Module 9
Serverless Computing: AWS Lambda – Part 1

Overview

In this module we will look at an AWS service known as **AWS Lambda**, which is part of the **Compute** family of services.

When thinking about computing in the cloud, the things that first come to mind are virtual machines (VMs). One provisions one or many VMs in the cloud and use them to run some computation (programs). Here are some use cases of why people provision VMs for computation.

a) As a desktop alternative (the simplest case): That is, you use a client software such as Remote Desktop Connection, SSH, etc. to remote connect to a machine in the cloud. You do most of your work on that machine. Your local machine is simply used to remote connect. The bulk of the work and computation is done on this cloud machine.

b) You need a GPU cluster with say 1 TB of memory, but you don't have such a thing on premise.

c) You need to test your software on say 10 different variants of Linux and Windows and you don't have all of them on premise. You can provision them and use them as your test machines.

d) You want a PC outside of your firewall – maybe to share with someone you are collaborating with.

e) Host an application, web service, or a website.

f) Etc.

VMs are part of **IaaS** (Infrastructure as a Service) - You are renting infrastructure from a cloud provider, and you are charged whether you use it or not. You are also responsible for any software, configuration, updates, and security patches that this VM needs. In many cases this might not be the optimal choice of doing computation. Let's look at three examples where choosing a VM for computation is a poor choice:

1. Assume that you developed an Alexa skill for your medium-size organization. The skill does a simple thing: it queries the organization's sales database and retrieves the total sale amount at that time of the day. Assuming your sales data is in a cloud database, one way to host this skill is to expose it as a web service hosted on a VM in the cloud. You configure Alexa to call this web service, which calls your database, computes the total sale value, and returns the value to Alexa. This is not a hypothetical story. One way to expose Alexa skills is as was described: by configuring the skill as a web service.

If 10 people a day in your organization use the skill, and if querying the database and computing the total sales amount takes roughly 250 milliseconds per call, then in total you are doing on the VM 2.5 seconds worth of computation per day (250 milliseconds per call x 10 calls). So you are paying for a VM for 24 hours to use it for 2.5 seconds. And on weekends and holidays, it is idle with no computation requested. Yet you are still being charged for those days.

2. You developed an educational app for use at schools and universities. The app allows students to upload images and your app stores the images in an S3 bucket. Schools are worried that some students might upload inappropriate pictures. You developed your own Machine Learning model that detects inappropriate pictures. The model validates a picture and rejects it if it is flagged as offensive. Your Machine Learning model takes 250 ms to validate a picture. Assume 100 pictures are uploaded daily, you are using a total of 25 seconds of computation per day (250 milliseconds per picture x 100 pictures). If you host the Machine Learning picture validation model on a VM, you are paying for 24 hours and using the VM for 25 seconds. Furthermore, if S3 does not support notifying your code on the VM, your code might need to periodically poll S3 for new images that have arrived (also not efficient).

3. Same as the example in (2), but now your educational app becomes popular worldwide and is used daily by millions of students. 100s of pictures are getting uploaded per second. Your code and VM weren't designed/configured for such heavy workload. You need to make improvements in the code and possibly use additional VMs to distribute the load. This by itself is challenging and require expertise.

An alternative way to run computation is to use what is known as serverless computing. The AWS service that gives you this capability is **AWS Lambda**. In what ways does serverless computing address the problems exposed in the above 3 examples?:

1. You do not rent hardware (VMs) and run code on it. You simply supply the code, and the cloud provider runs it for you on VMs it maintains. The VMs are not dedicated to you only. They are used to run your code and the code of other customers. So there is economy of scale: instead of you, the customer, absorbing the entire cost of that VM, the cost is shared among 100s of customers

2. There is no fixed cost. You pay for the amount of computation time. That is, if your code executes for a total of 2.5 seconds per day, you pay for 2.5 seconds worth of computation. And if on a holiday, no one uses your Alexa skill, you are not charged.

3. The cloud provider assumes the responsibility of auto scaling your function if loads increases. That is, you don't need to be an expert, or hire an expensive software expert, to design a system capable of handling millions of requests per second. This capability is built into how lambda works and you get that for free. You simply worry about what your code does, and

lambda takes care of auto-scaling it to support millions of requests per second.

4. It is simple. You upload code, and the cloud provider runs it for you.

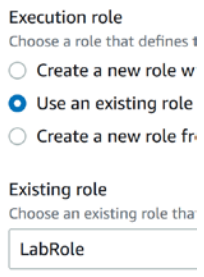
When you write an AWS Lambda function, the function can usually be invoked in several ways:

- In response to an event that happens in another service (e.g., S3, DynamoDB, etc.)
- Being called by another Lambda function.
- Used as the backend code of a REST API (in this case you use AWS Lambda with AWS Gateway – we will have one module that goes through such an example).
- As a cron job: lambda can be configured to run as a cron job (for example, once every 5 minutes to do something)

We can define AWS Lambda in simple terms: it is a service that allows you to upload code (all popular languages and runtimes are supported). The cloud provider (e.g., AWS, Azure, etc.) runs the code for you on VMs that they maintain and manage. That's what the word "serverless" mean in serverless computing. It is serverless in the sense that you (the developer who writes the code) don't need to rent and maintain a VM server in the cloud to run code – someone else does (the cloud provider). You only supply your code and pay for the amount of time your code runs. There is no upfront cost or things to maintain or set up.

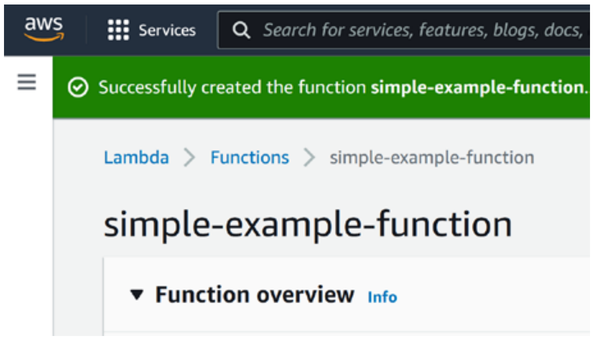
Create an AWS Lambda Function

1. Go to Canvas, start the lab, and click the AWS link to navigate to the AWS Console.
2. In the search for services textbox, enter **Lambda**. Then select it.
3. You should now be on the **AWS Lambda** service page. If you haven't created any function before it should say: Functions (0).
4. Click the **Create function** button.
5. Choose the **Author from scratch** option.
6. For the Function name, use: simple-example-function
7. You can use different languages. For this example, let's use Python.
   Select **Python 3.9** from the **Runtime** dropdown.
8. Expand the **Change default execution role**, choose **Use an existing role**, and select **LabRole** from the dropdown.


Execution role
Choose a role that defines t
Create a new role w [✓ Use an existing role] Create a new role fr
Existing role
Choose an existing role tha
LabRole

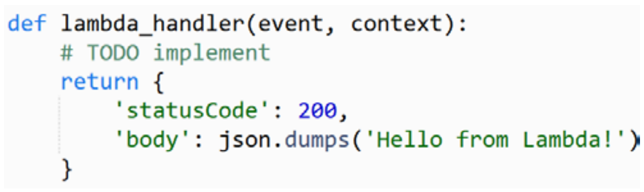
9. Click the **Create function** button.

   AWS creates the function for you and takes you to a new page, the simple-example-function page.


aws
Services
Search for services, features, blogs, docs,
Lambda > Functions > simple-example-function
simple-example-function
Function overview Info

Successfully created the function **simple-example-function**.

10. Scroll down and notice that some template code is already there. There is a file named **lambda_function.py** and in this file a function named **lambda_handler** with two arguments: `event` and `context` objects. The lambda_handler function is the function that Lambda calls when your function is invoked. So this lambda_handler function is the entry point to your code; when AWS executes your code it is going to start it in this function (you can think of it like main() of a regular program).

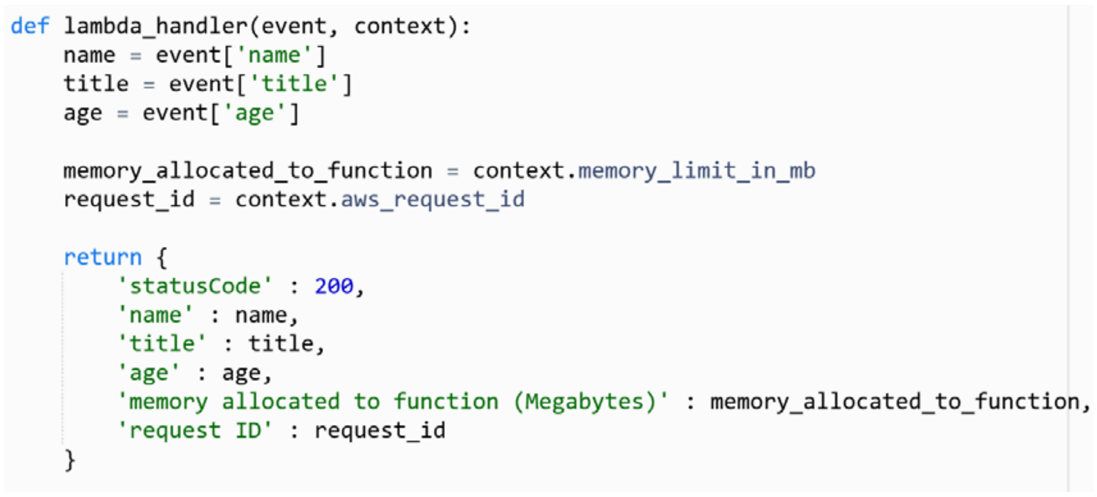


```
def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

The `event` argument represents the trigger that caused the lambda to be called. We said earlier that Lambda can be invoked in different ways. The AWS engine that manages your Lambda and runs the code will pass to you this argument, which gives you more information about the source of event. For example, if the Lambda was configured to execute when a file is dropped in a bucket, you probably need information about the bucket and the file that was dropped (maybe you need to read the file content and do some action). The `event` argument allows you to acquire this information.

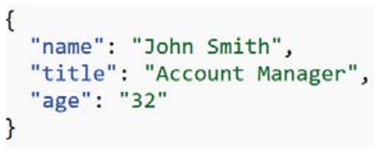
The `context` argument provides methods and properties that provide information about the invocation, function, and execution environment. For example, this argument can tell you the amount of memory allocated for the function. Go here to see all the fields that `context` gives you.

Change the lambda_handler function to make it look like this:



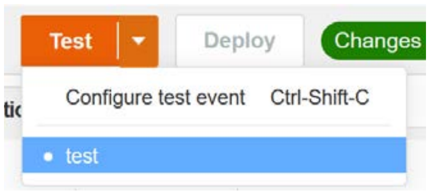
```python
def lambda_handler(event, context):
    name = event['name']
    title = event['title']
    age = event['age']

    memory_allocated_to_function = context.memory_limit_in_mb
    request_id = context.aws_request_id

    return {
        'statusCode' : 200,
        'name' : name,
        'title' : title,
        'age' : age,
        'memory allocated to function (Megabytes)' : memory_allocated_to_function,
        'request ID' : request_id
    }
```

11. Click the **Deploy** button to save and deploy your changes.
12. Click menu **Test → Configure test even**.
13. In the Event name textbox, enter: test.
14. Change the **Event JSON** to look like this:



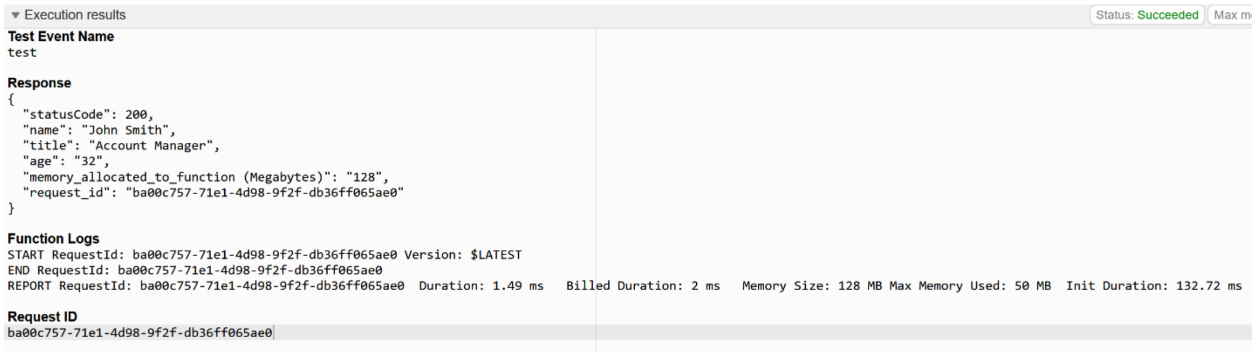
```json
{
  "name": "John Smith",
  "title": "Account Manager",
  "age": "32"
}
```

15. Click the **Save** button (bottom right).
16. Make sure menu item **Test → test** is selected



17. Click the orange **Test** button, and observe the result.

    You should see the input data printed along with 2 pieces of information you obtained from the context object. Also notice that Lambda calculated the time it takes to execute your code (1.49 ms in my case). Your output should look like this (your request ID will be different):



```
Test Event Name
test

Response
{
  "statusCode": 200,
  "name": "John Smith",
  "title": "Account Manager",
  "age": "32",
  "memory_allocated_to_function (Megabytes)": "128",
  "request_id": "ba00c757-71e1-4d98-9f2f-db36ff065ae0"
}

Function Logs
START RequestId: ba00c757-71e1-4d98-9f2f-db36ff065ae0 Version: $LATEST
END RequestId: ba00c757-71e1-4d98-9f2f-db36ff065ae0
REPORT RequestId: ba00c757-71e1-4d98-9f2f-db36ff065ae0  Duration: 1.49 ms   Billed Duration: 2 ms   Memory Size: 128 MB Max Memory Used: 50 MB  Init Duration: 132.72 ms

Request ID
ba00c757-71e1-4d98-9f2f-db36ff065ae0
```

Also notice that in my case the AWS Lambda engine allocated 128 MB of memory that my function can utilize.

In this simple Lambda function exercise, we typed the code in the AWS Console and the function doesn't have any dependencies on external packages. But it doesn't have to be that way. In future modules, you will author a Lambda function in Visual Studio (or VSCode) and upload the function to your account. Your Lambda function can have other classes, depend on external packages, etc.

18. Before we continue with this simple example, let's step back and contemplate how this relates to the first 2 examples on pages 1 and 2. Remember that we said serverless computing such as AWS Lambda allows us to run code without provisioning a VM for it. And this is exactly what we did here in our **simple-example-function** Lambda function. All we provided is some Python code. And AWS ran it for us on some computer that we don't care about.

    The word "serverless" is somewhat peculiar when used in this context. Of course the function is going to run on some server. So why is it "serverless" computing? The idea of serverless is that we, the developers who write code, are not aware of or care about provisioning some sort of VM server computer to run the code. We simply provide the code, and the cloud provider is responsible for provisioning and scaling the hardware on which our code runs. For the programmer, there is no administration to do, and therefore, is "serverless".

    One important concept to keep in mind is that when AWS deploys your Lambda function, it is likely to deploy it on multiple servers. Not only that, but there is no guarantee that subsequent calls to the function will hit the same server. The server on which your Lambda function is deployed is also hosting Lambdas of other customers (of course compartmentalized as to not interfere with each other's). Depending on the loads, AWS might route subsequent calls to the function to a different server to load balance workloads. From the AWS documentation:

    "Lambda manages the infrastructure that runs your code, and scales automatically in response to incoming requests. When your function is invoked more quickly than a single instance of your function can process events, Lambda scales up by running additional instances."

“*Your Lambda function code must be written in a stateless style, and have no affinity with the underlying compute infrastructure.*”

Two important lessons to take from the above documentation text:

Notice that Lambda automatically solves problem # 3 on page 2. If our application becomes very popular we don’t have to engineer scalability ourselves. Lambda provides this to us out of the box: “*When your function is invoked more quickly than a single instance of your function can process events, Lambda scales up by running additional instances*”.

And the last sentence: “… function code must be written is a stateless style”. This means, for example, that you should not save something in memory or on disk since there is no guarantee that 2 separate invocations of your code necessarily execute on the same machine.

CloudWatch

In a traditional compute setting where we run our programs on a local computer or a cloud VM that we provision (and have access to), we can always inspect the logs of a program to troubleshoot issues and problems. But how can this be done in a Lambda function when we don’t have access to the computer on which our Lambda function code is running. Remember it is “serverless” for us. We have no information about this computer. Any non-trivial function is likely to need some level of logging for troubleshooting purposes, or to log metrics that can later be queried and aggregated to provide insights about our application.

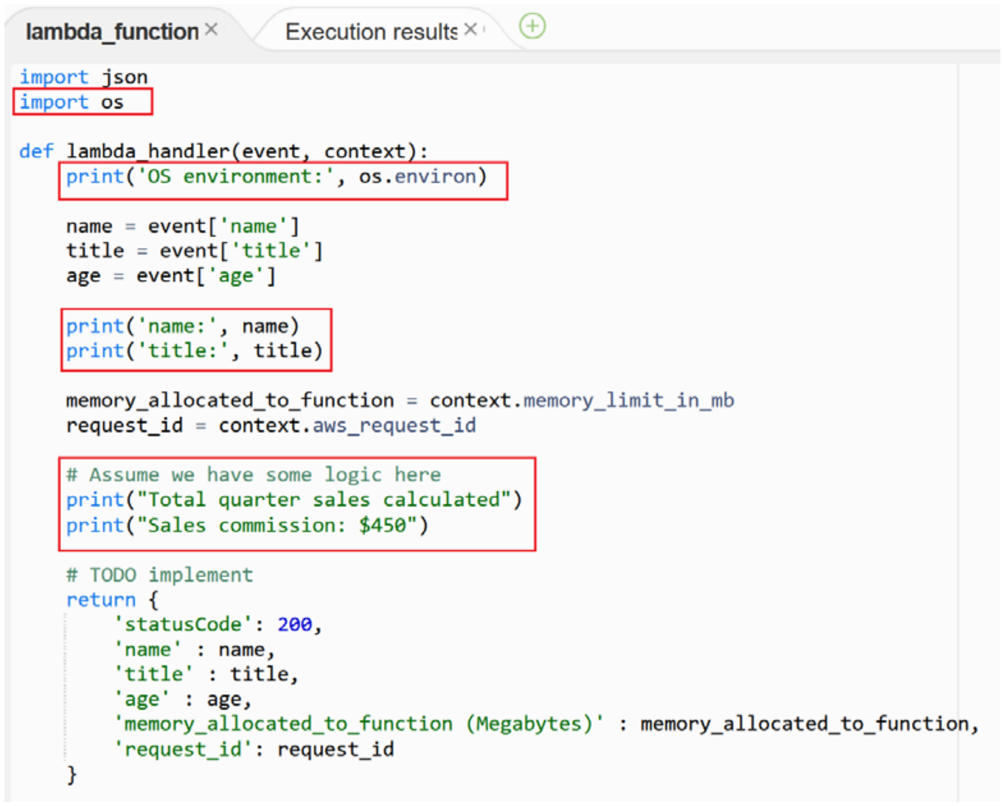
CloudWatch is an AWS service for applications monitoring. In addition to logs, it has many other monitoring and traceability features. Here is an introduction description from AWS:

“*Amazon CloudWatch is a monitoring and observability service built for DevOps engineers, developers, site reliability engineers (SREs), and IT managers. CloudWatch provides you with data and actionable insights to monitor your applications, respond to system-wide performance changes, optimize resource utilization, and get a unified view of operational health. CloudWatch collects monitoring and operational data in the form of logs, metrics, and events, providing you with a unified view of AWS resources, applications, and services that run on AWS and on-premises servers. You can use CloudWatch to detect anomalous behavior in your environments, set alarms, visualize logs and metrics side by side, take automated actions, troubleshoot issues, and discover insights to keep your applications running smoothly.*”

In this particular example, we will use CloudWatch to log some information from the simple-example-function Lambda function you wrote earlier (we will explore some of CloudWatch’s other capabilities and features in future modules).

1. Go back to the AWS Console, the Lambda service, and click the simple-example-function to open it.

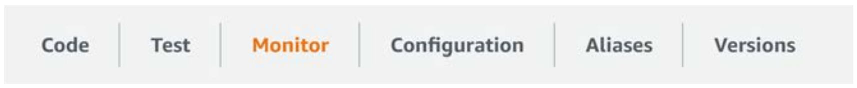
2. To output text to CloudWatch, use the Python `print()` method (or any logging library that writes to `stdout` or `stderr`). Change the code of function `lambda_handler` by adding a few print statements (highlighted in red below):



```python
import json
import os

def lambda_handler(event, context):
    print('OS environment:', os.environ)

    name = event['name']
    title = event['title']
    age = event['age']

    print('name:', name)
    print('title:', title)

    memory_allocated_to_function = context.memory_limit_in_mb
    request_id = context.aws_request_id

    # Assume we have some logic here
    print("Total quarter sales calculated")
    print("Sales commission: $450")

    # TODO implement
    return {
        'statusCode': 200,
        'name' : name,
        'title' : title,
        'age' : age,
        'memory_allocated_to_function (Megabytes)' : memory_allocated_to_function,
        'request_id': request_id
    }
```

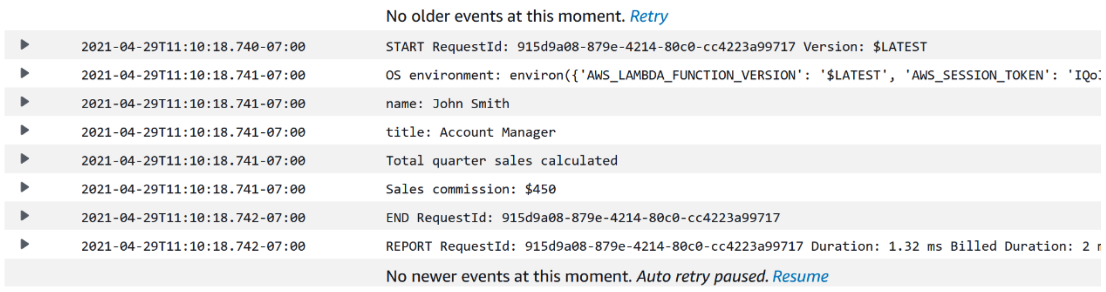
3. Click the **Deploy** button, then click the **Test** button.

4. Click the **Monitor** tab.



5. Click the **View logs in CloudWatch** button.

6. Click the most recent log stream link and observe the logging information that you outputted using `print()`.



```
No older events at this moment. Retry
▶  2021-04-29T11:10:18.740-07:00  START RequestId: 915d9a08-879e-4214-80c0-cc4223a99717 Version: $LATEST
▶  2021-04-29T11:10:18.741-07:00  OS environment: environ({'AWS_LAMBDA_FUNCTION_VERSION': '$LATEST', 'AWS_SESSION_TOKEN': 'IQo
▶  2021-04-29T11:10:18.741-07:00  name: John Smith
▶  2021-04-29T11:10:18.741-07:00  title: Account Manager
▶  2021-04-29T11:10:18.741-07:00  Total quarter sales calculated
▶  2021-04-29T11:10:18.741-07:00  Sales commission: $450
▶  2021-04-29T11:10:18.742-07:00  END RequestId: 915d9a08-879e-4214-80c0-cc4223a99717
▶  2021-04-29T11:10:18.742-07:00  REPORT RequestId: 915d9a08-879e-4214-80c0-cc4223a99717 Duration: 1.32 ms Billed Duration: 2 r
No newer events at this moment. Auto retry paused. Resume
```

7. Expand the os environment line arrow to see additional information.

8. Go back to the **simple-example-function** Lambda function and click the **Monitor** tab.
9. Look at the few built-in graphs. These can be used as a first and quick look at the health of your function. For instance, if you see unusual spikes in the Duration graph, it might indicate a possible problem. Is your function calling some other service? Is that other service experiencing delays, etc? You can even set alarms that trigger actions. For example, if such and such metric is exceeded, or if this threshold value is exceeded send a text message, an email, or open a ticket in a ticketing system. These are common operations cloud programmers do to monitor the health of their code and service.

   Hover your mouse over some of the graph labels to see additional popups with additional information.

   We will look at other CloudWatch examples in future modules. The purpose of introducing it here is for you to know how to log from a Lambda function and where that output ends up.

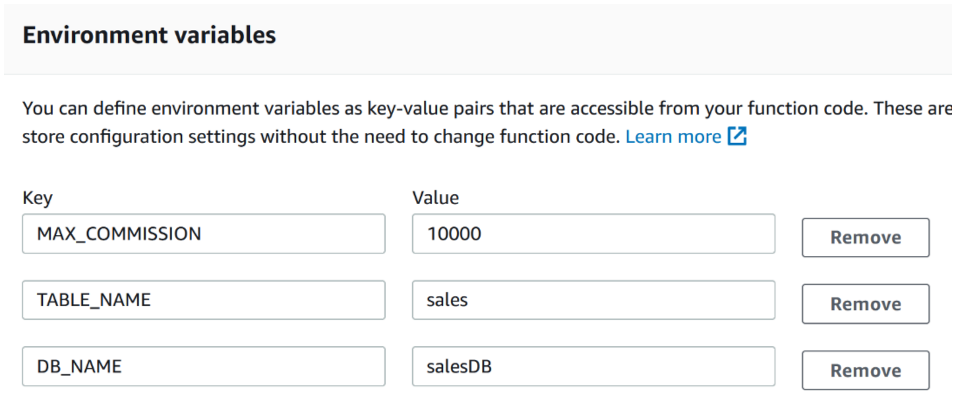
Environment and Configuration Variables for Lambda Function

A traditional program usually depends on some variables that aren't hardcoded in a program. A common strategy is to have some configuration file (e.g., config.xml) that has values of configuration settings the program depends on. At startup the program reads those settings and use them accordingly. To change the settings we edit the config.xml file and the program reads the new settings again on startup. This is usually done to make the program versatile, configurable, and to avoid having to recompile and install when said variables need to take on different values.

AWS Lambda functions also are likely to depend on configuration variables. It is also desirable to change the values of such variables without the need to change the Lambda function code. AWS Lambda provides you with the ability to achieve the same but in a different way.

a) Go back to the **simple-example-function** page, click the **Configuration** tab, then click **Environment variables** from the left.

b) Click the **Edit** button. Then click the **Add environment variable** button and add 3 environment variables:



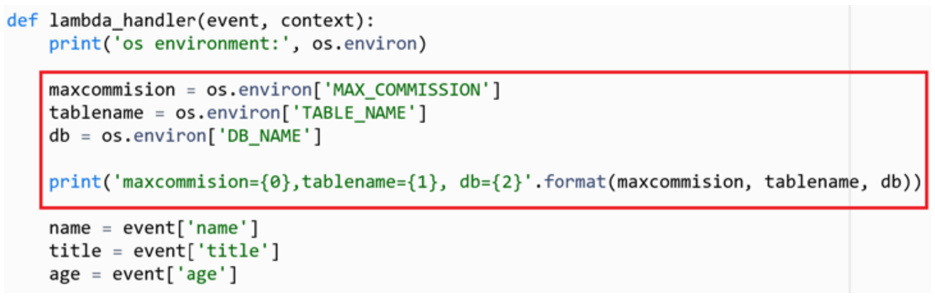
Environment variables

You can define environment variables as key-value pairs that are accessible from your function code. These are
store configuration settings without the need to change function code. Learn more

Key	Value	
MAX_COMMISSION	10000	Remove
TABLE_NAME	sales	Remove
DB_NAME	salesDB	Remove

c) Click the **Save** button to save the environment variables you just added.

d) In the Lambda code, read the environment variables using code as shown below:



```
def lambda_handler(event, context):
    print('os environment:', os.environ)

    maxcommision = os.environ['MAX_COMMISSION']
    tablename = os.environ['TABLE_NAME']
    db = os.environ['DB_NAME']

    print('maxcommision={0},tablename={1}, db={2}'.format(maxcommision, tablename, db))

    name = event['name']
    title = event['title']
    age = event['age']
```

e) Click the **Save** button, then click the **Test** button.

f) Go to CloudWatch and inspect the log. You should see the values of the environment variables:

Any Lambda configurable variable should be defined as such (and not hardcoded in the code).

Role used to execute Our Function

You might ask: how was our function allowed to write to CloudWatch? Can our Lambda do anything it wants in AWS? The answer is you have to give it permissions to do so. These permissions (IAM policies) are associated with an IAM Role associated with our Lambda function. In this case it is the **LabRole** that AWS Academy created for us.

1. Click the **Configuration** tab. Then click **Permissions** from the left.

2. You should see the **LabRole**. Click it.
3. It will take you to this role in the IAM service page. You should see several policies attached to this role. You should see here how IAM security primitives (roles and policies) are used to control what we can do in AWS.

Summary

We have looked at **S3** in the previous module. In this module we looked at the compute service **AWS Lambda**. In the next module we will connect them together and look at a white paper that describes a real-world case of how they are used together. We can also simulate solving example (2) from pages 1 and 2 (validating if an image has no offensive content – of course without the machine learning part).

Lambda is widely used in custom cloud applications and in AWS services built on top of other AWS services. We will be using it throughout the quarter. This is Part 1 of Lambda - we will do additional modules and explore other features that you will need.

You should start to notice a pattern emerging. We moved storage out of a local computer disk into a cloud storage (S3). And with Lambda, we moved computation out of a local computer CPU into the cloud. Bit by bit we are moving operations out of local computers (storage, execution) into the cloud. And as applications become more complex they need other things too: databases, queues, distributed hash tables, etc. And all these can be cloud resources too.

What to Submit:

Complete **Module9-Quiz** after you do the module.