AMAZON AWS SOLUTIONS ARCHITECT

AWS LAMBDA OVERVIEW

AWS LAMBDA

- Compute service that lets you run code without provisioning or managing servers
- AWS Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second
- You pay only for the compute time you consume - there is no charge when your code is not running
- Currently supports Node.js, Java, C#, Go, Python
- You can use AWS Lambda to run your code:
  - in response to events, such as changes to data in an Amazon S3 bucket or an Amazon DynamoDB table
  - to run your code in response to HTTP requests using Amazon API Gateway
  - to invoke your code using API calls made using AWS SDKs
- When using AWS Lambda, you are responsible only for your code
- AWS Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources

LAMBDA LIFEYCLE

- **Authoring code for your Lambda function** – What languages are supported? Is there a programming model that I need to follow? How do I package my code and dependencies for uploading to AWS Lambda? What tools are available?
- **Uploading code and creating Lambda functions** – How do I upload my code package to AWS Lambda? How do I tell AWS Lambda where to begin executing my code? How do I specify compute requirements like memory and timeout?
- **Monitoring and troubleshooting** – For my Lambda function that is in production, what metrics are available? If there are any failures, how do I get logs or troubleshoot issues?

PROGRAMMING MODEL

- **Handler** – Handler is the function AWS Lambda calls to start execution of your Lambda function. You identify the handler when you create your Lambda function. When a Lambda function is invoked, AWS Lambda starts executing your code by calling the handler function. AWS Lambda passes any event data to this handler as the first parameter. Your handler should process the incoming event data and may invoke any other functions/methods in your code.
- **Context object** – AWS Lambda also passes a context object to the handler function, as the second parameter. Via this context object your code can interact with AWS Lambda. For example, your code can find the execution time remaining before AWS Lambda terminates your Lambda function.
- **Logging** – Your Lambda function can contain logging statements. AWS Lambda writes these logs to CloudWatch Logs. Specific language statements generate log entries, depending on the language you use to author your Lambda function code.
- **Exceptions** – Your Lambda function needs to communicate the result of the function execution to AWS Lambda. Depending on the language you author your Lambda function code, there are different ways to end a request successfully or to notify AWS Lambda an error occurred during execution. If you invoke the function synchronously, then AWS Lambda forwards the result back to the client.

FUNCTION HANDLER

```
def my_handler (event, context):
        message = 'Hello {} {}!'.format(event['first_name'],
        event['last_name'])
        return { 'message' : message }
```

- **event** – AWS Lambda uses this parameter to pass in event data to the handler. This parameter is usually of the Python dict type. It can also be list, str, int, float, or NoneType type.
- **context** – AWS Lambda uses this parameter to provide runtime information to your handler. This parameter is of the LambdaContext type.
- Optionally, the handler can return a value. What happens to the returned value depends on the invocation type you use when invoking the Lambda function:
  - If you use the RequestResponse invocation type (synchronous execution), AWS Lambda returns the result of the Python function call to the client invoking the Lambda function (in the HTTP response to the invocation request, serialized into JSON). For example, AWS Lambda console uses the RequestResponse invocation type, so when you invoke the function using the console, the console will display the returned value.
  - If the handler returns NONE, AWS Lambda returns null.
  - If you use the Event invocation type (asynchronous execution), the value is discarded.

CONTEXT OBJECT

```
def get_my_log_stream(event, context):
        print("Log stream name:", context.log_stream_name)
        print("Log group name:", context.log_group_name)
        print("Request ID:",context.aws_request_id)
        print("Mem. limits(MB):", context.memory_limit_in_mb)
        time.sleep(1)
        print("Time remaining (MS):", context.get_remaining_time_in_millis())
```

- While a Lambda function is executing, it can interact with the AWS Lambda service to get useful runtime information such as:
  - How much time is remaining before AWS Lambda terminates your Lambda function (timeout is one of the Lambda function configuration properties).
  - The CloudWatch log group and log stream associated with the Lambda function that is executing.
  - The AWS request ID returned to the client that invoked the Lambda function. You can use the request ID for any follow up inquiry with AWS support.
  - If the Lambda function is invoked through AWS Mobile SDK, you can learn more about the mobile application calling the Lambda function.

CONTEXT OBJECT

- **function_name**: Name of the Lambda function that is executing
- **function_version**: The Lambda function version that is executing
- **invoked_function_arn:** The ARN used to invoke this function. It can be function ARN or alias ARN
- **memory_limit_in_mb**: Memory limit, in MB, you configured for the Lambda function. You set the memory limit at the time you create a Lambda function and you can change it later.
- **aws_request_id**: AWS request ID associated with the request. This is the ID returned to the client that called the invoke method.
- **log_group_name:** The name of the CloudWatch log group where you can find logs written by your Lambda function.
- **log_stream_name**: The name of the CloudWatch log stream where you can find logs written by your Lambda function. The log stream may or may not change for each invocation of the Lambda function.
- **identity**: Information about the Amazon Cognito identity provider when invoked through the AWS Mobile SDK. It can be null.
- **client_context**: Information about the client application and device when invoked through the AWS Mobile SDK. It can be null.

LOGGING

- Your Lambda function can contain logging statements
- AWS Lambda writes these logs to CloudWatch
- If you use the Lambda console to invoke your Lambda function, the console displays the same logs
- The following Python statements generate log entries:
  - print statements.
  - Logger functions in the logging module (for example, logging.Logger.info and logging.Logger.error)
- Both print and logging.* functions write logs to CloudWatch Logs but the logging.* functions write additional information to each log entry, such as time stamp and log level

FUNCTION ERRORS

- If your Lambda function raises an exception, AWS Lambda recognizes the failure and serializes the exception information into JSON and returns it
- Stack trace is returned as the stackTrace JSON array of stack trace elements
- How you get the error information back depends on the invocation type that the client specifies at the time of function invocation:
  - If a client specifies the RequestResponse invocation type (that is, synchronous execution), it returns the result to the client that made the invoke call.
    - For example, the console always use the RequestResponse invocation type, so the console will display the error in the **Execution result** section
    - The same information is also sent to CloudWatch and the **Log output** section shows the same logs.
  - If a client specifies the Event invocation type (that is, asynchronous execution), AWS Lambda will not return anything. Instead, it logs the error information to CloudWatch Logs. You can also see the error metrics in CloudWatch Metrics.

DEPLOYMENT PACKAGE

- To create a Lambda function you first create a Lambda function deployment package, a .zip or .jar file consisting of your code and any dependencies
- When creating the zip, include only the code and its dependencies, not the containing folder. You will then need to set the appropriate security permissions for the zip package.
- Zip packages uploaded with incorrect permissions may cause execution failure.
  - AWS Lambda requires global read permissions on code files and any dependent libraries that comprise your deployment package
- You can create a deployment package yourself or write your code directly in the Lambda console, in which case the console creates the deployment package for you and uploads it, creating your Lambda function

DEPLOYMENT PACKAGE

- Scenarios:
  - **Simple scenario** – If your custom code requires only the AWS SDK library, then you can use the inline editor in the AWS Lambda console. Using the console, you can edit and upload your code to AWS Lambda. The console will zip up your code with the relevant configuration information into a deployment package that the Lambda service can run. You can also test your code in the console by manually invoking it using sample event data.
  - **Advanced scenario** – If you are writing code that uses other resources, such as a graphics library for image processing, or you want to use the AWS CLI instead of the console, you need to first create the Lambda function deployment package, and then use the console or the CLI to upload the package.
- After you create a deployment package, you may either upload it directly or upload the .zip file first to an Amazon S3 bucket in the same AWS region where you want to create the Lambda function, and then specify the bucket name and object key name when you create the Lambda function using the console or the AWS CLI.

ACCESSING AWS RESOURCES

- Lambda does not enforce any restrictions on your function logic – if you can code for it, you can run it within a Lambda function. As part of your function, you may need to call other APIs, or access other AWS services like databases
- To access other AWS services, you can use the AWS SDK (Node.js, Java, Python, C#)or Go, AWS Lambda will automatically set the credentials required by the SDK to those of the IAM role associated with your function – you do not need to take any additional steps
- You can include any SDK to access any service as part of your Lambda function. For example, you can include the SDK for Twilio to access information from your Twilio account. You can use Environment Variables for storing the credential information for the SDKs after encrypting the credentials.

ACCESSING AWS RESOURCES

- By default, your service or API must be accessible over the public internet for AWS Lambda to access it.
- However, you may have APIs or services that are not exposed this way. Typically, you create these resources inside Amazon Virtual Private Cloud (Amazon VPC) so that they cannot be accessed over the public Internet.
  - These resources could be AWS service resources, such as Amazon Redshift data warehouses, Amazon ElastiCache clusters, or Amazon RDS instances.
  - They could also be your own services running on your own EC2 instances. By default, resources within a VPC are not accessible from within a Lambda function.
- AWS Lambda runs your function code securely within a VPC by default.
  - However, to enable your Lambda function to access resources inside your private VPC, you must provide additional VPC-specific configuration information that includes VPC subnet IDs and security group IDs.
  - AWS Lambda uses this information to set up elastic network interfaces (ENIs) that enable your function to connect securely to other resources within your private VPC.

ACCESSING VPC RESOURCES

- You add VPC information to your Lambda function configuration using theVpcConfig parameter, either at the time you create a Lambda function (see CreateFunction), or you can add it to the existing Lambda function configuration (see UpdateFunctionConfiguration).
- When you add VPC configuration to a Lambda function, it can only access resources in that VPC. If a Lambda function needs to access both VPC resources and the public Internet, the VPC needs to have a Network Address Translation (NAT) instance inside the VPC
- When a Lambda function is configured to run within a VPC, it incurs an additional ENI start-up penalty. This means address resolution may be delayed when trying to connect to network resources
- Your Lambda function automatically scales based on the number of events it processes
  - If your Lambda function accesses a VPC, you must make sure that your VPC has sufficient ENI capacity to support the scale requirements of your Lambda function
  - The subnets you specify should have sufficient available IP addresses to match the number of ENIs

ACCESSING THE INTERNET

- AWS Lambda uses the VPC information you provide to set up ENIs that allow your Lambda function to access VPC resources.
- Each ENI is assigned a private IP address from the IP address range within the Subnets you specify, but is not assigned any public IP addresses. Therefore, if your Lambda function requires Internet access (for example, to access AWS services that don't have VPC endpoints), you can configure a NAT instance inside your VPC or you can use the Amazon VPC NAT gateway.
- You cannot use an Internet gateway attached to your VPC, since that requires the ENI to have public IP addresses.
- If your Lambda function needs Internet access, do not attach it to a public subnet or to a private subnet without Internet access. Instead, attach it only to private subnets with Internet access through a NAT instance or an Amazon VPC NAT gateway.

EXECUTION MODEL

- When AWS Lambda executes your Lambda function on your behalf, it takes care of provisioning and managing resources needed to run your Lambda function
- When you create a Lambda function, you specify configuration information, such as the amount of memory and maximum execution time that you want to allow for your Lambda function
- When a Lambda function is invoked, AWS Lambda launches an Execution Context based on the configuration settings you provide. Execution Context is a temporary runtime environment that initializes any external dependencies of your Lambda function code, such as database connections or HTTP endpoints
- It takes time to set up an Execution Context and do the necessary "bootstrapping", which adds some latency each time the Lambda function is invoked. You typically see this latency when a Lambda function is invoked for the first time or after it has been updated because AWS Lambda tries to reuse the Execution Context for subsequent invocations of the Lambda function
- After a Lambda function is executed, AWS Lambda maintains the Execution Context for some time in anticipation of another Lambda function invocation. In effect, the service freezes the Execution Context after a Lambda function completes, and thaws the context for reuse, if AWS Lambda chooses to reuse the context when the Lambda function is invoked again

DEMO

- Accessing Amazon RDS in an Amazon VPC
  - https://docs.aws.amazon.com/lambda/latest/dg/vpc-tutorials.html

QUESTIONS?