Chandrahas Ravi Kiran

1 Follower About Follow

Everything with AWS Lambda

C Chandrahas Ravi Kiran Oct 14, 2020 · 10 min read

Running any application, be it a small mobile application or a massive application on which the entire organization is dependent on, starts with a requirement for computing resources. AWS offers different computing resources that allow you to develop, deploy, run, and scale your applications. One of the most commonly used compute services is AWS Lambda.

What is AWS Lambda?

Lambda is an event-driven, serverless computing service that can be implemented in many popular languages like Python, Java, Node.js, Go, etc. But you might be thinking, what do the terms 'event-driven' and 'serverless' mean? Lambda is defined as event-driven because you can associate the lambda with a trigger (find more on triggers in the following sections) invoking the lambda with the 'event' information. Lambda uses a 'serverless' design where you don't have to worry about managing or scaling servers. It's based on pay per use model where you can trigger and execute your function on demand and just pay for what you have used.

When to use lambda?

This would probably be the first question that runs through your mind when you think of AWS lambda. In simple terms, any process which is bound to the constraints of lambda be it the processing time taken or be it the memory the process might consume, or be it the number of concurrent executions can be executed using lambda. Lambda does have constraints defined on these parameters. For instance, during the time at which this article was published, the maximum timeout for lambda is of 900 sec and the maximum memory allowed to use is 3008MB. As these values might vary, you can refer to the official website to get the latest values. So, using lambda is not conducive if the given process might exceed any one of the offered capabilities. Lambda can be used for data migration or it can be used as a consumer of a stream or it can be used for building lightweight APIs.

How does lambda work?

Whenever a lambda function gets invoked to serve a request, a container is created to cater to the request. Now, what is a container? A container is an *isolated* entity packaging application code, configurations and dependencies into a single object. What does it mean by saying an isolated entity? Well, it means that resources allocated to the container cannot be used or accessed by a process that runs outside the container. You can think of a container as a mini virtual machine. When a container is created for executing the lambda function, all the dependencies required, i.e, the programming language, the execution code are made readily available in the container. Before the function starts, each function's container is allocated with CPU and RAM capacity. The customer is then billed based on the memory reserved by the lambda function (but not the fraction of memory used during execution) and the total execution time.

The entire infrastructure layer of Lambda is managed by AWS. The end consumers using lambda don't get much visibility on how these systems operate behind the scenes. However, the consumers don't have to worry about updating the underlying infrastructure such as the software, security patches, etc which AWS takes care of automatically.

Each container can serve one request at a time. But, what if you get multiple requests to the lambda at the same time? Multiple containers are created for the same lambda function. All the created containers will be used to serve all the requests concurrently. Despite the increase in the number of concurrent executions, you will *only* be charged based on the execution time of all the containers combined. This makes lambda a good fit for deploying highly scalable cloud applications in terms of cost and maintenance. During the time at which this article was published, the maximum number of concurrent requests supported by lambda *by default* is 1000.

Common practices of lambda.

As stated earlier the lambda can be used in several ways in the real world scenarios and the most common use cases include

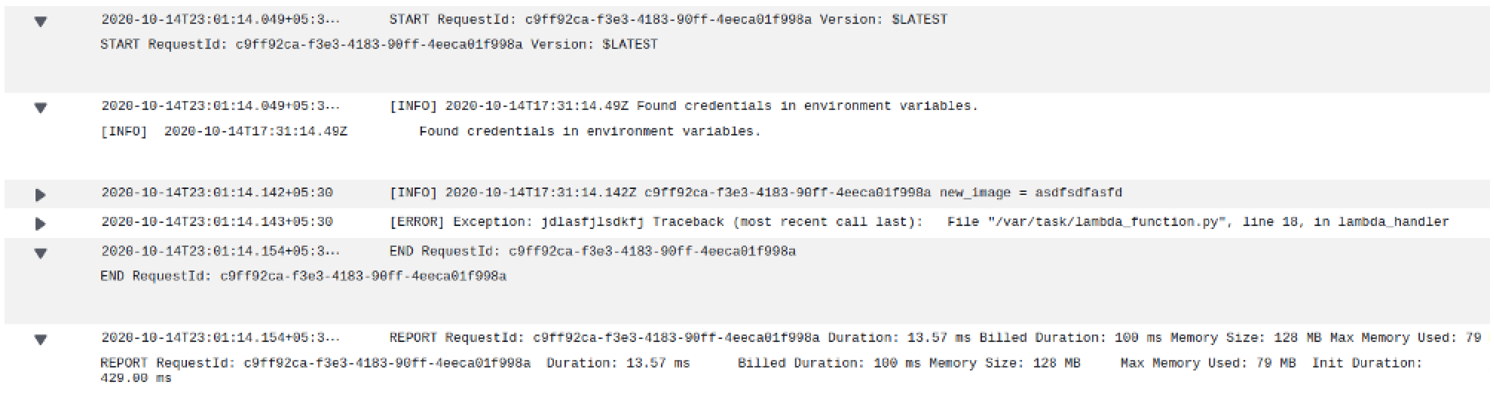
- Lambda as an API
- Lambda invoking Lambda
- Lambda as a Trigger
- Lambda as a cron job

API using lambda

To create an API using Lambda, you would integrate lambda with a service called API Gateway. In simple terms, API Gateway forwards all the data from the client (entity making requests using the API) to the AWS lambda. The lambda will process the data received from API Gateway and returns the result to API Gateway which in turn returns the result to the client.

A usual API flow with AWS Lambda

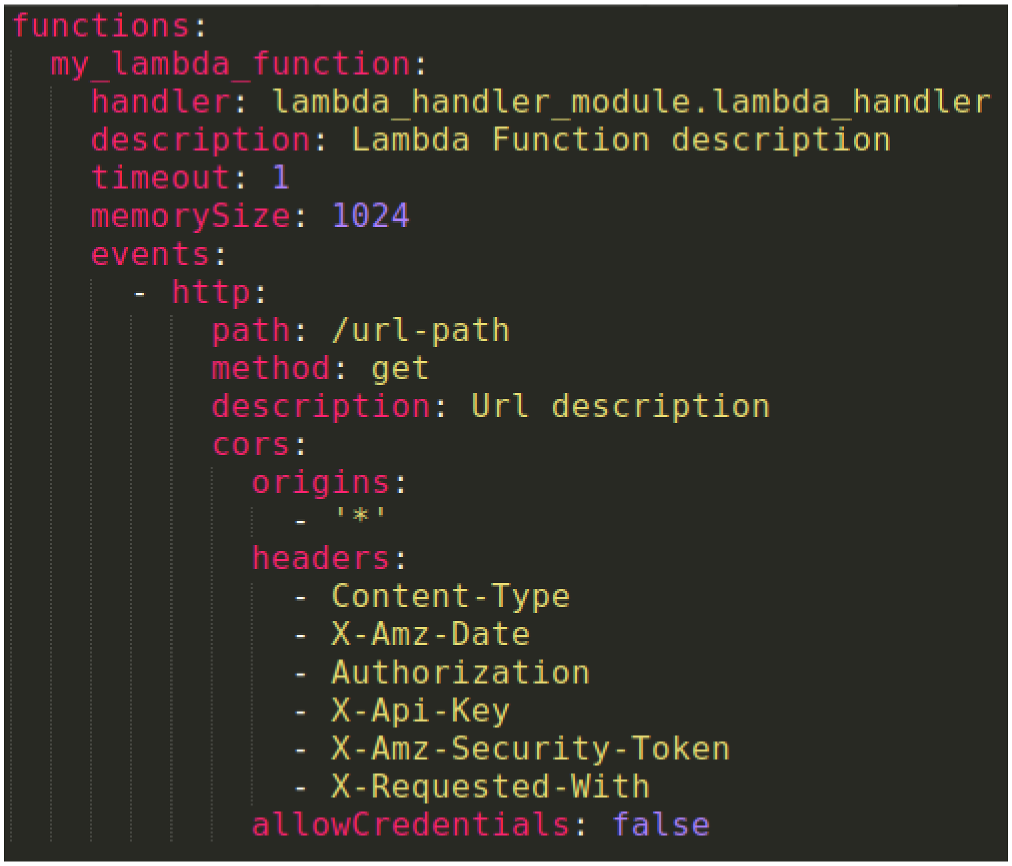
For every invocation of lambda, a unique UUID formatted request-id is generated and the same can be seen in the AWS Cloudwatch logs for the invoked lambda.



```
2020-10-14T23:01:14.049+05:3...    START RequestId: c9ff92ca-f3e3-4183-90ff-4eeca01f998a Version: $LATEST
START RequestId: c9ff92ca-f3e3-4183-90ff-4eeca01f998a Version: $LATEST

2020-10-14T23:01:14.049+05:3...    [INFO] 2020-10-14T17:31:14.49Z Found credentials in environment variables.
[INFO]  2020-10-14T17:31:14.49Z       Found credentials in environment variables.

2020-10-14T23:01:14.142+05:30      [INFO] 2020-10-14T17:31:14.142Z c9ff92ca-f3e3-4183-90ff-4eeca01f998a new_image = asdfsdfasfd
2020-10-14T23:01:14.143+05:30      [ERROR] Exception: jdlasfjlsdkfj Traceback (most recent call last):   File "/var/task/lambda_function.py", line 18, in lambda_handler
2020-10-14T23:01:14.154+05:3...    END RequestId: c9ff92ca-f3e3-4183-90ff-4eeca01f998a
END RequestId: c9ff92ca-f3e3-4183-90ff-4eeca01f998a

2020-10-14T23:01:14.154+05:3...    REPORT RequestId: c9ff92ca-f3e3-4183-90ff-4eeca01f998a Duration: 13.57 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 79
REPORT RequestId: c9ff92ca-f3e3-4183-90ff-4eeca01f998a  Duration: 13.57 ms     Billed Duration: 100 ms Memory Size: 128 MB    Max Memory Used: 79 MB  Init Duration:
429.00 ms
```

AWS Lambda logs from AWS Cloudwatch.

You can implement all the HTTP request methods like GET, PUT, POST, DELETE, etc. using API Gateway and lambda integration.



```yaml
functions:
  my_lambda_function:
    handler: lambda_handler_module.lambda_handler
    description: Lambda Function description
    timeout: 1
    memorySize: 1024
    events:
      - http:
          path: /url-path
          method: get
          description: Url description
          cors:
            origins:
              - '*'
            headers:
              - Content-Type
              - X-Amz-Date
              - Authorization
              - X-Api-Key
              - X-Amz-Security-Token
              - X-Requested-With
            allowCredentials: false
```

serverless.yml snippet for the lambda when used as API

Lambda invoking another Lambda

There might be a scenario where you don't want to do the processing in the existing lambda because either it was a demanding situation due to limited capabilities or you intentionally wanted to have the subsequent logic in a separate lambda. Let us look at two scenarios that best depicts this use case.

Scenario 1 (Synchronous Invocation):

Consider a scenario where you are trying to build an API and part of it is already being handled by another lambda. So, instead of redefining the same functions and logic in the existing lambda, you can directly invoke the lambda which does the same work by providing the required payload and get the output from the lambda and then work upon it.

Scenario 2 (Asynchronous Invocation):

Consider a scenario where you have an API whose functionality is to mail the requested data to the client. Sometimes the requested data can be pretty large enough that it cannot be processed under the API gateway timeout limit which is 30sec and the API Gateway might give a 504. Remember that the lambda invoked by API

Gateway continues to execute in the background because it hasn't reached its timeout limit and might process the requested data and then return to the API Gateway which had already raised 504 to the client. Stated in this way, this leads to some sort of ambiguity at the client side as the request was processed and at the same time 504 was raised by API. This scenario can be handled by having two lambda one which is exposed to API gateway and the other which does the actual work. The lambda which is attached to the API Gateway accepts the request and calls the data processing lambda asynchronously and immediately gives a success message to the client saying that your request is being processed and will be mailed soon.

The lambda invocation is achieved by *invoke()* method and the type of invocation is defined by a variable called *InvocationType* and the payload is provided to the lambda by using the payload variable of *invoke()* method.

The function parameter invocation_type takes three values

1. RequestResponse (Synchronous Invocation)
2. Event (Asynchronous Invocation)
3. DryRun (Validating the input payload of the main lambda)

Invoking AWS Lambda asynchronously from other Lambda

Trigger invoking lambda

This is probably one of the most important use cases of lambda, where a lambda can be used as a consumer of a stream or to perform an action by having a trigger defined on it. For Instance, let's say you have an Amazon Dynamodb table and the same data is also being maintained in Amazon Elasticsearch Service and whenever there is an update of an existing item or creation of a new item, in order to reflect the updated data on to Amazon Elasticsearch Service we can define an Amazon Dynamodb stream and make lambda listen to the stream. Whenever there is an update or creation of an item in Amazon Dynamodb it pushes the new data into the Amazon Dynamodb stream and the same is consumed by lambda from where the data is again pushed to Amazon Elasticsearch Service.

Now let's look at a different scenario that delineates how a lambda can be used to process the triggers. Let's say you have a file in s3 and whenever there is an update on the file you might want to alert the users by sending a copy of data to the users through the mail. This can be achieved by defining an s3 trigger on the lambda and whenever a specific file got modified a trigger indicating the same is made to lambda. Inside the lambda, you can have a logic to read the modified file through the key variable which is obtained in the trigger event and send the mail to the desired users.

Some of the most common triggers include but are not limited to the following:

- API Gateway event
- S3 event (Simple Storage Service)
- DynamoDB event
- SQS (Simple Queuing Service)
- AWS Kinesis

serverless.yml snippet of s3 trigger defined on AWS Lambda

Lambda as a cron

We can configure a lambda as a scheduled job which runs in particular intervals or once in a day depending on the use case. This can be achieved by using a scheduled event while configuring lambda in the serverless.yml file. If the lambda has to be configured as a job which runs every 2 hours we use rate syntax or if it has to be a job which runs daily once we use cron syntax. The same has been depicted in the code snippet below.

serverless.yml snippet for AWS Lambda as a cron

Creating AWS Lambda using the blueprint

There are three ways of creating lambda:

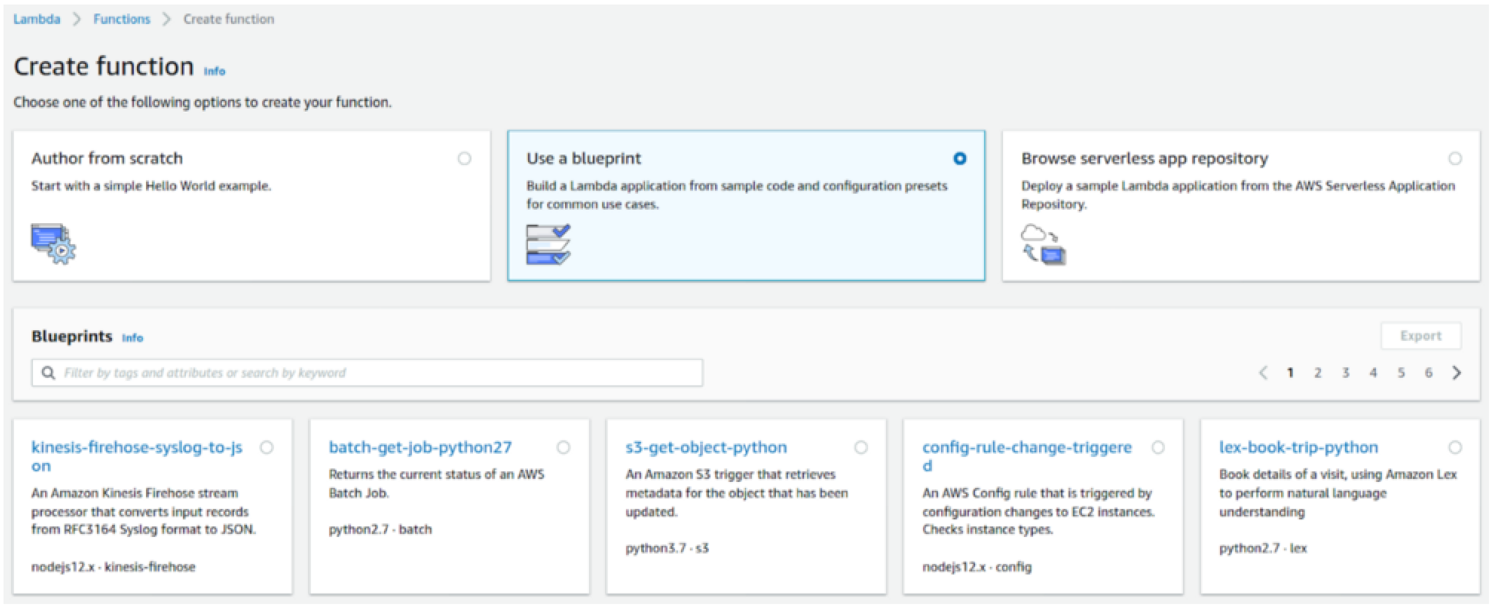
1. Authoring the lambda from scratch
2. Deploying a lambda using a blueprint.
3. Using the serverless app repository by AWS.

We will be discussing the second method of creating lambdas here, which is using blueprints. Blueprints are some common templates that are readily available to create Lambda functions. They are simple to get-started templates that are preconfigured with a trigger type and code.

Eg: You can find the following blueprints which can be used to create lambda functions:

- s3-get-object-python: This will have a preconfigured s3 trigger and the trigger related details can be given while creating the lambda. The other thing to notice is that this blueprint is configured for Python 3.7. You will get a similar blueprint for Node.js as AWS Lambda runtime language.

- dynamo-process-stream: This blueprint can be used to create a lambda that will log all the updates that are made to a Dynamodb table.


Lambda > Functions > Create function

Create function Info

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.

Blueprints Info

Export

Filter by tags and attributes or search by keyword

< 1 2 3 4 5 6 >

kinesis-firehose-syslog-to-json
An Amazon Kinesis Firehose stream processor that converts input records from RFC3164 Syslog format to JSON.
nodejs12.x · kinesis-firehose

batch-get-job-python27
Returns the current status of an AWS Batch Job.
python2.7 · batch

s3-get-object-python
An Amazon S3 trigger that retrieves metadata for the object that has been updated.
python3.7 · s3

config-rule-change-triggered
An AWS Config rule that is triggered by configuration changes to EC2 instances. Checks instance types.
nodejs12.x · config

lex-book-trip-python
Book details of a visit, using Amazon Lex to perform natural language understanding
python2.7 · lex

AWS Lambda function creation using blueprint

Lambda Limitations

Till now we have been looking at different scenarios discussing how a lambda can be used to get your work done but do remember that as stated earlier, there are limitations defined in terms of memory, time, concurrency executions, deployment package size and etc. All these restraints have to be well analyzed before you make up your choice of using lambda.

https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

Miscellaneous

1. **AWS Lambda layers:**
   As mentioned in the *Lambda limitations* section above, the maximum size of the deployment package of a lambda can be 50 MB. But there could be use cases where your lambda deployment package can easily exceed the 50 MB limit. For example, packages like numpy and pandas can take up large portions of the deployment package leaving little space for the actual growing application code. This is where AWS lambda layers will be of help. AWS lambda layers is a ZIP archive used to store all your 3rd party libraries, custom libraries, or other dependencies that will be used by one or many lambda functions. The content in

layers will be pulled in as additional code and content while executing the lambda. AWS Lambda layers will help keep your deployment package size low, making deployment easier. You can use a maximum of 5 layers for one lambda function at a time. One limitation to keep in mind while adding layers is the total size of the layers. The total unzipped size of the lambda function and all the layers associated with it cannot exceed the size limit of 250MB.

2. **Dead letter queue (DLQ):**
In an asynchronous invocation of lambda whenever the lambda fails to process a particular request on the first attempt it retries the same for another two times and the lambda gets blocked even if it cannot process the request for the third attempt hence, blocking all other requests in the streams. In order to overcome such a situation, you can configure a 'DeadLetterConfig' property when creating or updating your Lambda function. This can be achieved by providing SQS(Simple Queue Service) or SNS(Simple Notification Service) as TargetARN for DLQ. Lambda, when configured with DLQ, will push all the requests to TargetARN specified when it is unsuccessful in processing the request after three attempts.

AWS Lambda AWS Lambda Serverless Dead Letter Queue