AWS LAMBDA

AWS
Lambda

Presented By

SHUBHAM CHOUHAN

OOGLELABS

What is AWS Lambda?

AWS Lambda is a serverless, event-driven computing platform offered by Amazon as part of Amazon Web Services. It is a computing service that runs code in response to events and manages the computing resources required by that code automatically.

AWS Lambda was created for use cases such as uploading images or objects to Amazon S3, updating DynamoDB tables, responding to website clicks, or responding to sensor readings from an IoT-connected device. AWS Lambda can also be used to automatically provision back-end services triggered by custom HTTP requests, as well as "spin down" such services when they are not in use to save resources. These custom HTTP requests are set up in AWS API Gateway, which can also handle authentication and authorization when used in conjunction with AWS Cognito.

Key Features of AWS Lambda

Integrated Fault Tolerance: To help protect your code against individual machine or data center facility failures, AWS Lambda maintains compute capacity across multiple Availability Zones (AZs) in each AWS Region.

Scaling on the fly: AWS Lambda executes your code only when it is required and scales automatically to meet the volume of incoming requests without any manual configuration. Your code has no limit on the number of requests it can handle.

Access Shared File Systems: You can securely read, write, and persist large volumes of data at any scale using the Amazon Elastic File System (EFS) for AWS Lambda. To process data, you do not need to write code or download it to temporary storage. This saves you time, allowing you to concentrate on your business logic.

Custom Logic can be used to extend other AWS services: AWS Lambda enables you to add custom logic to AWS resources like Amazon S3 buckets and Amazon DynamoDB tables, allowing you to easily compute data as it enters or moves through the cloud.

Create your own Backend Services: AWS Lambda can be used to build new backend application services that are triggered on-demand via the Lambda application programming interface (API) or custom API endpoints built with Amazon API Gateway.

What is Boto3?

Boto is a Software Development Kit (SDK) that aims to improve the use of Python programming in Amazon Web Services. The Boto project began as a user-contributed library to assist developers in building Python-based cloud applications by converting AWS application programming interface (API) responses into Python classes.

Boto has been designated as the official AWS Python SDK. Boto comes in three flavors:

1. Boto
2. Boto3
3. Botocore

Boto3 is the most recent version of the SDK, and it supports Python versions 2.6.5, 2.7, and 3.3. Boto3 includes a number of service-specific features to make development easier. Boto is compatible with all current AWS cloud services, such as Elastic Compute Cloud, DynamoDB, AWS Config, CloudWatch, and Simple Storage Service.

Boto3 replaced Boto version 2, which is incompatible with the most recent versions of Python but remains appealing to software developers who use older versions of the programming language. Botocore provides more basic access to AWS tools, allowing users to make low-level client requests and receive results from APIs.

Key Features of Boto3

APIs for Resources: Boto3's APIs are divided into two levels. Client (or "low-level") APIs map to the underlying HTTP API operations one-to-one.

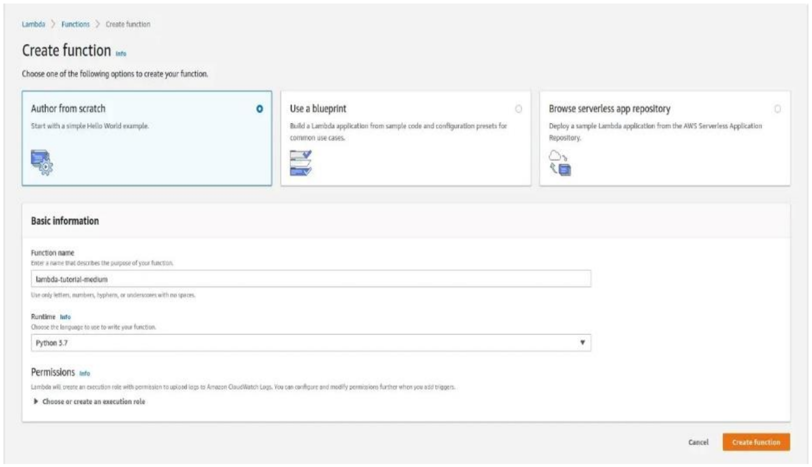
Consistent and up-to-date Interface: Boto3's 'client' and 'resource' interfaces are driven by JSON models that describe AWS APIs and have dynamically generated classes. This enables us to provide extremely fast updates while maintaining high consistency across all supported services.

Waiters: Boto3 includes 'waiters,' which poll for pre-defined status changes in AWS resources. You can, for example, start an Amazon EC2 instance and use a waiter to wait until it reaches the 'running' state, or you can create a new Amazon DynamoDB table and wait until it is ready to use.

High-level Service-Specific Features: Boto3 includes many service-specific features, such as automatic multi-part transfers for Amazon S3 and simplified query conditions for Amazon DynamoDB.

Amazon Lambda Basic Structure

One of the main troubles I encountered when trying to implement my first Lambda functions was trying to understand the file structure used by AWS to invoke scripts and load libraries. If you follow the default options 'Author from scratch' (Figure 1) for creating a Lambda function, you'll end up with a folder with the name of your function and Python script named lambda_function.py inside it.



Choose one of the following options to create your 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.

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

Basic information

Function name: lambda-tutorial-medium
Enter a name that describes the purpose of your function.
Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime: Python 3.7
Choose the language to use to write your function.

Permissions
Lambda will create an execution role with permission to upload logs to Amazon CloudWatch Logs. You can configure and modify permissions further when you add triggers.
▸ Choose or create an execution role

The **lambda_function.py** file has a very simple structure and the code is the following:

```python
import json
def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

These 8 lines of code are key to understanding Amazon Lambda, so we are going through each line to explain it.

```python
import json
```
You can import Python modules to use on your function and AWS provides you with a list of available Python libraries already built on Amazon Lambda, like json and many more. The problem starts when you need libraries that are not available (we will solve this problem later using Lambda Layers).

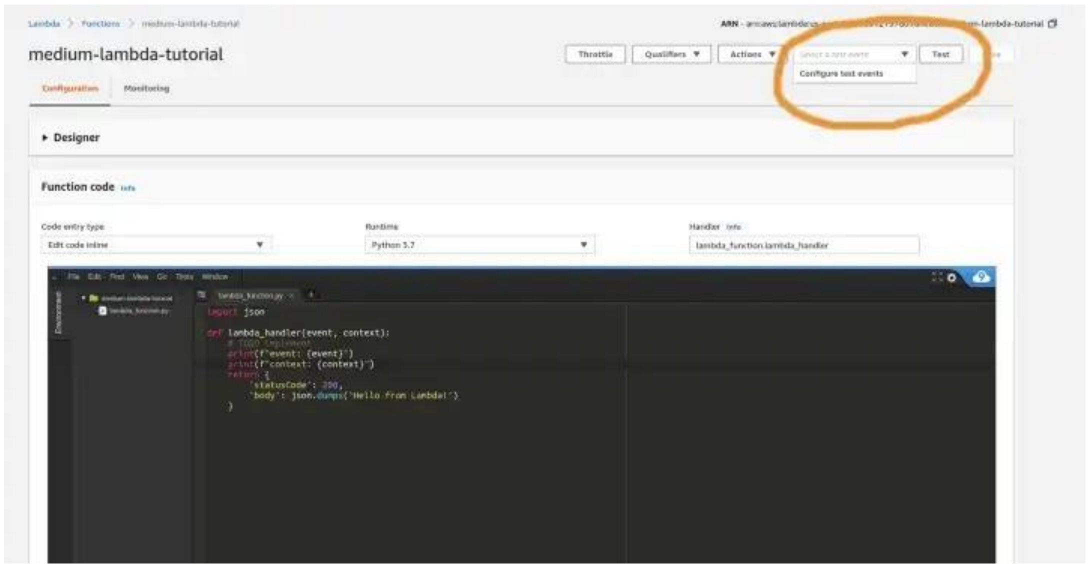
```python
def lambda_handler(event, context):
```
This is the main function your Amazon Lambda is going to call when you run the service. It has two parameters event and context. The first one is used to pass data that can be used on the function itself (more on this later), and the second is used to provide runtime and metadata information.

```python
  # TODO implement
```
Here is where the magic happens! You can use the body of the lambda_handler function to implement any Python code you want.

```python
   return
```
This part of the function is going to return a default dictionary with statusCode equal to 200, and body with a “Hello from Lambda”. You can change this return later to any Python object that suits your needs.

Before running our first test, it’s important to explain a key topic related to Amazon Lambda: Triggers. Triggers are basically ways in which you invoke your Lambda function. There are many ways for you to set up your trigger using events like adding a file to a S3 bucket, changing a value on a DynamoDB table or using an HTTP request through Amazon API Gateway. You can pretty much integrate your Lambda function to be invoked by a wide range of AWS services and this is probably one of the advantages offered by Lambda. One way we can do it to integrate with your Python code is by using boto3 to call your Lambda function, and that’s the approach we are going to use later on this tutorial.

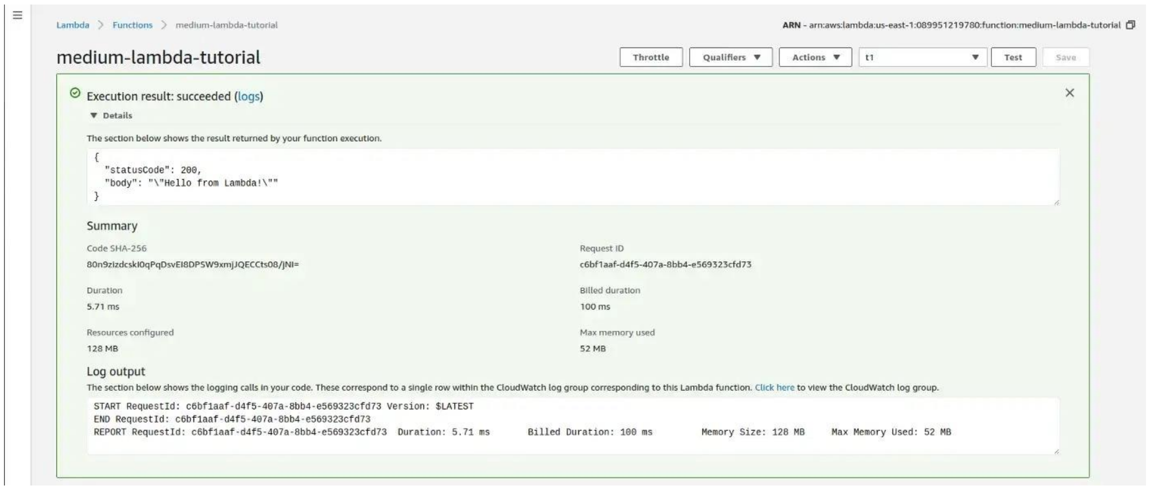
As you can see, the template structure offered by AWS is super simple, and you can test it by configuring a test event and running it (Figure 2).



```
import json

def lambda_handler(event, context):
    # TODO implement
    print(f"event: {event}")
    print(f"context: {context}")
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

As we didn't change anything on the code of the Lambda Function, the test runs the process and we receive a green alert describing the successful event (Figure 3).


Lambda > Functions > medium-lambda-tutorial
ARN - arn:aws:lambda:us-east-1:089951219780:function:medium-lambda-tutorial
medium-lambda-tutorial
Throttle
Qualifiers
Actions
t1
Test
Save
Details

Execution result: succeeded (logs)

The section below shows the result returned by your function execution.

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

Summary

Code SHA-256: 80n9zizdcskI0qPqDsvEI8DPSW9xmjJQECCts08/jNI=

Request ID: c6bf1aaf-d4f5-407a-8bb4-e569323cfd73

Duration: 5.71 ms

Billed duration: 100 ms

Resources configured: 128 MB

Max memory used: 52 MB

Log output

The section below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.

```
START RequestId: c6bf1aaf-d4f5-407a-8bb4-e569323cfd73 Version: $LATEST
END RequestId: c6bf1aaf-d4f5-407a-8bb4-e569323cfd73
REPORT RequestId: c6bf1aaf-d4f5-407a-8bb4-e569323cfd73  Duration: 5.71 ms       Billed Duration: 100 ms         Memory Size: 128 MB     Max Memory Used: 52 MB
```

Figure 3 illustrates the layout of the Lambda invocation result. On the upper part you can see the dictionary contained on the returned statement. Underneath there is the **Summary** part, where we can see some important metrics related to the Lambda Function like Request ID, duration of the function, the billing duration and the amount of memory configured and used. We won't go deep on Amazon Lambda pricing, but it is important to know that is charged based on:

- duration the function is running (rounded up to the nearest 100ms)
- the amount of memory/CPU used
- the number of requests (how many times you invoke your function)
- amount of data transferred in and out of Lambda

In general, it is really cheap to test and use it, so you probably won't have billing problems when using Amazon Lambda for small workloads.

Another important detail related to the pricing and performance is how CPU and memory are available. You choose the amount of memory for running your function and “Lambda allocates CPU power linearly in proportion to the amount of memory configured”.

At the bottom of Figure 3, you can see the **Log output** session where you can check all the execution lines printed by your Lambda function. One great feature implemented on Amazon Lambda is that it is integrated with Amazon CloudWatch, where you can find all the logs generated by your Lambda functions. For more details on monitoring execution and logs, please refer to Casey Dunham great Lambda Article.

We have covered the basic features of Amazon Lambda, so on the next sessions, we are going to increase the complexity of our task to show you a real-world use providing a few insights into how to run a serverless service on a daily basis.

Creating Instance using Lambda Function

```
import os
import boto3

#Environment variables

AMI = os.environ['AMI']
INSTANCE_TYPE = os.environ['INSTSNCE_TYPE']
KEY_NAME = os.environ['KEY_NAME']
SUBNET_ID = os.environ['SUBNET_ID']
REGION = os.environ['REGION']

ec2 = boto3.client('ec2' , region_name=REGION)

#Execution Command in Terminal

def lambda_handler(event, context):
    init_script = """#!/bin/bash
                yum update -y"""

    instance = ec2.run_instances(
        ImageId=AMI,
        InstanceType=INSTANCE_TYPE,
        KeyName=KEY_NAME,
        SubnetId=SUBNET_ID,
        MaxCount=1,
        MinCount=1,
        InstanceInititiatedShutdownBehaviour='terminate',
        UserData=init_script
    )
```

Creating VPC using Lambda Function

```python
import boto3
ec2 = boto3.resource('ec2', region_name='us-east-2')

# create VPC
vpc = ec2.create_vpc(CidrBlock='192.168.0.0/16')
# we can assign a name to vpc, or any resource, by using tag
vpc.create_tags(Tags=[{"Key": "Name", "Value": "default_vpc"}])
vpc.wait_until_available()
print(vpc.id)

# create then attach internet gateway
ig = ec2.create_internet_gateway()
vpc.attach_internet_gateway(InternetGatewayId=ig.id)
print(ig.id)

# create a route table and a public route
route_table = vpc.create_route_table()
route = route_table.create_route(
    DestinationCidrBlock='0.0.0.0/0',
    GatewayId=ig.id
)
print(route_table.id)

# create subnet
subnet = ec2.create_subnet(CidrBlock='192.168.1.0/24', VpcId=vpc.id)
print(subnet.id)

# associate the route table with the subnet
route_table.associate_with_subnet(SubnetId=subnet.id)

# Create sec group
sec_group = ec2.create_security_group(
    GroupName='slice_0', Description='slice_0 sec group', VpcId=vpc.id)
sec_group.authorize_ingress(
    CidrIp='0.0.0.0/0',
    IpProtocol='icmp',
    FromPort=-1,
    ToPort=-1
)
print(sec_group.id)

def lambda_handler(event, context):
    init_script = """#!/bin/bash"""
```

Stop Instance using Lambda Function

```python
import boto3
region = 'us-east-1'
instances = ['i-001e2d04e37ccde3b']
ec2 = boto3.client('ec2', region_name=region)

def lambda_handler(event, context):
    print('Stopping instances')
    ec2.stop_instances(InstanceIds=instances)
```

Start Instance using Lambda Function

```python
import boto3
region = 'us-east-1'
instances = ['i-001e2d04e37ccde3b']
ec2 = boto3.client('ec2', region_name=region)

def lambda_handler(event, context):
    print('Starting instances')
    ec2.start_instances(InstanceIds=instances)
```