      "Service": "sns.amazonaws.com"
    },
    "Action": "sqs:SendMessage",
    "Resource": "<ARN of your SQS queue>",
    "Condition": {
       "ArnEquals": {
          "aws:SourceArn": "<ARN of your SNS topic>"
       }
    }
}
```

8. In each of your SQS queues, subscribe each queue to your SNS Topic
   - In the "SNS subscriptions" tab, click "Subscribe to Amazon SNS Topic" and select the SNS topic your created above.
9. Create a Dead Letter Queue. You create a dead letter queue the same way you create a regular SQS queue.
   - Create a "Standard" SQS Queue and name it with "dead-letter" in the name so you know which of your queues is the dead letter queue.
   - "Enable" the "Redrive allow policy." Select "By Queue." And select your two SQS queues that were created above as valid sources.
10. Go back and edit your two SQS queues, select the "Dead Letter" tab, and enable the "Dead Letter Queue" option. Then select your dead letter queue you just created. For each one:
11. Create a private S3 "Upload Event" bucket. When objects are uploaded to this bucket, an event notification will be sent to SNS.
12. Select your new Upload bucket and go to "Properties". Scroll down and select "Create Event Notification" in the Event Notifications section.
    - Scroll down to the "Event types" section and check the box for "All object create events". This will trigger whenever an object is added to the bucket.

    - Scroll down to the "Destination" section and select "SNS topic" as the destination. Select your SNS Topic.

13. Create the lambda triggers
    - For both of your two SQS queues, select the "Lambda Triggers" tab and press "Configure Lambda function trigger." Select the appropriate lambda function for each queue.

Testing

We recommend first creating skeleton code for each lambda and making sure you have all the "pipes" connected (e.g. S3 triggering SNS... which fans out to 2 SQS queues... which triggers lambda... which writes to CloudWatch.)

- From your SNS Topic, click the "Publish Message" button. Enter a subject and a body. Then press "Publish Message." If you have completed all the above steps, then you should see a CloudWatch message in each of your Lambda's LogGroups.
    - You could also test from your SQS queues, where you can also "Poll for Messages". This would be useful if you have not yet configured your lambda function to be triggered by you SQS queue.

Now try to upload a file to your S3 "Upload" bucket and see if the messages flow end-to-end by looking in the CloudWatch logs. You will see the format of the message sent to Lambda in the CloudWatch logs. Copy/paste that into an online JSON editor so you can study the format. Now you can work on the Lambda code to process the thumbnail and write the metadata to DynamoDB.

You will notice that the body of the message is escaped. You can use another online tool to convert that to regular JSON so that you can better create your lambda code to pull the information you need from the body.

A GREAT way to develop and test your lambda code that parses the incoming JSON is to copy the "real" message that appeared in your CloudWatch log and use the "Configure Test Event" feature in Lambda. Paste you event into the "Event JSON" text box and "Save". Now you can quickly develop and fine tune the code needed to extract the required elements from the incoming JSON message. We STRONGLY recommend this approach. It will save you a LOT of time.

You will likely find that you need to use the "json.loads(<string>)" python method twice in each Python function. That methods converts a string into a json object. That will allow you to easily pull structured data from you incoming message.

Lambda Code

First, create the lambda code needed to parse the incoming message, as described in the Testing section above.

Next, use the example GitHub yaml file to create a GitHub Actions pipeline to deploy your two lambda functions to AWS. This creates a "deployment package" for that Thumbnail lambda that includes the modules it needs to run.

Use the starter code provided to complete your two functions and deploy them to/in AWS. Once deployed, test them again with the Lambda Test message approach to make sure they fully work before you try and test with the full end-to-end trigger process.

MAKE SURE you are not writing your thumbnail images to the original "Upload" bucket. Doing so would cause an infinite loop and your AWS Learner Lab account will get deactivated.