Lambda

Syllabus of Lambda:

AWS Lambda - Features, Time limit and memory limit for Lambda functions, Event notification from S3, Sample code to put the data in DynamoDB in case of event.

Features of Lambda:

1. **Extend other Amazon Web Services services with custom logic:**
   Amazon Lambda allows you to add custom logic to Amazon Web Services resources such as Amazon S3 buckets and Amazon DynamoDB tables, making it easy to apply compute to data as it is enters or moves through the cloud.
2. **Build custom back-end services:**
   Lambda enables you to create new back-end services for your applications triggered on-demand using the Lambda API or custom API endpoints built with Amazon API Gateway.
   This helps avoid client platform variations, reduce battery drain, and enable easier updates.
3. **Completely automated administration:**
   Lambda manages all the infrastructure required to run your code on highly available, fault-tolerant infrastructure.
   It handles administration, maintenance, security patches, and provides built-in logging and monitoring through Amazon CloudWatch.
4. **Built-in fault tolerance:**
   Lambda maintains compute capacity across multiple Availability Zones to protect your code against individual machine or data center failures.
   It provides predictable and reliable operational performance with no scheduled downtimes.
5. **Automatic scaling:**
   Lambda scales automatically to handle the rate of incoming requests without any configuration required.
   It can handle any number of requests, starting the code execution within milliseconds of an event.

6. **Flexible resource model:**
   You can choose the amount of memory to allocate to your functions, and Lambda allocates proportional CPU power, network bandwidth, and disk I/O accordingly.

7. **Bring Your Own Code:**
   Lambda supports various programming languages, including Java, Node.js, C#, and Python, allowing you to use any third-party library or native code.
8. **Only pay for what you use:**
   With Lambda, you pay for execution duration rather than by server unit. Billing is based on the number of requests served and the compute time required to run your code, with metering in increments of 1 millisecond.

Sample code to put the data in DynamoDB in case of event:

```
import json
import boto3

def lambda_handler(event, context):
    # TODO implement

    bucketname = event['Records'][0]['s3']['bucket']['name']
    objectname = event['Records'][0]['s3']['object']['key']
    eventtime = event['Records'][0]['eventTime']

    dynamodb = boto3.client('dynamodb')
    response = dynamodb.put_item(
    TableName='s3objectinfo',
    Item = {
        'creationtime':{
            'S': eventtime,
        },
        'bucketname':{
            'S': bucketname,
        },
        'objectname':{
            'S': objectname,
        }
    }
    )


    print("The event is happened at: ",eventtime)
    print("The bucket name is: ",bucketname)
    print("Object name is: ",objectname)
    print("The response from dynamodb is : ",response)


    return {
        'statusCode': 200,
        'body': json.dumps('Hello from my first lambda function')
    }
```

[JSON Format AWS Documentation Link]

Certainly! Here's a breakdown of the code section:

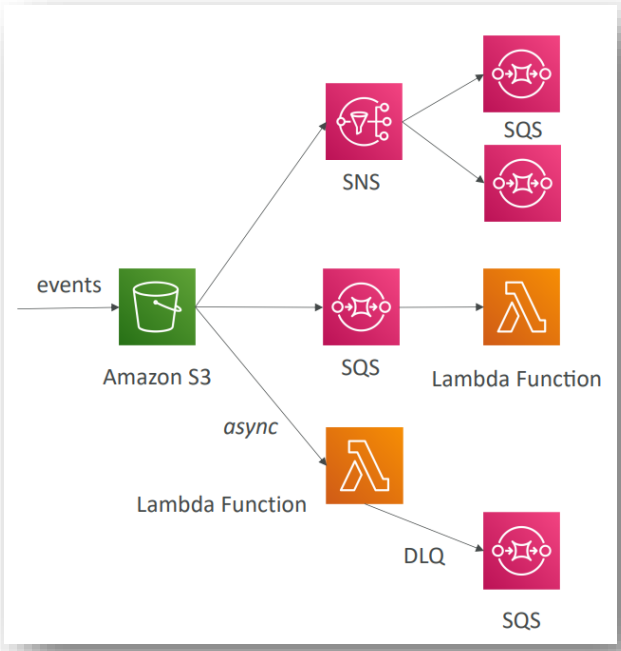
1. **Event Extraction**:
    - **bucketname = event['Records'][0]['s3']['bucket']['name']**: Extracts the name of the S3 bucket from the event data. It accesses the nested dictionary structure of the event to retrieve the bucket name.
    - **objectname = event['Records'][0]['s3']['object']['key']**: Extracts the key (object name) of the S3 object that triggered the event.
2. **Event Time Extraction**:
    - **eventtime = event['Records'][0]['eventTime']**: Extracts the timestamp of the event. It retrieves the value associated with the 'eventTime' key in the event data.
3. **DynamoDB Initialization**:
    - **dynamodb = boto3.client('dynamodb')**: Initializes the DynamoDB client using the **boto3** library. This allows interaction with DynamoDB from within the Lambda function.
4. **DynamoDB Item Insertion**:
    - **response = dynamodb.put_item(TableName='s3objectinfo', Item = {...})**: Inserts a new item into the 's3objectinfo' table in DynamoDB. The item is specified as a dictionary with the attribute names as keys and their corresponding values.
        - **'creationtime': {'S': eventtime}**: Sets the 'creationtime' attribute with the event timestamp as a string value.
        - **'bucketname': {'S': bucketname}**: Sets the 'bucketname' attribute with the S3 bucket name as a string value.
        - **'objectname': {'S': objectname}**: Sets the 'objectname' attribute with the S3 object key as a string value.
5. **Print Statements**:
    - **print("The event is happened at: ", eventtime)**: Prints the event timestamp for debugging purposes.

- **print("The bucket name is: ", bucketname)**: Prints the S3 bucket name for debugging purposes.
- **print("Object name is: ", objectname)**: Prints the S3 object key for debugging purposes.
- **print("The response from dynamodb is : ", response)**: Prints the response from the DynamoDB put_item operation for debugging purposes.

6. **Response**:
    - **return {'statusCode': 200, 'body': json.dumps('Hello from my first lambda function')}**: Returns a response with a status code of 200 and a JSON-encoded body message. This is a simple response indicating the successful execution of the Lambda function.

This code demonstrates how to extract relevant information from an S3 event, insert it into a DynamoDB table, and provide a response. It can be customized or expanded based on specific use cases or requirements.

Event notification from S3:

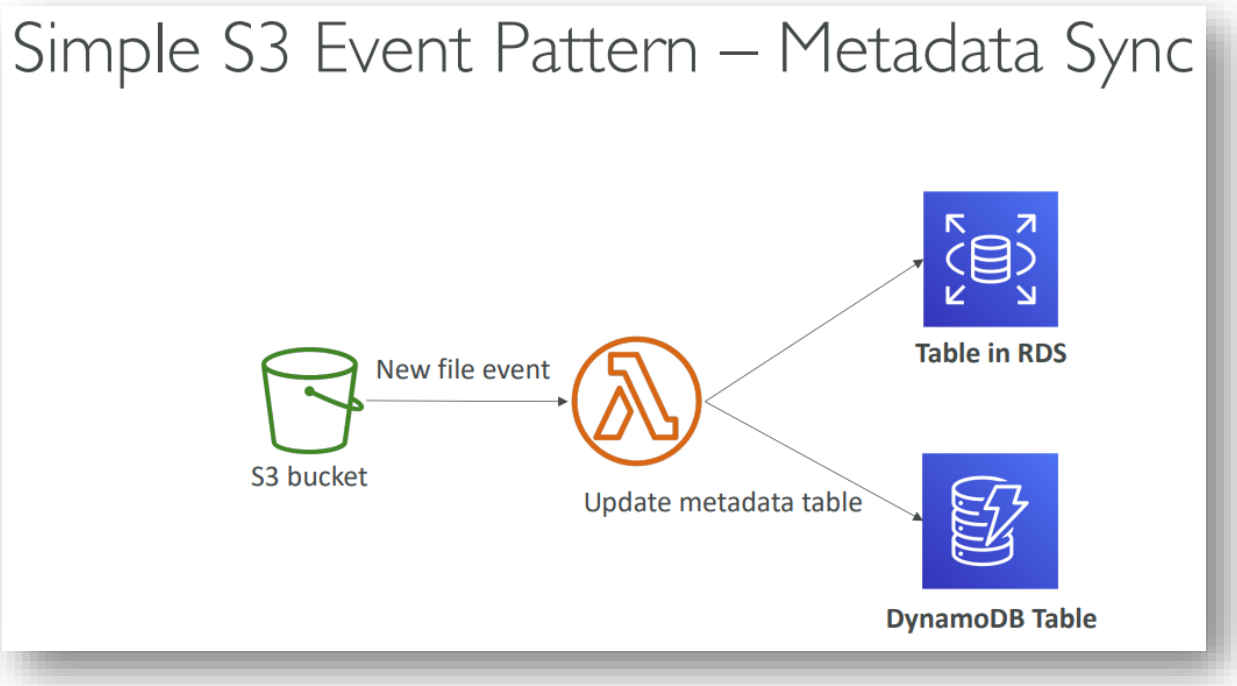


To set up event notification, you need to follow these steps:

1. **Create an S3 bucket:** Start by creating an S3 bucket or selecting an existing one where your objects will be stored.
2. **Configure event notification:** Access the bucket properties in the S3 Management Console and navigate to the "Events" section. Create a new event notification and specify the event(s) that should trigger the notification, such as object creation, deletion, or modification. You can also set filters based on object key prefixes or suffixes to further refine the triggering conditions.
3. **Select Lambda Function as the destination:** When configuring the event notification, choose "Lambda Function" as the destination for the event. This indicates that your Lambda function will be invoked when the specified event occurs.
4. **Configure the Lambda function:** In the Lambda Management Console, create a new Lambda function or select an existing one that will be triggered by the S3 event notification. Make sure to set the appropriate runtime and configure the function with the necessary permissions to access the S3 bucket.

5. **Implement the Lambda function:** Write the code for your Lambda function to process the S3 event. When triggered, the event data will be passed to the Lambda function as an input parameter. Extract the relevant information from the event, such as the bucket name, object key, or event type. Implement the desired logic within the function to handle the event, such as processing the uploaded file, performing data transformations, or triggering further actions.
6. **Test and monitor:** Save and deploy your Lambda function. Upload or modify objects in the configured S3 bucket to trigger the Lambda function based on the defined event. Monitor the execution logs and CloudWatch metrics to verify that the Lambda function is triggered correctly and performing as expected.

By leveraging event notification from S3, you can automate various processes, such as generating thumbnails, processing uploaded files, updating databases, or triggering downstream workflows based on changes in your S3 bucket. This integration provides a powerful mechanism for event-driven architectures and enables seamless integration between S3 and Lambda services.



Time limit and memory limit for Lambda functions:

AWS Lambda provides time and memory limits for the execution of Lambda functions. These limits ensure that functions run within defined constraints to promote efficient resource utilization and overall system stability. Here are the details:

1. **Time Limit:**
    - The default execution duration for a single invocation of a Lambda function is *3 seconds* and the maximum execution is *900 seconds* (15 minutes).
    - If a function exceeds this time limit, AWS Lambda terminates its execution and returns an error to the invoking service or client.
    - It's important to design your functions to complete within this time limit and handle any long-running tasks asynchronously or through other mechanisms.
2. **Memory Limit:**
    - AWS Lambda allows you to configure the amount of memory allocated to a function, ranging from 128 MB to a maximum of 10,240 MB (10 GB).
    - The memory allocation also determines the proportional CPU power, network bandwidth, and disk I/O available to the function.
    - Increasing the memory allocation generally results in higher performance and faster execution, as it allows for more available resources.
    - However, keep in mind that higher memory allocations also impact the cost of running the function, as you are billed based on the memory consumption and the duration of the function's execution.

It's important to optimize your Lambda functions by considering both the time and memory limits. This involves efficiently using resources, choosing appropriate memory allocations, and designing functions to complete their tasks within the specified time limit. Monitoring and performance testing can help identify any performance bottlenecks and optimize the resource usage of your functions.