Minimum requirements for this course:

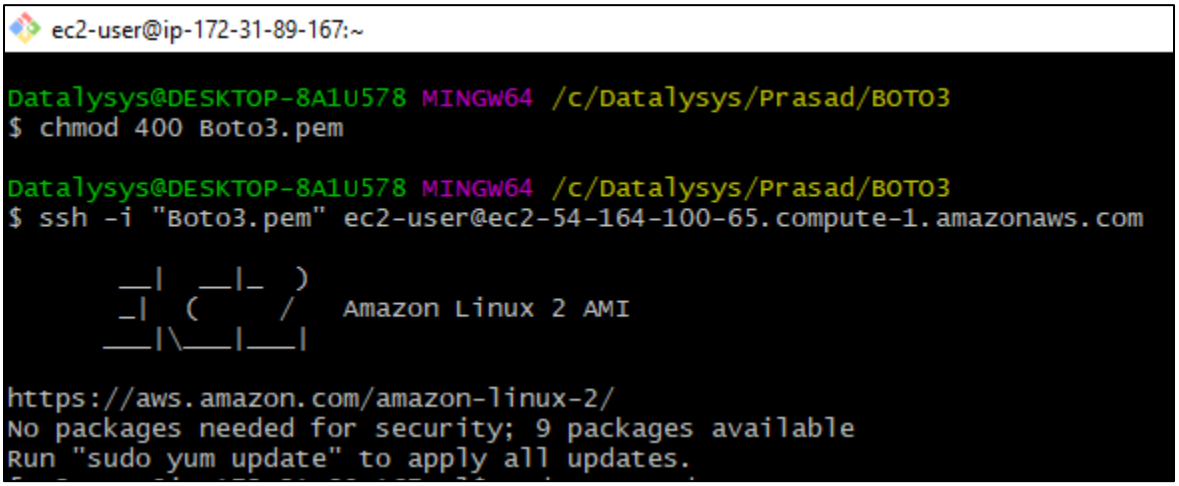
AWS Account and some knowledge on Python

What is BOTO3?

- ➔ It is the name of the Python SDK/ Module for AWS for creating, deleting AWS Services
- ➔ Boto3 is built on the top of BotoCore Module.
- ➔ We need to install the boto3 module in our local machine by typing "**pip install boto3**"
- ➔ Download the required Python version in your OS by navigating to [www.python.org/downloads](www.python.org/downloads)

Installing BOTO3 and Python in AWS EC2 Instance:

- ➔ Create a Free tier EC2 instance in AWS Management Console.
- ➔ After creating the instance, choose an existing keypair.
- ➔ Go to instances section, and wait for the instance to be created.
- ➔ After that, click on that instance and select "Connect" button on the top, you will get an ssh URL for launching the instance.
- ➔ Use GIT BASH to run the instance in windows machine, first use "**chmod 400 <keypair file>**" to provide necessary permissions for the keypair file.
- ➔ Make sure that the keypair file is present in the present folder in windows.
- ➔ Copy the SSH URL and paste it in your command line and hit Enter to start the EC2 instance.




- ➔ Run "**python --version**" to get the current python version, it will be normally 2.7 by default.
- ➔ Now we are required to have the python version 3.7 installed in the ec2 instance, so we will be following the steps provided in the Installation steps.txt present in the boto3 folder and executing the commands one by one.

➔ After finishing the steps provided in the txt document, try to import the boto3 module, it should not throw any errors.

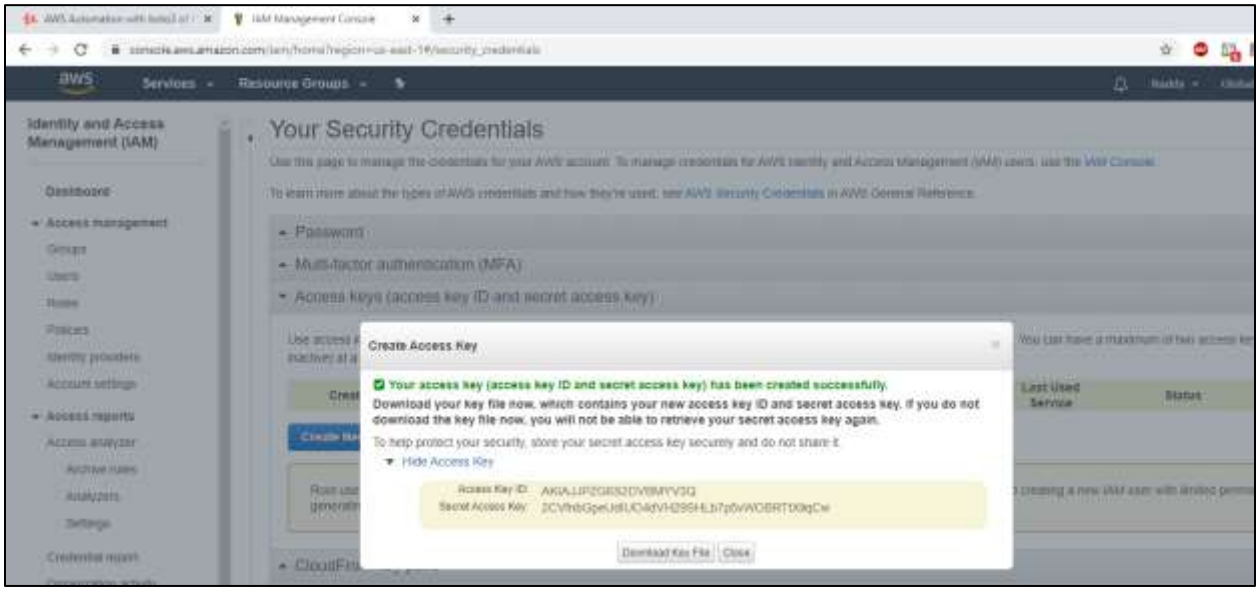
```
[root@ip-172-31-89-167 bin]# python3
Python 3.7.4 (default, May 10 2020, 08:12:12)
[GCC 7.3.1 20180712 (Red Hat 7.3.1-6)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import boto3
>>>
```

➔ This BOTO3 is installed only for python3 and not for python2, so make sure to run it using python3 only.

**Setting up the BOTO3 Environment for local Windows machine:**

➔ Since our BOTO3 is not aware of our AWS account, we need to install the programmatic access to link our account.

➔ We can get it from our AWS Management Console, Go to the console and select your username and navigate to “Security Credentials” and you need to create the access keys and select “Show Access Key” to get the access keys.




➔ It is best practise to do it with IAM users, rather than doing it with root user, so create a new user in the IAM section and provide the user with programmatic access.

➔ Provide the S3full permissions policy for the user.

➔ We get the access keys for the IAM user when we hit on Create-user button while creating the user.

- ➔ We can even create the access keys by selecting user and going to “Security-Credentials” tab and clicking on Create Access key.
- ➔ Now as we have the keys, we need to link the keys using AWSCLI commands, for that we need to install the AWSCLI using “**pip3 install awscli**”
- ➔ We can configure the Keys with account using the command “**aws configure –profile <IAM UserName>**”
- ➔ We can also use the “**aws configure**” shorthand command and it will ask for the access key, secret key, default region and output format.
- ➔ We will provide the root user’s access key, secret key and default region would be us-east-1 by default and output format will be passed as json
- ➔ Now there is a default directory created with the name “.aws” type, “**cd .aws**” to navigate to that directory and type “dir” to get the list of the files present in them.
- ➔ Now type “**type config**” to get the configuration details and “**type credentials**” to get the keys.
- ➔ If we want to configure for more than one users, then using “aws configure” shorthand command will not work, as it will overwrite the exisiting one, so will re-create the new aws configs, first we need to delete the present aws configs.
- ➔ We can do that by navigating to C:\Users\<Username>\ and delete the .aws file.
- ➔ Now we will be using the complete command to create the configurations, type “**aws configure –profile <IAM Username or root>**” and provide the keys, region and output format.
- ➔ Use “**type config**” and “**type credentials”** to get the information related to profile and keys.

```
C:\Users\Datalysys\.aws>type config
[profile Practise]
region = us-east-1
output = json

C:\Users\Datalysys\.aws>type credentials
[Practise]
aws_access_key_id = AKIAWXPNV524P4CYCHWM
aws_secret_access_key = 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
```

**Setting up BOTO3 environment in Ec2 instance:**

- ➔ Connect to the Ec2-instance using the GITBASH and then type "**sudo pip3 install awscli**"
- ➔ Configure the aws keys using the command "aws configure –profile <IAM USER or root> and pass the keys, region and output format.
- ➔ Now type "ls -a" to get the list of directories, there you can find the .aws directory created.
- ➔ Navigate to that using "cd .aws" and type "cat config" and "cat credentials" to view the keys and profile configurations.

```
[root@ip-172-31-89-167 .aws]# cat config
[profile Practise]
output = json
region = us-east-1
[root@ip-172-31-89-167 .aws]# cat credentials
[Practise]
aws_access_key_id = AKIAWXPNV524P4CYCHWM
aws_secret_access_key = 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
[root@ip-172-31-89-167 .aws]#
```

**Verifying the Environment with a simple script:**

- ➔ We will be writing a BOTO3 script to list all the IAM users in our console.

The Algorithm is as follows:

***STEP 1: Go to the AWS Management console. We will be writing the BOTO3 command for this***

**import boto3**

**AWS_MGMT_CONSOLE = boto3.session.Session(profile_name="<ProfileName here>")**

1a. We will be importing the BOTO3 module.

1b. We will be creating a BOTO3 session with our configured profile name i.e., IAM user or Root.

***STEP 2: Go to the IAM service in the AWS Console***.

**IAM_CON_AWS = AWS_MGMT_CONSOLE.resource('iam')**

2a. We will be calling the resource function for our AWS Management Variable and passing the service we want to work on. Here in our case it is IAM.

***STEP 3: Iterating over the IAM_CON_AWS to get the list of users.***

**for each_user in IAM_CON_AWS.users.all():**

**print(each_user.name)**

3a. We will be iterating over the IAM_CON_AWS and calling users.all() to get all the users and printing the name of each users on the terminal.

The sample code as written in VSCODE looks like this: (filename : AWS_Sample_Boto3_script.py)

```
import boto3        #Importing BOTO3 module

AWS_MGMT_CONSOLE = boto3.session.Session(profile_name="root")   #Accessing AWS Console with root credentials
IAM_CONSOLE_AWS = AWS_MGMT_CONSOLE.resource('iam')  #Opening the IAM service

for each_user in IAM_CONSOLE_AWS.users.all():       #Getting all the users from IAM
    print(each_user.name)                           #Printing the users names
```

The output of the program will be like this:

```
Practise
User1
```

Let us write another code to list all the S3 Buckets present in our AWS Console, we will be refactoring the code as functions to re-use the functions, our final code should look like this:

```
import boto3        #Importing BOTO3 module

AWS_MGMT_CONSOLE = boto3.session.Session(profile_name="root")   #Accessing AWS Console with root credentials

def ListIAMUsers():
    try:
        IAM_CONSOLE_AWS = AWS_MGMT_CONSOLE.resource('iam')  #Opening the IAM service

        for each_user in IAM_CONSOLE_AWS.users.all():       #Getting all the users from IAM
            print(each_user.name)                           #Printing the users names
    except Exception as e: print("Error listing users from IAM")

def ListS3Buckets():
    try:
        S3_CONSOLE_AWS = AWS_MGMT_CONSOLE.resource('s3')        #Opening the S3 Service

        for each_bucket in S3_CONSOLE_AWS.buckets.all():        #Getting all the S3 buckets from S3
            print(each_bucket.name)                                #Printing each buckets

    except Exception as e: print("Error listing the S3 Buckets")
```

The output for s3 buckets will be like this:

```
cf-templates-4u351hjm9psg-us-east-1
samplebucketforprogram
```

Lets test the same from the EC2-instance also, copy the code and paste it as a file in ec2-instance.

1. Type **vim sample_boto3_script.py** to open a vim editor with the specified python file.
2. Copy the exact code from the Local and paste it here.
3. Now type **Esc** and type **:wq** to save the code.
4. Type **python3 sample_boto3_script.py** to run the program and we get exact same results.
5. Don't forget to add the #!/usr/bin/python3 in the first line of the code in linux system and type **chmod +x <filename>** to provide the execute permissions for the file.

```
[root@ip-172-31-89-167 boto3scripts]# which python3
/bin/python3
[root@ip-172-31-89-167 boto3scripts]# ^C
[root@ip-172-31-89-167 boto3scripts]# vim sample_boto3_script.py
[root@ip-172-31-89-167 boto3scripts]# chmod +x sample_boto3_script.py
[root@ip-172-31-89-167 boto3scripts]# ./sample_boto3_script.py
Practise
User1
cf-templates-4u351hjm9psg-us-east-1
[root@ip-172-31-89-167 boto3scripts]#
```

**BOTO3 Concepts:**

The main concepts of the BOTO3 are:

a. Session
b. Resource
c. Client
d. Meta
e. Collections
f. Waiters
g. Paginators

**Session:**

- ➔ Generally, this is our AWS Management Console itself, It stores the configuration information based on our credentials passed (basically our aws configuration profiles)
- ➔ It allows us to create service clients and resources.
- ➔ It creates default session for us when needed.

**Resource and Client:**

- ➔ We can create a service of AWS such as EC2, S3, IAM etc by using resource and clients.
- ➔ We can pass the service name and region name as keyword arguments for the resource, if nothing is passed for region_name then, the region specified in the aws profile configuration is taken.
- ➔ The difference between resource and client is as follows:

Resource is useful for only limited services for AWS such as : **CloudFormation, CloudWatch, DynamoDB, EC2, Glacier, IAM, Opsworks, S3, SNS, SQS** and it is an High-Level-Access that deals with Object Type data.

Clients are useful to get all kinds of services in AWS and it is an low-level-access and it deals with dictionary kinds of data.

Example for working with Clients:

We get a lot of data, in form of dictionaries and we need to search inside each key to get the required information, the sample code is like this:

**iam_con_cli=aws_mag_con_root.client(service_name='iam',region_name="us-east-2")**

**for each in iam_con_cli.list_users()['Users']:**

**print(each['UserName'])**

**Types of BOTO3 Sessions:**

a. Custom Session
b. Default Session

**Custom session:**

- ➔ This is what we did until now, taking the programmatic access keys from console for root and IAM users, creating aws configure files and using boto3.session.Session(profile_name= "IAM user or root")
- ➔ Next, we have invoked the AWS services using either Resource or Client.

**Default Session:**

- ➔ This session will be created by BOTO3 itself.
- ➔ We create the aws configuration files using the command "**aws config**" and enter the root programmatic access keys. This will create the default profile.
- ➔ We will not be including the line for the boto3 sessions, instead we will be writing the boto3 instead of the aws_management variable.
- ➔ Refer the ListIAMUsersWithoutDefaultSession() function for more details in the sample script.

We can have both Custom and Default Sessions in a single script, but it is a best practice to use Custom sessions instead of Default Sessions.

**Implementing Python BOTO3 scripts using client Objects:**

- ➔ First create the custom session using a profile name for accessing AWS management console.
- ➔ Next call the client object and pass the service name and region name as keyword arguments to the client object.

```
def AWSServicesUsingClientObject():
    try:
        IAM_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='iam', region_name='us-east-1')
        EC2_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='ec2', region_name='us-east-1')
        S3_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='s3', region_name='us-east-1')
```

➔ Suppose, we want to list all the users in our IAM service, we can use the list_users() function to get the users information by using client object and as the response is in the form of a JSON, we will be storing it in a response variable.

```
IAMResponse = IAM_CONSOLE_AWS.list_users()
print(IAMResponse)
```

➔ If we print the output, we need to get the output like the following figure.

```
{
    'Users': [
        {
            'Path': 'string',
            'UserName': 'string',
            'UserId': 'string',
            'Arn': 'string',
            'CreateDate': datetime(2015, 1, 1),
            'PasswordLastUsed': datetime(2015, 1, 1),
            'PermissionsBoundary': {
                'PermissionsBoundaryType': 'PermissionsBoundaryPolicy',
                'PermissionsBoundaryArn': 'string'
            },
            'Tags': [
                {
                    'Key': 'string',
                    'Value': 'string'
                },
            ]
        },
    ],
    'IsTruncated': True|False,
    'Marker': 'string'
}
```

➔ Suppose we are interested only in the 'Users' key values, then we can print only that key results like this:

```
IAMResponse = IAM_CONSOLE_AWS.list_users()
print(IAMResponse['Users'])
```

➔ We get the values present such as Path, Username, Arn, CreateDate etc like this in a form of json:

```
[{
    "Path": "/",
    "UserName": "Practise",
    "UserId": "AIDAWXPNV524CYEG5XFGP",
    "Arn": "arn:aws:iam::462744317624:user/Practise",
    "CreateDate": datetime.datetime(2020, 5, 10, 9, 13, 6, tzinfo = tzutc())
}, {
    "Path": "/",
    "UserName": "User1",
    "UserId": "AIDAWXPNV524FNKSN45VO",
    "Arn": "arn:aws:iam::462744317624:user/User1",
    "CreateDate": datetime.datetime(2019, 8, 24, 7, 51, 4, tzinfo = tzutc())
}]
```

➔ Now we will be directly printing just the usernames from this json object by writing this code:

```
def AWSServicesUsingClientObject():
    try:
        IAM_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='iam', region_name='us-east-1')
        EC2_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='ec2', region_name='us-east-1')
        S3_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='s3', region_name='us-east-1')

        IAMResponse = IAM_CONSOLE_AWS.list_users()
        for each_user in IAMResponse['Users']:
            print("UserName : "+each_user['UserName'])

    except Exception as e: print("Error listing users from IAM")
```

➔ We get the following output:

```
C:\Datalysys\Prasad\BOTO3>
cript.py
UserName : Practise
UserName : User1
```

➔ Next, we will be working with ec2 instances, where we will be describing each and every ec2 instance in our console. We will be using describe_instances() function to do this.

```
EC2_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='ec2', region_name='us-east-1')
EC2Response = EC2_CONSOLE_AWS.describe_instances()  #response json for ec2 instances description
```

➔ Since this is a longer json, we need to iterate inside the key-list pairs and find our required operation, we are printing the instance IDs so we need to iterate through – “Reservations -> Instances -> InstanceID” to get our required result.

```
EC2_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='ec2', region_name='us-east-1')
EC2Response = EC2_CONSOLE_AWS.describe_instances()  #response json for ec2 instances description
for sub_list in EC2Response['Reservations']:
    for each_instance in sub_list['Instances']:
        print("Instance ID: "+each_instance['InstanceId'])
```

➔ We get the following output:

```
Instance ID: i-0c3161e49064246a9
```

➔ Next, we will be printing all the buckets in the s3 service using client object. First we need to have the Response from S3 client object and Iterate over “Buckets” to get the bucket name like this:

```
S3_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='s3', region_name='us-east-1')
S3Response = S3_CONSOLE_AWS.list_buckets()
for each_bucket in S3Response['Buckets']:
    print("Bucket Name : "+each_bucket['Name'])
```

➔ We will get the names of buckets like this:

```
Bucket Name : cf-templates-4u351hjm9psg-us-east-1
```

**Implement Boto3 scripts using Resource Object:**

➔ Resource Objects are available only for selected services in AWS and needs to be handled based on the availability.

➔ The resource actions are classified as three sets for “Resource” objects:

a. First set to create/modify the existing objects for an AWS service.
   Eg: Create user, Create policy, Change password etc.
b. Second set to work on the particular existing objects in the AWS Service.
c. Third set to see the existing objects as a collection/group.

➔ Suppose, we need to print all the IAM users present in our AWS Console, we can write the code like this: (taking the custom session of root profile)

```
IAM_CONSOLE_AWS = AWS_MGMT_CONSOLE.resource(service_name='iam', region_name='us-east-1')
print("The IAM Users present in the console are :")
for each_user in IAM_CONSOLE_AWS.users.all():
    print(each_user.name)
```

➔ We get output like this:

```
The IAM Users present in the console are :
Practise
User1
```

➔ Suppose, we want to apply the limit for getting the results, we can do like this:

```
print("Applying limit on results:")
for each_user in IAM_CONSOLE_AWS.users.limit(1):
    print(each_user.name)
```

➔ We get only one user if we apply the above limit.
➔ Now we will be printing the Instance ID of all the instances present in our console like this:

```
EC2_CONSOLE_AWS = AWS_MGMT_CONSOLE.resource(service_name='ec2', region_name='us-east-1')
print("Fetching all the EC2 instances ID from the console")
for each_instance in EC2_CONSOLE_AWS.instances.all():
    print(each_instance.id)
```

➔ We get the ID of the instance, if we want to print the name of the instance then we need to make sure that instance has a tag name, so that we get the name of the instance by the tagname.

➔ Now we will be printing the bucket names in the s3 service by this code:

```
S3_CONSOLE_AWS = AWS_MGMT_CONSOLE.resource(service_name='s3', region_name='us-east-1')
for each_bucket in S3_CONSOLE_AWS.buckets.all():
    print("Bucket Name : "+each_bucket.name)     #printing every buckets from s3
```

➔ We will be getting all the bucket names in the output.

**Boto3 script to get our AWS AccountID:**

➔ We will be getting our AWS Account ID from STS (Security Token Service) with the help of the client object.
➔ Start with the custom session with root profile name.
➔ Next call the sts client object with service name as “sts” and region name as “us-east-1”
➔ We will be using the **get_caller_identity()** method to get the username, accountID etc from sts service.
➔ We will get the dictionary response containing the username, accountID, ARN and other fields.

- As we are using the root profile, the username and the accountID must be same, we get different username for different IAM users.
- Now we need to access the dictionary response's ACCOUNT key to get the required account ID.
- If we want to print the username, we need to access the ARN key, in the end we get the name with user/<userName>
- The final function script looks like this:

```
def GetAccountIDUsingSTSClientObject():
    try:
        STS_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='sts', region_name='us-east-1')  #Opening the STS service
        stsResponse = STS_CONSOLE_AWS.get_caller_identity()
        print("Account ID is : "+stsResponse['Account'])
        print("User name inside the ARN is : "+stsResponse['Arn'])

    except Exception as e: print("Error getting the Account ID Using STS using Client object {}".format(e))
```

- The output will be like this:

```
Account ID is : 462744317624
User name inside the ARN is : arn:aws:iam::462744317624:root
```

**BOTO3 scripts on EC2 Services :**

- We will be working on the simple EC2 services here with client object.
- First let's list out all the EC2 instances ID, state and when they were loaded in the console.
- We will be starting the custom session on root profile.
- We will make use of the **describe_instances()** function to work with this activity.
- First lets print the ImageID, InstanceID, loading Date and Instance State of an EC2 instance, check out the **"WorkingOnEC2ServicesUsingClientObject()"** function that has the implementation for it.
- The output will be:

```
Instance ID: i-0c3161e49064246a9
Instance Image ID: ami-0323c3dd2da7fb37d
Instance Launch Time: 2020-05-10
Instance State is: stopped
```

- Next we will be working on volumes of the EC2 instances using the **describe_volumes()** function.
- We will be printing the Volume ID, Availability Zone, Volume Type in the terminal. Refer the same function "**WorkingOnEc2ServicesUsingClientObject**()".
- We will be getting the output as:

```
The Volume ID is: vol-0b8e338014d6d8921
Availability Zone: us-east-1c
Volume Type: gp2
```

- ➔ Next we will be printing all the security groups present in our ec2 service by using the **describe_security_groups()** function.
- ➔ We will get the output like this:

```
The Security groups present are:
secure-group
webdmz
default
```

**Menu Driven Script to perform different actions on EC2 instances:**

- ➔ We will be starting, stopping, terminating and exiting the menu driven based script.
- ➔ User needs to provide one of the options and the specific action will be performed.
- ➔ Refer to the script named "MenuDrivenScript_EC2.py" for complete code.
- ➔ For Resource, we need to pass the instance ID and call the start(), stop(), terminate() methods for that ID.
- ➔ For Client, we have functions namely start_resource(), stop_resource(), terminate_resource() and pass the Instance ID as the parameter.
- ➔ If we have multiple Instance ID to be performed these operations, then pass them as a list of strings to client object. Like this: [**'InstanceID1','InstanceID2','InstanceID3']**

**BOTO3 Waiters:**

- ➔ This is a way to block something, until a certain state has been reached.
- ➔ Consider that we are starting an EC2-instance using boto3 script and we are printing the status on the terminal as "Instance is up and running", but there will be a time gap between moving the instance from stopped state to running state.
- ➔ The Sample script is listed below:

```python
import boto3

#Accessing AWS Console using the Root profile
AWS_CONSOLE = boto3.session.Session(profile_name='root')

#We are taking both EC2 Resource and Client objects here
EC2_RESOURCE = AWS_CONSOLE.resource(service_name='ec2',region_name='us-east-1')
EC2_CLIENT = AWS_CONSOLE.client(service_name='ec2',region_name='us-east-1')

INSTANCE_OBJECT = EC2_RESOURCE.Instance('i-0c3161e49064246a9')      #Working on BOTO3 instance by its ID

print("Starting the Instance....")
INSTANCE_OBJECT.start()     #Function to start the instance
print("Instance is up and running...")
```

➔ You can see that we get both the print statements output in console, but instance is still in pending state.

| Name | Instance ID | Instance Type | Availability Zone | Instance State | Status Checks |
|---|---|---|---|---|---|
| Boto3 | i-0c3161e49064246a9 | t2.micro | us-east-1c | pending | Initializing |

➔ Lets add our own logic to wait until the state changes from pending to running and then print the "up and running" print statement.

➔ We can fetch the instance current state by using **.state** method like this:

```
print(INSTANCE_OBJECT.state['Name'])
```

➔ We will be writing a loop to check the status of the instance and then print the required results on the screen like this:

```
while True:     #Simple Waiter loop
    INSTANCE_OBJECT = EC2_RESOURCE.Instance('i-0c3161e49064246a9')          #Working on BOTO3 instance by ID
    print("Current Instance State: {}".format(INSTANCE_OBJECT.state['Name']))       #Fetching current state
    if INSTANCE_OBJECT.state['Name']=='running':        #If state=running, exit the loop
        break
    else:                   #Else, wait for the status to be changed back to running state
        print("Waiting for the status to be changed to Running state..")
        time.sleep(5)       #Wait for next 5 seconds
```

➔ After the status of the instance is changed to running state, we get the final output. Our code has waited for the instance to be changed to running state.

Our sample output would be like this:

```
Starting the Instance....
Current Instance State: pending
Waiting for the status to be changed to Running state..
Current Instance State: pending
Waiting for the status to be changed to Running state..
Current Instance State: pending
Waiting for the status to be changed to Running state..
Current Instance State: pending
Waiting for the status to be changed to Running state..
Current Instance State: running
Instance is up and running...
```

➔ Otherwise, we can use the predefined function inside the BOTO3 to wait until the state gets changed to running state: the predefined function is: **wait_until_running()**

➔ The sample script would be like this:

```
def waiterFunctionUsingBOTO3():
    try:
        #We are taking both EC2 Resource and Client objects here
        EC2_RESOURCE = AWS_CONSOLE.resource(service_name='ec2',region_name='us-east-1')
        EC2_CLIENT = AWS_CONSOLE.client(service_name='ec2',region_name='us-east-1')

        INSTANCE_OBJECT = EC2_RESOURCE.Instance('i-0c3161e49064246a9')      #Working on BOTO3 instance by its ID

        print("Starting the Instance....")
        INSTANCE_OBJECT.start()     #Function to start the instance
        INSTANCE_OBJECT.wait_until_running()       #predefined function to wait until state changes to running state
        print("Instance is up and running...")          #Now instance is started..
    except Exception as e: print("Issue starting the instance : {}".format(e))
```

➔ Now by default, the running state will be checked and second print statement will be outputted automatically once the instance changes to running state.

➔ **Important point:** ***"Resource waiter waits for 200 seconds and checks for the running state, if this state is not attained inside the 200 seconds then we get an exception" (40 checks for every 5 seconds).***

➔ Now we will be writing the same kind of logic using the client object with the help of the boto3 docs.

➔ The sample script would be:

```
def waiterFunctionUsingClientWaiters():
    try:
        #We are taking only EC2 Resource objects here
        EC2_CLIENT = AWS_CONSOLE.client(service_name='ec2',region_name='us-east-1') #Client Object
        print("Starting the Instance....")
        EC2_CLIENT.start_instances(InstanceIds=['i-0c3161e49064246a9'])     #Starting EC2 instance using ID
        waiter = EC2_CLIENT.get_waiter('instance_running')          #Creating the waiter to wait for running status
        waiter.wait(InstanceIds=['i-0c3161e49064246a9'])        #Waiter object with instance ID
        print("Instance is up and running...")          #Now instance is started..
    except Exception as e: print("Issue starting the instance : {}".format(e))
```

➔ We can attach the client waiters to the resource objects also, it will work. Refer the sample script code for this function.

➔ **Important point:** ***"Client waiters wait for 600 seconds and checks for the running state, if this state is not attained inside the 600 seconds then we can get the exception" (40 checks for every 15 seconds).***

➔ It is a good practice to use Client Waiters because they have high waiting time.

**BOTO3 Meta Concept:**

- ➔ Useful to enter into client object from the resource object.
- ➔ Suppose, we are working with an AWS service using the resource object, and we find out that there is no required resource function for that service, suddenly we might want to switch to its client object, then this META concept is useful.
- ➔ Consider this scenario: **"Listing Available Regions for EC2 Service"**
- ➔ We don't have the resource object function for this operation, so we might want to move to client object for this.
- ➔ We follow the following syntax for that:
  **<Resource Object>.meta.client.<Client Object Function>**
- ➔ The sample code would be like this:

```
def MetaObjectExample():
    try:
        EC2_CONSOLE_RESOURCE = AWS_MGMT_CONSOLE.resource(service_name='ec2')
        for each_region in EC2_CONSOLE_RESOURCE.meta.client.describe_regions()['Regions']:  #Iterating over Regions Key
            print("{}".format(each_region['RegionName']))       #We get Regions under 'RegionName' key
    except Exception as e: print("Error Performing the Meta operation : {}".format(e))
```

- ➔ The sample output will be like this:

```
eu-north-1
ap-south-1
eu-west-3
eu-west-2
eu-west-1
ap-northeast-2
ap-northeast-1
sa-east-1
ca-central-1
ap-southeast-1
ap-southeast-2
eu-central-1
us-east-1
us-east-2
us-west-1
us-west-2
```

**BOTO3 Collections Concept:**

- ➔ This is useful for collecting the information about the AWS Service as a group.
- ➔ First, we will be dealing with Collections concept in **Resource** group.
- ➔ We can collect the information as a group with use of EC2's Snapshots, Volumes, AMIs etc.
- ➔ When we refer the BOTO3 documentations, we can get the clear idea of what are all the services we can use as a collection under COLLECTIONS section of an AWS Service.
- ➔ Suppose, we want to work on the instances. Click on **Instances** under collections.
- ➔ Primarily, we would be looking at **ALL, LIMIT, FILTER** functions under collections.

- ➔ **ALL** property is useful for listing all the instances present in our EC2 dashboard for specified region.
- ➔ **LIMIT** property is used to limit the number of instances that are outputted on terminal, suppose we want only top 10 instances then we can use LIMIT (10)
- ➔ **FILTER** is used to filter the results based on the condition.
- ➔ Suppose we want to filter the instances based on the running instances only, then we need to pass the first filter like this:

***Filter1= {"Name":"instance-state-name", "Values":["running"]}***

- ➔ The sample code written would be like this:

```
def ResourceCollectionsExample():
    try:
        EC2_CONSOLE_RESOURCE = AWS_MGMT_CONSOLE.resource(service_name='ec2')
        for each_instance in EC2_CONSOLE_RESOURCE.instances.all():      #All Property for getting all instances
            print(each_instance.id)         #Printing the ID of the instances

        for each_limitted_instances in EC2_CONSOLE_RESOURCE.instances.limit(10):    #Limiting to specified number
            print(each_limitted_instances.id)       #Displaying only top 10 instance ID

        FirstFilter={"Name":"instance-state-name","Values":['running','stopped']}     #Filter to get only running and sto
        for each_filtered_instances in EC2_CONSOLE_RESOURCE.instances.filter(Filters=[FirstFilter]):   #Passing List of f
            print(each_filtered_instances.id)       #Instance ID Filtered with only running and stopped status
    except Exception as e: print("Issue working on collections : {}".format(e))
```

- ➔ We can also add the filter to fetch only t2.micro instances only by this format:

***Filter2= {"Name":"instance-type", "Values":["t2.micro"]}***

**BOTO3 Collections concept to start,stop,terminate required Instances at once:**

- ➔ We will use the start(), stop(), terminate() function to start, stop and terminate the instances at once.
- ➔ We will keep the waiter functions also to make sure that we get proper messages after the action is completed.
- ➔ Before applying the waiters, we need to collect our entire EC2 instance IDs.
- ➔ We will call the ALL method and store all the instance IDs in a list.
- ➔ Since we are applying the waiters using client waiters, we need to have the list of Instance IDs.
- ➔ The sample script for running all the instances with client waiters will look like this:

```python
def CollectionsConceptForEC2():
    try:
        EC2_CONSOLE_RESOURCE = AWS_MGMT_CONSOLE.resource(service_name='ec2',region_name='us-east-1')    #resource object
        EC2_CONSOLE_CLIENT = AWS_MGMT_CONSOLE.client(service_name='ec2',region_name='us-east-1')    #client object

        AllInstanceIDS=[]       #List to store all the instance ID

        for each_id in EC2_CONSOLE_RESOURCE.instances.all():        #Collecting all instance ID
            AllInstanceIDS.append(each_id.id)               #appending to list

        waiter = EC2_CONSOLE_CLIENT.get_waiter('instance_running')      #Waiter object for running instances

        print("Starting all the instances....")
        EC2_CONSOLE_RESOURCE.instances.start()          #Starting the instances
        waiter.wait(InstanceIds=AllInstanceIDS)     #Wait for all instances to move from pending to running state
        print("All instances are moved to running state...")

    except Exception as e: print("Issue occured while performing operations on EC2 : {}".format(e))
```

- ➔ Suppose, we need to start only selective servers with using the function, but unfortunately we don't have the selective start() for resource object. So we will be using the client object for this.
- ➔ Provide some tags for the instances, for example we are providing three instances with tags – 1 with PROD and 2 with NON_PROD, now we need to start only NON_PROD instances.
- ➔ First, we will filter the instances based on the tagname like this:

***Filter = {'Name':'tag:Name','Values':['NON_PROD']}***

- ➔ The sample script for filtering the Tags for the instances using resource and client object is like this:

```python
def FilterBasedOnTagNames():
    try:
        EC2_CONSOLE_RESOURCE = AWS_MGMT_CONSOLE.resource(service_name='ec2',region_name='us-east-1')    #resource object
        EC2_CONSOLE_CLIENT = AWS_MGMT_CONSOLE.client(service_name='ec2',region_name='us-east-1')    #client object

        ResourceTagID=[]    #List to store the filtered instance ID
        ClientTagID=[]      #List of client object filtered Instances ID

        TagFilter = {'Name':'tag:Name','Values':['NON_PROD']}       #filtering non_prod servers

        for each_id in EC2_CONSOLE_RESOURCE.instances.filter(Filters=[TagFilter]):  #Iterating over the filter
            ResourceTagID.append(each_id.id)        #appending the ID to TagID List

        for each_item in EC2_CONSOLE_CLIENT.describe_instances(Filters=[TagFilter])['Reservations']:    #Iterate over Rese
            for each_instances in each_item['Instances']:       #Iterating over Instances Key
                ClientTagID.append(each_instances['InstanceId'])       #InstanceID key has the instances ID

    except Exception as e: print("Unable to filter the instances based on tagnames : {}".format(e))
```

- ➔ Now we have implemented the script to start only the NON-prod instances in our Ec2 dashboard using the client objects like this:

```
def StartInstancesBasedOnIDWithWaiter():
    try:
        EC2_CONSOLE_CLIENT = AWS_MGMT_CONSOLE.client(service_name='ec2',region_name='us-east-1')    #client object
        ClientTagID=[]      #List of client object filtered Instances ID

        TagFilter = {'Name':'tag:Name','Values':['NON_PROD']}       #filtering non_prod servers

        for each_item in EC2_CONSOLE_CLIENT.describe_instances(Filters=[TagFilter])['Reservations']:    #Iterate over Rese
            for each_instances in each_item['Instances']:       #Iterating over Instances Key
                ClientTagID.append(each_instances['InstanceId'])       #InstanceID key has the instances ID

        print("Starting the Non-prod instances:....")
        EC2_CONSOLE_CLIENT.start_instances(InstanceIds=ClientTagID)     #based on the tag IDs
        waiter = EC2_CONSOLE_CLIENT.get_waiter('instance_running')      #Waiter object for client
        waiter.wait(InstanceIds=ClientTagID)                        #Wait until the instances are moved to running state

        print("Your Non prod instances are now up and running...")

    except Exception as e: print("Unable to start instances : {}".format(e))
```

➔ Similar way, we can stop the instances by replacing the **start_instances()** function with **stop_instances()** and terminate the instances by replacing the **start_instances()** function with **terminate_instances()** function.

**Delete Unused and Untagged EBS Volumes:**

➔ Make sure to create an EC2 instance object because the volumes are present under EC2 instances.
➔ We will be working on the EC2 Resource object.
➔ With the help of **EC2_ResourceObject.volumes.all()** function, we will be printing the Volume ID and Volume state like this:

```
def CleanUpVolumes():
    try:
        EC2_CONSOLE_RESOURCE = AWS_MGMT_CONSOLE.resource(service_name='ec2',region_name='us-east-1')    #resource object
        for eachVolume in EC2_CONSOLE_RESOURCE.volumes.all():
            print("Volume ID : {}\tVolume State: {}".format(eachVolume.id,eachVolume.state))
    except Exception as e: print("Issue cleaning up the volumes: {}".format(e))
```

➔ The output will be like this:

```
Volume ID : vol-0b8e338014d6d8921       Volume State: in-use
```

➔ Suppose we want to print only the available state volumes, then we will be applying the filter based on the status of the volume like this:

```
def CleanUpVolumes():
    try:
        EC2_CONSOLE_RESOURCE = AWS_MGMT_CONSOLE.resource(service_name='ec2',region_name='us-east-1')    #resource object
        StateFilter={'Name':'status','Values':['available']}
        for eachVolume in EC2_CONSOLE_RESOURCE.volumes.filter(Filters=[StateFilter]):
            print("Volume ID : {}\tVolume State: {}".format(eachVolume.id,eachVolume.state))
    except Exception as e: print("Issue cleaning up the volumes: {}".format(e))
```

➔ Some of the volumes are provided with some tag names, we need to find out those volumes are not going to delete those volumes. We will be also printing the tags associated to each volume in our output by using the **volume.tags** property

```
def CleanUpVolumes():
    try:
        EC2_CONSOLE_RESOURCE = AWS_MGMT_CONSOLE.resource(service_name='ec2',region_name='us-east-1')    #resource object
        # StateFilter={'Name':'status','Values':['available']}
        # Filters=[StateFilter]
        for eachVolume in EC2_CONSOLE_RESOURCE.volumes.all():
            print("Volume ID : {}\tVolume State: {}\tVolume Tag: {}".format(eachVolume.id,eachVolume.state,eachVolume.tags
    except Exception as e: print("Issue cleaning up the volumes: {}".format(e))
```

➔ The output will be like this:

```
Volume ID : vol-0b8e338014d6d8921       Volume State: in-use    Volume Tag: None
```

➔ We would be using the same property to print all the volumes that are having no tags by using a if statement like this: **"if not eachVolume.tags:"**

```
def CleanUpVolumes():
    try:
        EC2_CONSOLE_RESOURCE = AWS_MGMT_CONSOLE.resource(service_name='ec2',region_name='us-east-1')    #resource object
        # StateFilter={'Name':'status','Values':['available']}
        # Filters=[StateFilter]
        for eachVolume in EC2_CONSOLE_RESOURCE.volumes.all():   #To get all the volumes
            if not eachVolume.tags:     #To get only the volumes without tags
                print("Volume ID : {}\tVolume State: {}\tVolume Tag: {}".format(eachVolume.id,eachVolume.state,eachVolume.
    except Exception as e: print("Issue cleaning up the volumes: {}".format(e))
```

➔ Now we will be deleting the available and untagged volumes from our AWS Console by using this:

```
def CleanUpVolumes():
    try:
        EC2_CONSOLE_RESOURCE = AWS_MGMT_CONSOLE.resource(service_name='ec2',region_name='us-east-1')    #resource object
        StateFilter={'Name':'status','Values':['available']}
        for eachVolume in EC2_CONSOLE_RESOURCE.volumes.filter(Filters=[StateFilter]):   #To get all the volumes
            if not eachVolume.tags:     #To get only the volumes without tags
                print("Unused and Untagged Volumes are : ")
                print("Volume ID : {}\tVolume State: {}\tVolume Tag: {}".format(eachVolume.id,eachVolume.state,eachVolume.
                print("Deleting the unused and untagged volumes:")
                eachVolume.delete()     #This function is used to delete the volumes
    except Exception as e: print("Issue cleaning up the volumes: {}".format(e))
```

➔ Till now we implemented the code using the Resource object, now we will be using the Client Object.

➔ Firstly, we need to print only the volumes that are untagged, so we will be writing this code:

```
def CleanUpVolumesByClientObject():
    try:
        EC2_CONSOLE_CLIENT = AWS_MGMT_CONSOLE.client(service_name='ec2',region_name='us-east-1')    #client object
        for eachVolume in EC2_CONSOLE_CLIENT.describe_volumes()['Volumes']: #Iterating over the Volumes Key
            if not 'Tags' in eachVolume:            #If there is no Key named Tags, then we will be printing the volumes
                print(eachVolume['VolumeId'])
    except Exception as e: print("Issue cleaning up the volumes: {}".format(e))
```

- ➔ Now we will be searching for only volumes that are having available state.
- ➔ Now we will be deleting each volume by its volume ID.
- ➔ The final code would look like this:

```
def CleanUpVolumesByClientObject():
    try:
        EC2_CONSOLE_CLIENT = AWS_MGMT_CONSOLE.client(service_name='ec2',region_name='us-east-1')    #client object
        print("Deleting the Unused and untagged volumes..")
        for eachVolume in EC2_CONSOLE_CLIENT.describe_volumes()['Volumes']: #Iterating over the Volumes Key
            if not 'Tags' in eachVolume and eachVolume['State']=='available':  #If there is no Key named Tags, then we wi
                print("{} is deleted".format(eachVolume['VolumeId']))
                EC2_CONSOLE_CLIENT.delete_volume(VolumeId=eachVolume['VolumeId'])
            else:
                print("There are no untagged and unused volumes")
    except Exception as e: print("Issue cleaning up the volumes: {}".format(e))
```

**EC2 Instances Inventory Script:**

- ➔ We will be writing a boto3 script to discover the EC2 instances in our Console, and then outputting the discovery results to a csv.
- ➔ The CSV Contains the SerialNumber, Instance Name, Instance Type, Architecture, Created time, Private IP in it.
- ➔ Refer the **EC2InventoryCSV.py** file to get the complete script.
- ➔ We will not be able to list out the volumes here, so instead try out this script with client object to get extra columns.
- ➔ Now we will be able to extract Instance, Volumes, KeyPair related data easily with the use of Client Object.

**List all Snapshots:**

- ➔ Here we will be listing all the snapshots for our Region in AWS Console.
- ➔ We will be using the resource object to call the snapshots property and then filter the ID based on our AWS Account ID, because we want to list only our snaps rather than default snaps present in AWS.
- ➔ We will be using the STS service and getting the Account ID and passing it to Filter the snaps based on our AWS Account ID.

```
def ListSnapshotsByResourceObject():
    try:
        EC2_CONSOLE_RESOURCE = AWS_MGMT_CONSOLE.resource(service_name='ec2',region_name='us-east-1')    #resource object
        STS_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='sts', region_name='us-east-1')  #Opening the STS service
        stsResponse = STS_CONSOLE_AWS.get_caller_identity()
        OwnerID = stsResponse['Account']        #Getting our Account ID
        for snap in EC2_CONSOLE_RESOURCE.snapshots.filter(OwnerIds=[OwnerID]):  #Getting snaps based on AccountID
            print(snap)     #printing all the snaps
    except Exception as e: print("Issue listing the snapshots: {}".format(e))
```

- ➔ Based on the Client Object, we have this script, here we need to convert the AccountID as List and send for the filter:

```
def ListSnapshotsByClientObject():
    try:
        OwnerIDList=[]
        EC2_CONSOLE_CLIENT = AWS_MGMT_CONSOLE.client(service_name='ec2',region_name='us-east-1')    #client object
        STS_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='sts', region_name='us-east-1')  #Opening the STS service
        stsResponse = STS_CONSOLE_AWS.get_caller_identity()
        OwnerIDList.append(stsResponse['Account'])       #Getting our Account ID
        for snap in EC2_CONSOLE_CLIENT.describe_snapshots(OwnerIds=OwnerIDList)['Snapshots']:   #based on our AccountID
            print(snap['SnapshotId'])       #Printing the SnapID
    except Exception as e: print("Issue listing the snapshots: {}".format(e))
```

**List Owned Snapshots based on Size:**

- ➔ Consider we have three snapshots in our console with two of them having 8 GB of space and another one with 10 GB of space.
- ➔ We need to list out the snapshots having the 8GB of space in our console.
- ➔ We will do it by implementing a script like this:

```
def ListSnapshotsBasedOnSizeUsingClientObject():
    try:
        OwnerIDList=[]
        SizeFilterList=[]
        EC2_CONSOLE_CLIENT = AWS_MGMT_CONSOLE.client(service_name='ec2',region_name='us-east-1')    #client object
        STS_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='sts', region_name='us-east-1')  #Opening the STS service
        stsResponse = STS_CONSOLE_AWS.get_caller_identity()
        OwnerIDList.append(stsResponse['Account'])       #Getting our Account ID
        SizeFilter={'Name':'volume-size','Values':['8']}          #Specifying size in GB
        SizeFilterList.append(SizeFilter)   #We need to pass filters as list, so appending filter to a list
        for snap in EC2_CONSOLE_CLIENT.describe_snapshots(OwnerIds=OwnerIDList,Filters=SizeFilterList)['Snapshots']:   #ba
            print(snap['SnapshotId'])       #Printing the SnapID
    except Exception as e: print("Issue listing the snapshots: {}".format(e))
```

**List Snapshots based on the start-time:**

- ➔ We will be using the start_time() property to print the starting time of the snapshot.
- ➔ Consider that we have a snapshot created at particular time and we want to list the snap, for that we need to have a datetime object with date,time specified in it.
- ➔ Now we will be matching the start-time condition with the snap start-time and printing all the snapshots that match the condition by using the script like this:

```
def ListSnapshotsBasedOnStartTimeUsingClientObject():
    try:
        OwnerIDList=[]
        EC2_CONSOLE_CLIENT = AWS_MGMT_CONSOLE.client(service_name='ec2',region_name='us-east-1')    #client object
        STS_CONSOLE_AWS = AWS_MGMT_CONSOLE.client(service_name='sts', region_name='us-east-1')  #Opening the STS service
        stsResponse = STS_CONSOLE_AWS.get_caller_identity()
        OwnerIDList.append(stsResponse['Account'])       #Getting our Account ID
        today = datetime.now()
        StartTime = str(datetime(today.year,today.month,today.day,10,10,21)) #Time matching condition

        for snap in EC2_CONSOLE_CLIENT.describe_snapshots(OwnerIds=OwnerIDList)['Snapshots']:   #based on our AccountID
            if snap['StartTime'].strftime("%Y-%m-%d %H:%M:%S")==StartTime:  #Check if snap time matches condition
                print(snap['SnapshotId'],snap['StartTime'].strftime("%Y-%m-%d %H:%M:%S"))       #Printing the SnapID, Snap
            else:   #Otherwise
                print("No snapshots created for the time : {}".format(StartTime))
    except Exception as e: print("Issue listing the snapshots: {}".format(e))
```

**Creating EC2 instances by using CRON Job for particular time interval:**

- Here we will be using the python-crontab module to create a CRON job to run for every 5 mins and create new ec2 instances for every 5 mins.
- Firstly, install python-crontab by using **pip install python-crontab**.
- In windows, we have command named "SCHTASKS", we will be using it to create the tasks automatically for every 5 mins.
- First create the bat file by entering the following in the bat file:
  1. Directory in which the task should run.

     Eg: We need to log every output of script to a log file, our file is located at: **"C:\Datalysys\Prasad\BOTO3\logs"** so the first line would be like this:

     ***cd "C:/Datalysys/Prasad/BOTO3/"***

  2. Next, we need to specify the python.exe path and the path of the script. So our second line in bat file would be like this:

     ***"C:\Windows\py.exe" "C:/Datalysys/Prasad/BOTO3/CreateEC2Instances.py"***

     We can also add the arguments if required, like this:

     ***"C:\Windows\py.exe" "C:/Datalysys/Prasad/BOTO3/CreateEC2Instances.py" "1000"***

  3. We need to pause after the script is executed, so we will be adding the pause command at the end
- The complete lines of the batch file would be like this:

```
cd "C:/Datalysys/Prasad/BOTO3/"
"C:\Windows\py.exe" "C:/Datalysys/Prasad/BOTO3/CreateEC2Instances.py" "1000"
pause
```

- We need to write the command to create the schedule at the given time and interval like this in our BOTO3 directory command line opened as admin:

***"schtasks.exe /Create /tn "MyFirstTask" /SC ONCE /TR***
***"C:\Datalysys\Prasad\BOTO3\runscript.bat" /st 19:28"***

Where **/create** is used to create the task, **/SC** is used for interval, **/TR** is the path of the bat file **/ST** is used for the time to start the script

- We can also run the schedule manually from our end by passing the Taskname like this:

***"schtasks /run /tn MyFirstTask"***

- We can list the schedule and know its information by using this command:

***"schtasks /query /fo LIST /tn Calculator"***

- Now we need to start writing our main python script for creating instances.

➔ Our sample script would be like this:

```
response = Ec2Client.run_instances(
BlockDeviceMappings=[
        {
            'DeviceName': '/dev/xvda',
            'Ebs': {

                'DeleteOnTermination': True,
                'VolumeSize': 8,
                'VolumeType': 'gp2'
            },
        },
    ],
    ImageId='ami-09d95fab7fff3776c',
    InstanceType='t2.micro',
    MaxCount=1,
    MinCount=1,
    Monitoring={
        'Enabled': False
    },
    SecurityGroupIds=[
        'secure-group',
    ],
)
```

➔ Refer the "**CreateEC2Instances.py**" for detailed script.

**Running BOTO3 scripts from an EC2 instance that has an IAM Role:**

➔ First, we will be removing the programmatic access keys assigned in **.aws** directory in our system by typing "**rm -rf .aws/**" in the instance that is running.

➔ After that go back to IAM section in the AWS Console and click on Create-Role and select the exisiting role based on the requirement such as **EC2 Full access** or else you can also select the **FullAdministrator** role if you want to perform all the actions from this instance.

➔ Now after creating the role, go to back to EC2 and select Instance **settings > Attach / Replace IAM Role.**

➔ Now select the newly created role from IAM.

➔ Now we no need to use the programmatic access keys, we can use the default session and then run our scripts from that EC2 instance with attached role.

➔ Now we can create resource or client object like this:
***"EC2_CONSOLE_RESOURCE = boto3.resource(service_name='ec2',region_name='us-east-1')"***

➔ Going forward, we will be using this role-based login for performing BOTO3 scripts it will be useful when we are executing the scripts using **AWS LAMBDA.**

**BOTO3 Exception Handling:**

- First and most common errors while executing python BOTO3 scripts would be **Syntax errors.**
- If we miss the predefined rule or way of writing the program, we will encounter the syntax errors.
- Eg: 1. "**imporx sys"** instead of **"import sys"**

  2. **"print("Hello)"** instead of **"print("Hello")"**

- Next, we can encounter the **Runtime Errors** that will occur if any of the module is not installed in our machine and we are trying to import that module in our program. We can handle that by using **try:** and **except:**

- In BOTO3 also we have some exceptions class to handle the exceptions, we can list them out by typing **dir(boto3.exceptions)**
- Suppose, we are trying to use the profile name that is not listed in our **.aws** directory, then we will get the **botocore.ProfileNotFound** Exception.
- We can handle this exception by passing the Exception line to **except:** like this

  ***except botocore.exceptions.ProfileNotFound:***

  followed by the lines that are responsible for handling this exception.

- We need to configure **botocore** module also for handling botocore exceptions.
- Suppose, we are using the S3-Developer role to work on the EC2 operations, we get the Access Error, but not in a friendly manner, we can extract the response in our way like this:

```
Aws_management_console = boto3.session.Session(profile_name='s3-developer')
try:
        iam_console_re = aws_management_console.resource(service_name='iam')
        for each_user in iam_console_re.users.all():
                print(each_user)
except botocore.exceptions.ClientError as e:
        print(e.response)
```

- For every exception we get, it contains a response json, we will be looking after that json and printing the exception in our way like this:

```
        Print(e.response['Error']['Code'])
```

- This will iterate through Error key and Code Key to produce the meaningful response.
- We can capture this response and produce meaningful messages by using the if statements also like this:

```
if e.response['Error']['Code']=='AccessDenied':
        print("Your Profile is not having necessary permissions")
```

**Getting all IAM User Details:**

- ➔ We need to create the IAM User object by calling the **get_user()** function using the Client Object.
- ➔ We will be preferring the Client Object because, we can get the detailed results from Client Object and it is applicable to all the services in the AWS.
- ➔ We will be printing the user's name, ID, ARN, Creation Date using **get_user()** like this:

```
def GetUserDetailsFromIAM():
    try:
        IAM_CLIENT = AWS_MGMT_CONSOLE.client(service_name='iam')    #client object, as IAM is global so no region
        clientObject = IAM_CLIENT.get_user(UserName='Practise')['User']    #getting details of 'Practise' user
        print(clientObject['UserName'],clientObject['UserId'],clientObject['Arn'],clientObject['CreateDate'].strftime("%Y-
    except Exception as e: print("Issue listing the user's details : {}".format(e))
```

- ➔ We will be parsing the date we get as **%Y-%m-%d** for **CreateDate**.
- ➔ The output would be like this:

```
Practise AIDAWXPNV524CYEG5XFGP arn:aws:iam::462744317624:user/Practise 2020-05-10
```

- ➔ Suppose we want to list all the users details then we would be using the **list_users()** function.
- ➔ The sample program for listing all the users details in the same manner as above would be:

```
def GetAllUsersDetailsFromIAM():
    try:
        IAM_CLIENT = AWS_MGMT_CONSOLE.client(service_name='iam')    #client object, as IAM is global so no region
        for eachUser in IAM_CLIENT.list_users()['Users']:    #getting details of all users in IAM
            print(eachUser['UserName'],eachUser['UserId'],eachUser['Arn'],eachUser['CreateDate'].strftime("%Y-%m-%d"))
    except Exception as e: print("Issue listing the user's details : {}".format(e))
```

- ➔ The output would be like this:

```
Practise AIDAWXPNV524CYEG5XFGP arn:aws:iam::462744317624:user/Practise 2020-05-10
User1 AIDAWXPNV524FNKSN45VO arn:aws:iam::462744317624:user/User1 2019-08-24
```

**BOTO3 Script to get information about all the groups present in IAM:**

- ➔ We will be listing the groups information by using the **list_groups()** function of the client Object.
- ➔ The sample program would be like this:

```
def GetGroupInformationFromIAM():
    try:
        IAM_CLIENT = AWS_MGMT_CONSOLE.client(service_name='iam')    #client object, as IAM is global so no region
        for eachGroup in IAM_CLIENT.list_groups()['Groups']:
            print(eachGroup['Arn'],eachGroup['GroupId'],eachGroup['GroupName'],eachGroup['CreateDate'].strftime("%Y-%m-%d'
    except Exception as e: print("Issue listing the group details : {}".format(e))
```

- ➔ We will be parsing the **CreateDate** as **%Y-%m-%d.**
- ➔ The sample output will be like this:

```
arn:aws:iam::462744317624:group/EC2Users AGPAWXPNV524K4MJEXSW4 EC2Users 2020-06-06
arn:aws:iam::462744317624:group/ExampleGroup AGPAWXPNV524DKRD5EMCG ExampleGroup 2020-06-06
```

**Random Password Generator using Python:**

- ➔ We will be generating the password randomly using Python and we would be using this logic going forward for the creation of IAM Users.
- ➔ We would be using the "choice" class inside the random module.
- ➔ The Algorithm for this program would be like this:
    1. ***from random import choice.***
    2. ***Decide the length of the password.***
    3. ***Decide on the valid characters for the password.***
    4. ***Print the choice (valid Characters) to generate the single character for password.***
    5. ***So, we will be appending these characters to a list, so declare a list to store them.***
    6. ***Run a "For Loop" with range (length of the password) and store each choice to list.***
    7. ***Now as we are having list, convert it to string by using join function.***
- ➔ The sample code looks like this:

```
from random import choice           #Importing the choice class from random module

lenOfPassword = 8       #setting the length of the password
validChars = 'abcdefghijklmnopqrstuvwxyz01234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ!@#$%^&*()?'   #allowed chars
passwordList = []       #List to hold the output from choice() class

#function to generate the password with length, validChars, passwordList as arguments
def generatePassword(lenOfPassword, validChars, passwordList):
    try:
        for eachCharacter in range(lenOfPassword):      #iterating using the length of password
            passwordList.append(choice(validChars))     #appending the output of choice class to a list
        finalPassword = "".join(passwordList)           #creating the password string from list
        return finalPassword
    except Exception as e:
        print("Issue generating the password : {}".format(e))
        return -1

def generatePasswordUsingListComprehension(lenOfPassword, validChars):
    try:
        finalPassword = "".join(choice(validChars) for eachCharacter in range(lenOfPassword))
        return finalPassword
    except Exception as e:
        print("Issue generating the password : {}".format(e))
        return -1
```

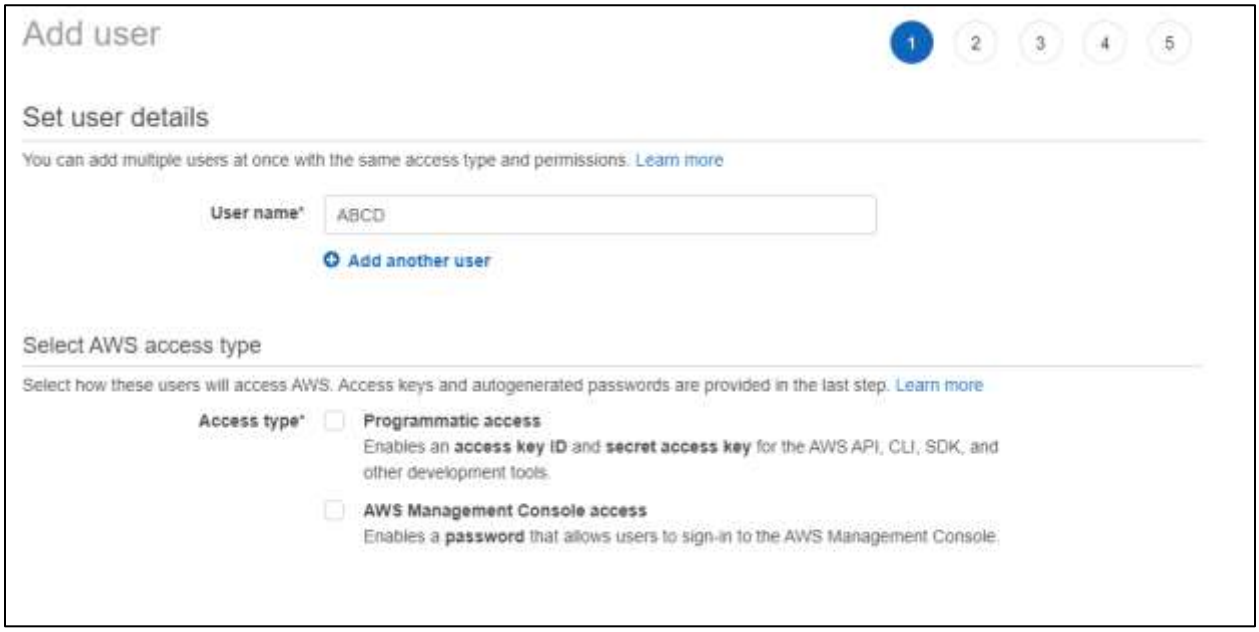
- ➔ We would get randomly 8 character passwords from the output.

**Creating IAM user with Console Login Access using BOTO3:**

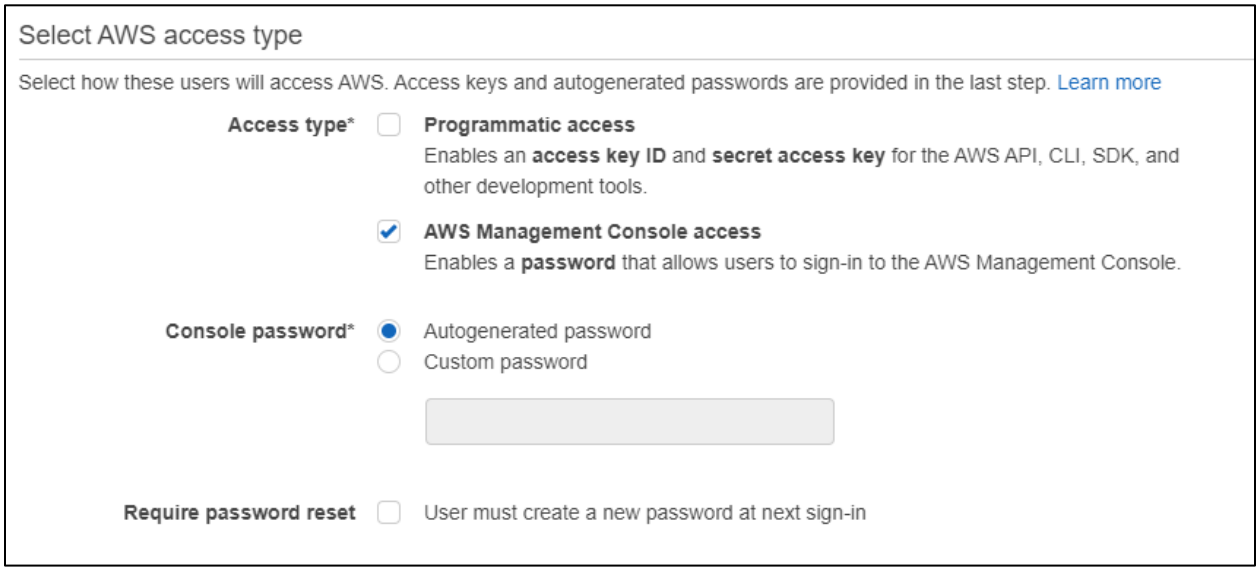
- ➔ Things that must be planned before writing this script:
    a. **Get an IAM User Name.**
    b. **Programmatic Access or Console Access or Both for the user (Permissions for user).**
    c. **Policy that user needs to be attached with (Roles that this user will play).**

➔ Things that we perform while creating the users in AWS Console using IAM service:

1. **Assign a new name to the user.**
2. **Assign the user with Programmatic or Console access or both by checking the checkboxes.**




3. **Creating the Console Password by selecting either "Autogenerated" or "Custom Password".**
4. **We can also select if we want users to reset their password after they login for first time or not.**




➔ We don't have all the required operations as above in the Resource object, so we will be going with Client object to perform the creation of IAM users.

- ➔ The sample script for creating users has been written in **"createIAMUserWithConsoleAccess.py"** please refer the file for complete script.

- ➔ After user is created, the details are stored in the csv file, open the file for temp password and then use the ACCOUNT -ID (12 digits) that can be obtained from IAM console along with IAM user name and temp password to login as IAM user to AWS console.

- ➔ After logging in as IAM user, now you have to reset the password and create a new password.

- ➔ If we run the program again with same username, we will get the error "**User already exists**", we need to handle this error.

- ➔ We have modified the program to add the function to create IAM user with programmatic access keys also.

- ➔ A new function that takes care of both Programmatic and Console Access is listed too in the program.

**Creating Multiple IAM Users By Reading the Inputs from CSV:**

- ➔ We will be providing the User details in a csv and reading the csv from pandas.
- ➔ From the DataFrame, we will be creating the users based on the requirements.
- ➔ The complete script is in "**MultipleIAMUsersWithCSV.py**"

**Creating IAM Inventory such as list of users and their information into a csv:**

- ➔ We will be taking three loops for getting User, Group and Policy attributes respectively.
- ➔ Client object will be used to get all the details here.
- ➔ Refer the script "**IAMInventory.py**" file for the complete script.
- ➔ The output will be stored in a csv file.

**AWS Lambda Functions:**

- ➔ AWS Lambda is a serverless computing platform that allows us to create a small function, configure the function in AWS Console and having the code executed without the need of provisional servers.
- ➔ Assume AWS Lambda as an IDE for writing and executing the codes just as we do in VSCode, PyCharm etc. But it provides additional features compared to normal IDE.
- ➔ Supports languages such as Python, Go, Java, Node JS etc.
- ➔ AWS Lambda comes built it inside the Linux Instances.

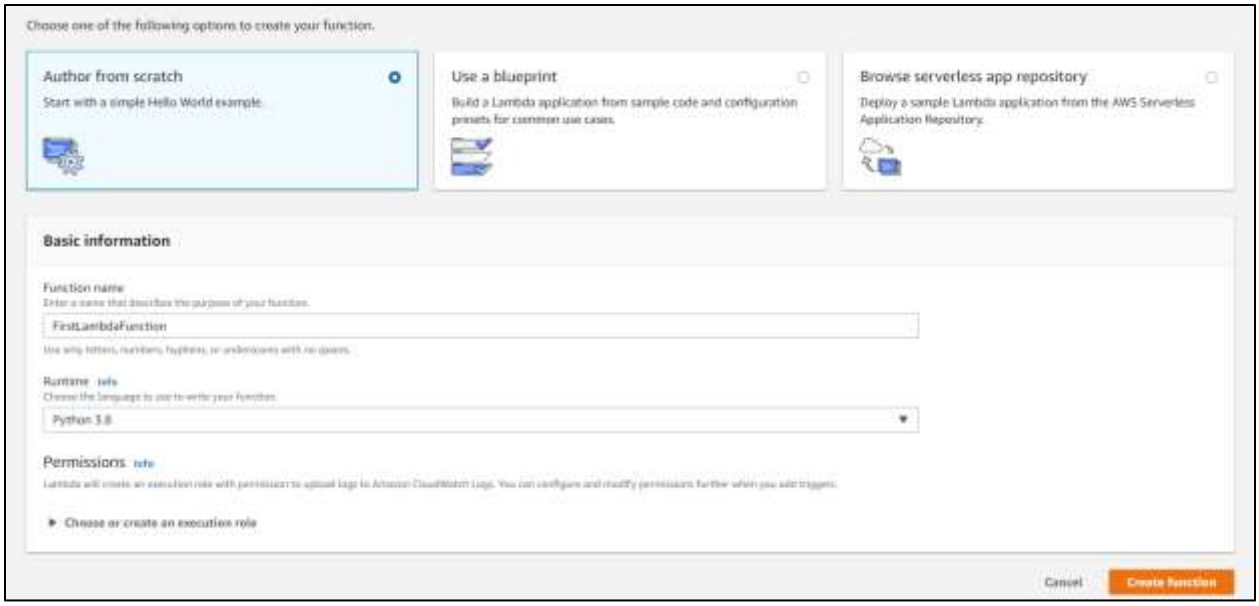
**Requirements for AWS Lambda:**

- ➔ We need to provide the handlers. Basically, handler is a function and it acts as an entry point for the AWS Lambda.
- ➔ Lambda accepts JSON-formatted inputs and provides output in the JSON as well.
- ➔ We need to specify the Runtime environment for Lambda. The runtime correlates with the language that we are using to write the code. Eg: Python.
- ➔ Final requirement is a trigger. The trigger can be manual or automatic.
- ➔ Manual Trigger will be run by us in response to any event occurring in the console.
- ➔ Automatic Trigger happens whenever an event occurs in the AWS console such as:
  a. **New File getting uploaded to S3.**
  b. **Change in the DynamoDB table.**
  c. **AWS API Gateway.**
- ➔ We can also configure Lambda to trigger based on the timer triggered by the AWS CloudWatch.
- ➔ We can run the AWS Lambda with two ways:
  a. **Creating Programmatic Access keys.**
  b. **Creating IAM Role and attaching the role to AWS Lambda.**

**Working on AWS Lambda from the console:**

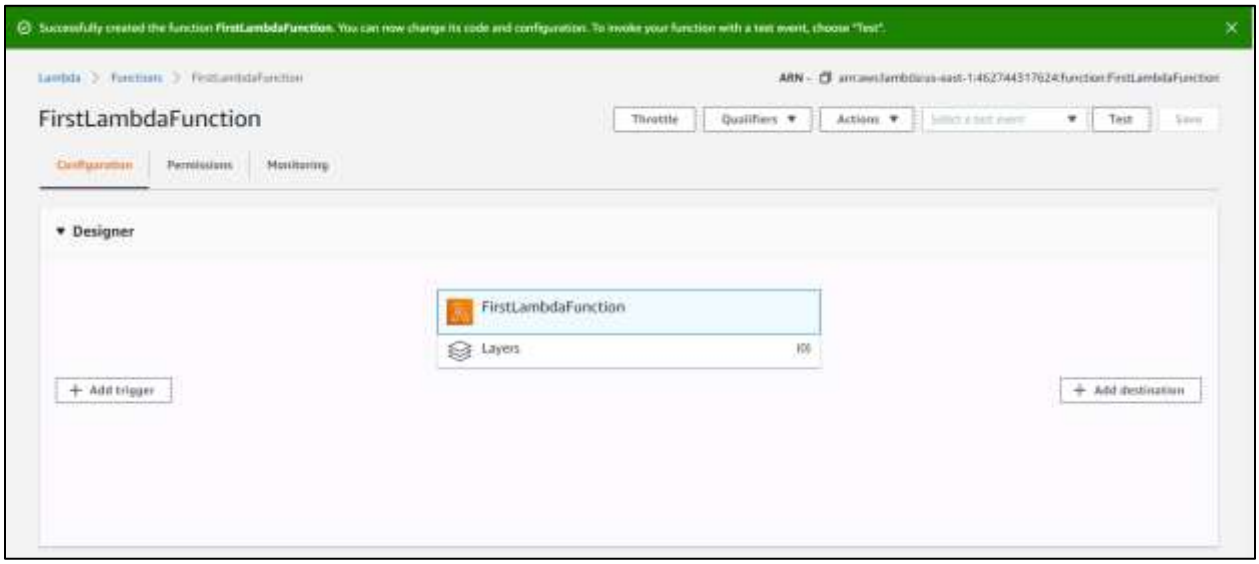
- ➔ Logon to AWS Management Console.
- ➔ Navigate to Compute > and you will get the service named Lambda. Click on it.


- ➔ By default, we have no handler functions, so it will be empty by default.
- ➔ Now click on Create Function to start writing a script.
- ➔ You will get the following screen, we need to enter the function name and the runtime environment and presently we will be working on “Creating from scratch”.

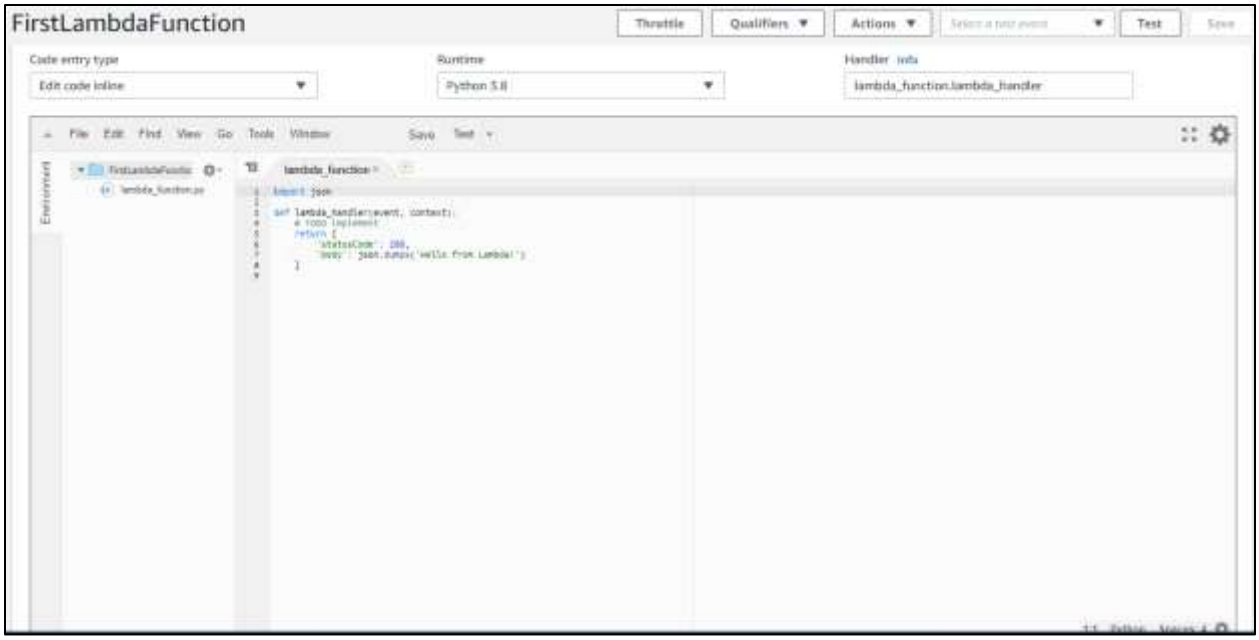



➔ After entering the details, click on Create Function to start the Lambda.

➔ We get this dialog box on the top that allows us to add Triggers and Destinations:

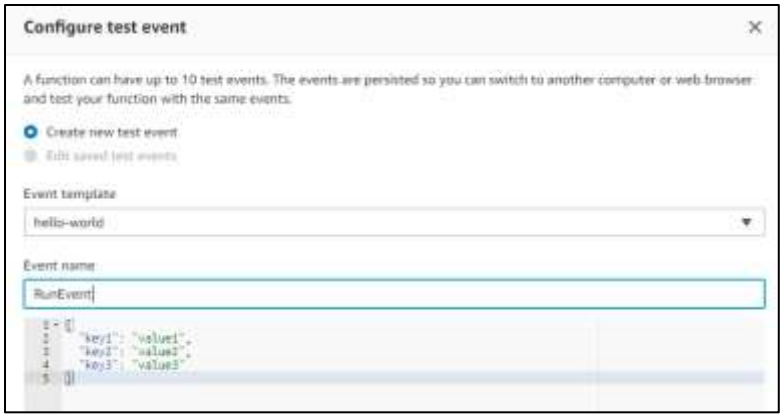


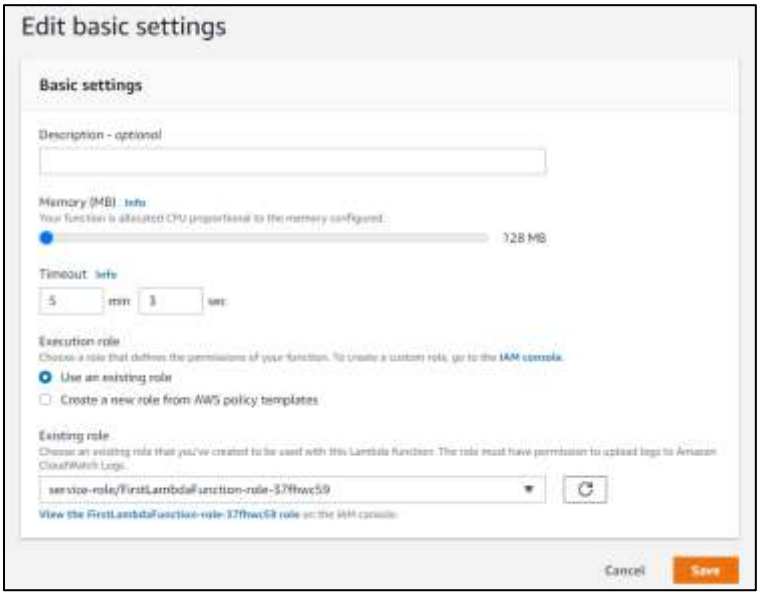

➔ Scroll down to find the editor area, where you can type all the code in python language:




- ➔ We see that there is already a handler function created by default. This is the entry point of the program where the script executes.
- ➔ Now let's write a simple program to list all the IAM Users in our IAM.
- ➔ Here, lets pass our AWS access key and AWS Secret key as parameters to session object instead of calling the profile_name.

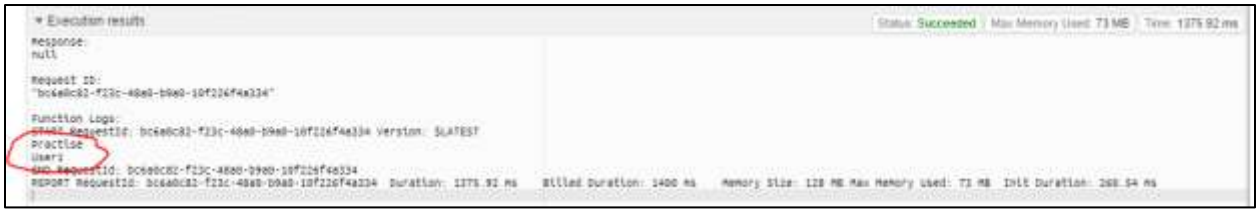

- ➔ Now we have to run the program by clicking on Test Button on the top. Make sure to create an event with a name to run the program.
- ➔ Now we need to select the event we created and click on Test to get our output.
- ➔ Remember, that we have only 3 seconds for our program to get timeout. We need to change this settings by scrolling down and changing the timeout to more than 5 minutes for safer side.



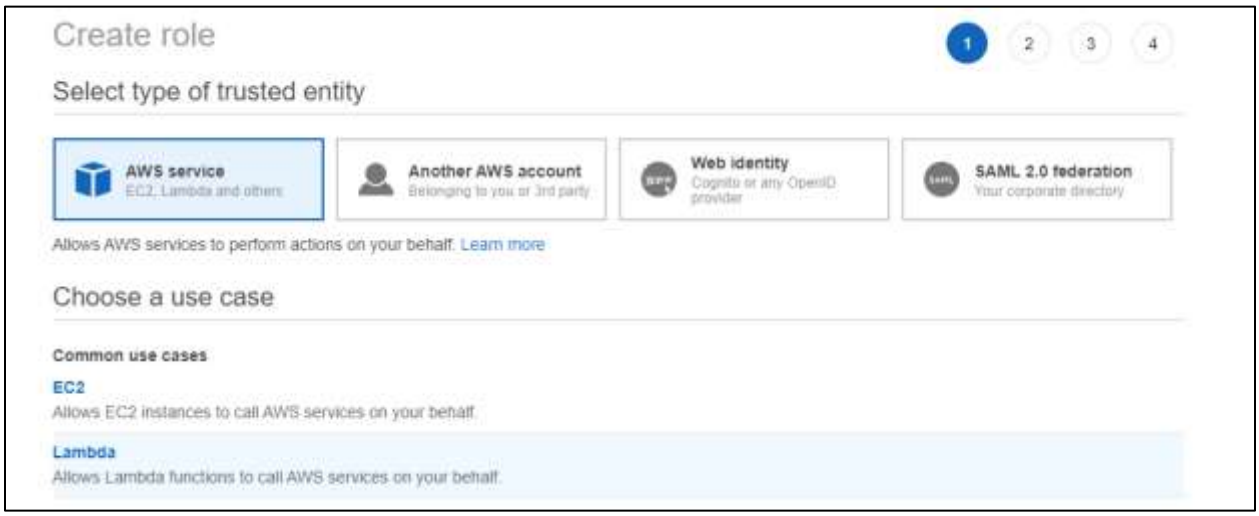




➔ Now we will be running the program with our Event and clicking on Test.

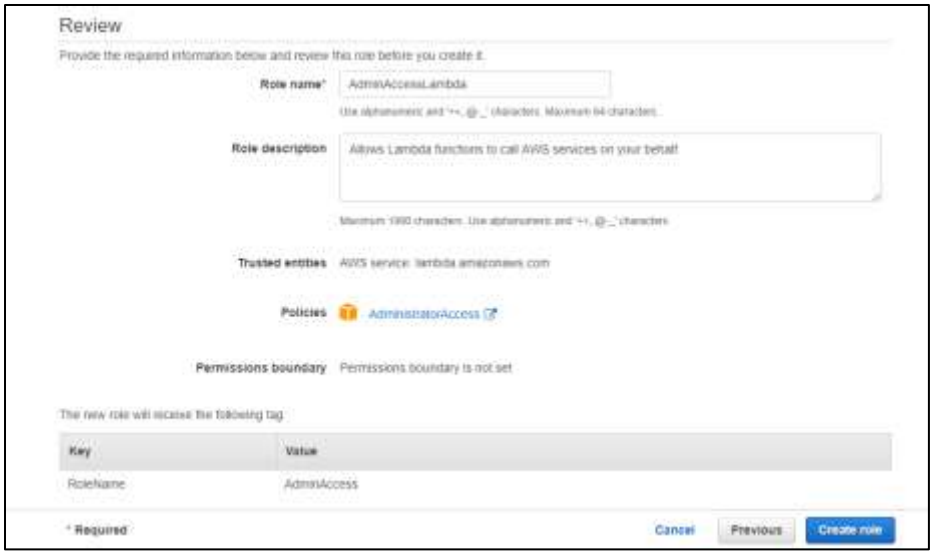

➔ The sample output will be displayed on the bottom of the code along with JSON response and time, memory taken to execute the script like this:



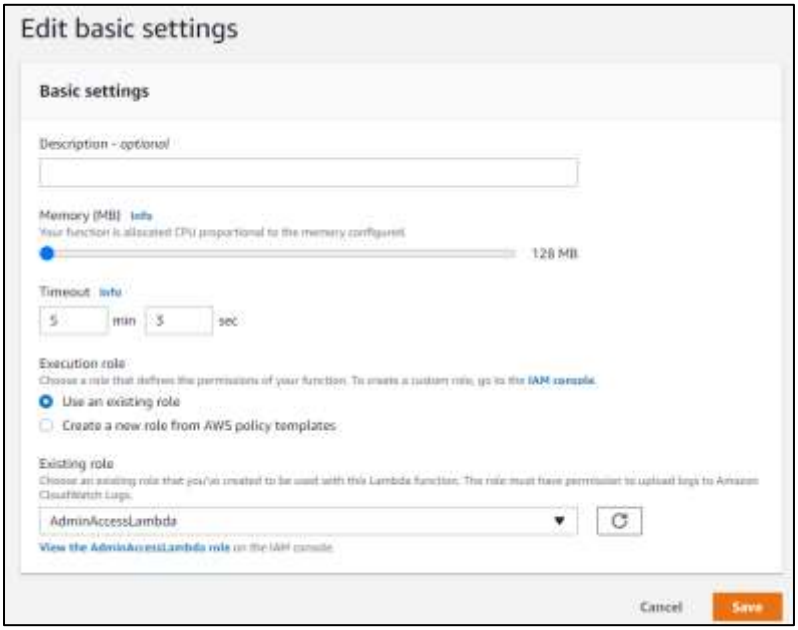

- ➔ If we want to work on the environment variables here, we can make use of the environment variables section provided here.
- ➔ In local machine, we make us of the "os" module to work with environment variable, but here we can directly use such variables in a different section.
- ➔ Suppose, we want to assign a role to our AWS Lambda and work based on roles instead of passing the secret and access keys in our program, we can do it in roles section.
- ➔ Now we will creating the role for our AWS Lambda by going to IAM console.
- ➔ Click on create new role and select "AWS Service" and select "Lambda" below:




- ➔ And then select "AdminAccess" role and click on Next and provide the Role Name, Role Description, Trusted Entities and click on create Role.




➔ Now we will be assigning the role to AWS Lambda, go back to the basic settings where we changed the timeout, there select the newly created role and hit on save.




➔ After assigning the role, we no need to create sessions, we can make use of the default sessions for operations. Now change the sessions lines and make it to default session.

```
import boto3

def lambda_handler(event, context):
    IAM_CONSOLE_AWS = boto3.resource(service_name='iam')
    for each_user in IAM_CONSOLE_AWS.users.all():       #Getting all the users from IAM
        print(each_user.name)                           #Printing the users names
```

➔ Now we can see that we are getting the output as same as our previous programs, list of all users.

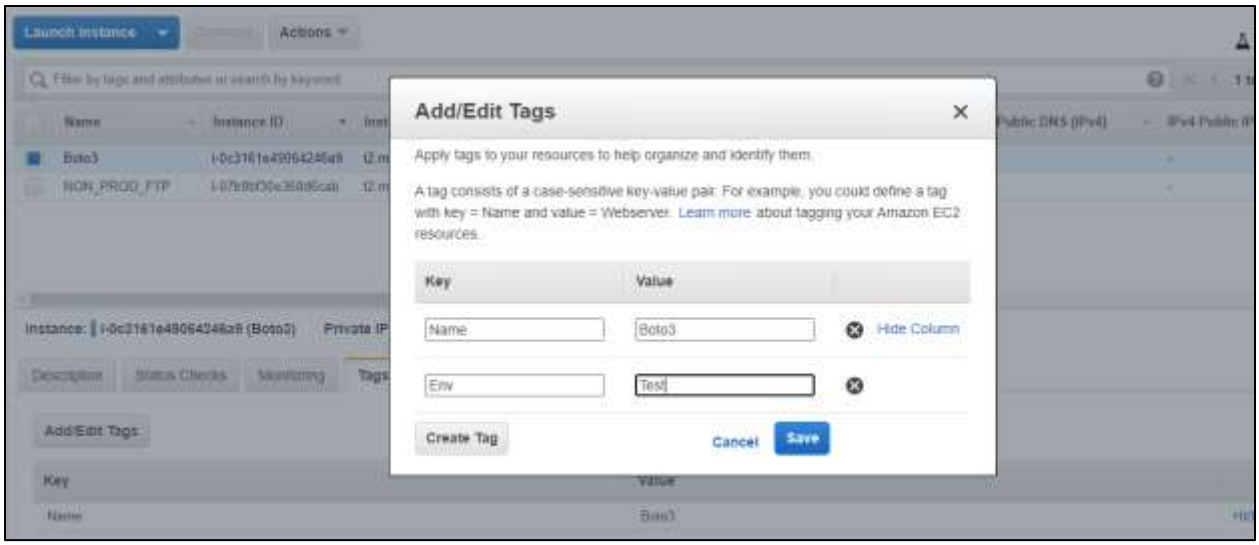
➔ If we try to access anything outside of our roles, then we get the ACCESS DENIED exception in the output.

**Automating the Start and Stop of EC2 Instances from Test Environment:**

➔ Suppose we want to start and stop some number of instances in our office hours and we need to automate it using LAMBDA, we can do it here.

Steps to be followed for implementing this:

a. **Create an IAM Role for the AWS Lambda, if already exists then attach that role to AWS Lambda.**
b. **Write the required Lambda function using python and BOTO3 in Lambda console.**
c. **Assign a tag to our required instances to perform operation with Keyname as "Env" and Value as "Test".**

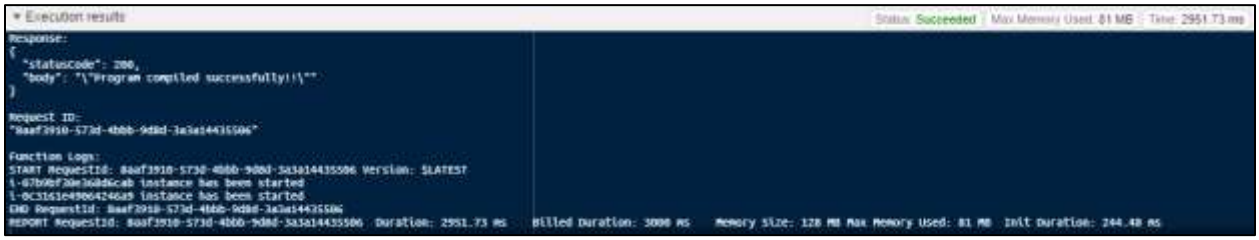



d. **Write separate Lambda Function for starting and stopping the instances. That is, we need to write two Lambda functions.**
e. **Firstly, we will start with writing the Start function, and make necessary changes such as Roles, Timeout settings and Test Events configuration.**

➔ Now we will be writing the code to start the instances which have the tagname with "Env" as "Test"

```python
import json
import boto3

def lambda_handler(event, context):
    EC2ResourceObject = boto3.resource(service_name='ec2',region_name='us-east-1')
    filter1 = {'Name':'tag:Env','Values':['Test']}      #Filtering the instances based on the tagname
    for eachInstance in EC2ResourceObject.instances.filter(Filters=[filter1]):
        print("{} instance has been started".format(eachInstance.id))
        eachInstance.start()        #To start the instance we filtered
    return {
        'statusCode': 200,
        'body': json.dumps('Program compiled successfully!!')
    }
```

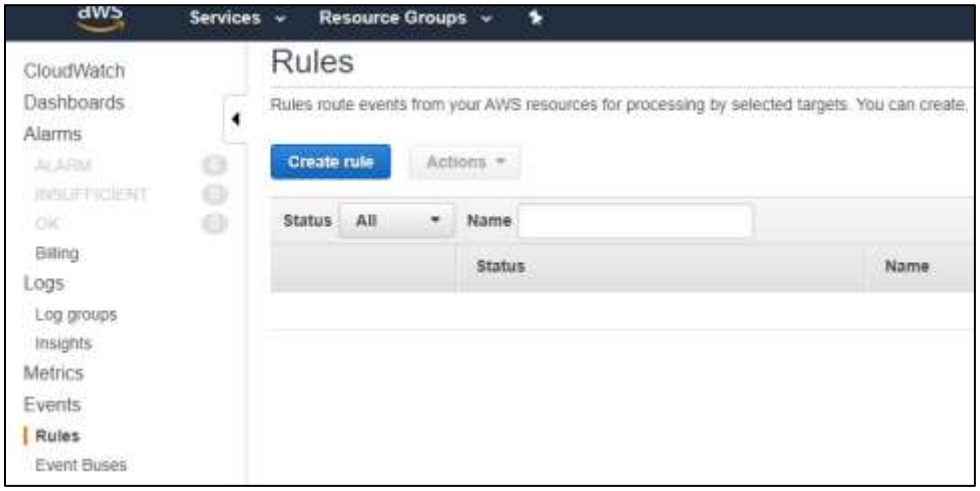
➔ The output will be like this:




➔ Now, we need to schedule the job at required time.

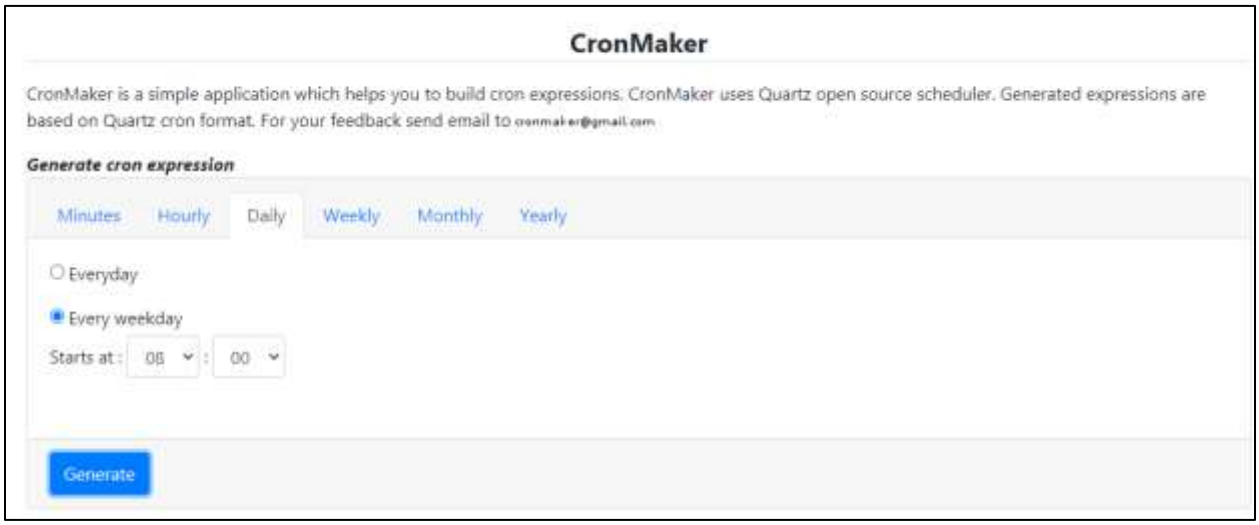
➔ For scheduling the script to run at a specified time, we will be using the cloudwatch's rules section and writing a rule to run the script at needed time.

➔ Navigate to Cloud-watch and select the rules and select "Create-Rule"

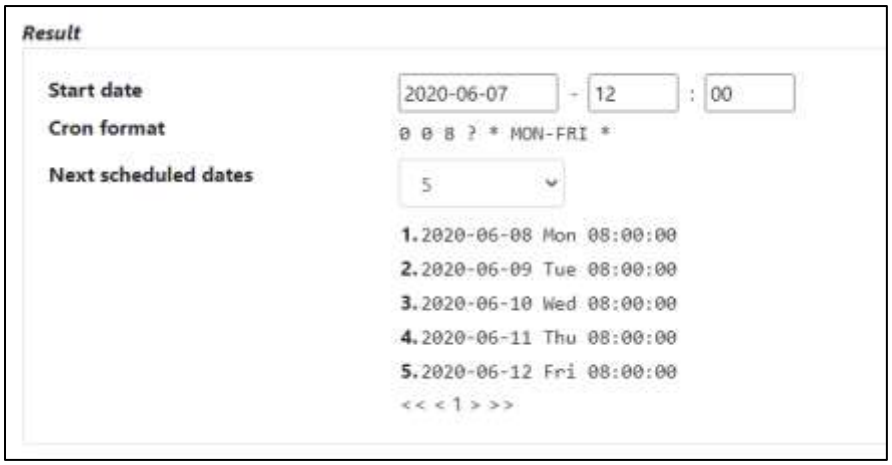



➔ Now select the cron job expression and provide the cron pattern to schedule the job

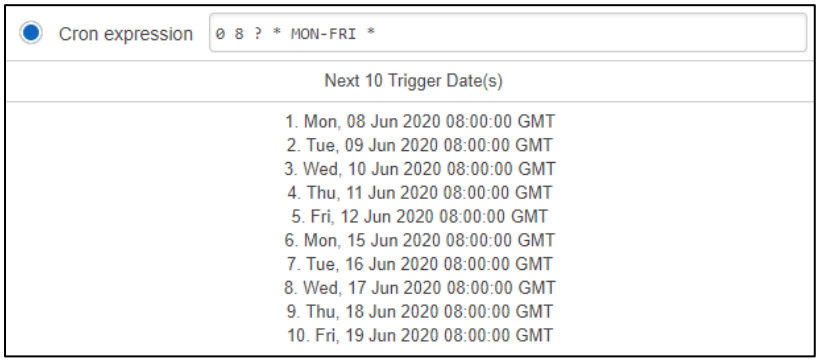
- ➔ For making the cron expressions, navigate to: http://www.cronmaker.com/?1
- ➔ Enter the conditions and it will generate the cron expression for you.
- ➔ We need to schedule the job for every weekday starting at 8 am in the morning, so we will select this:



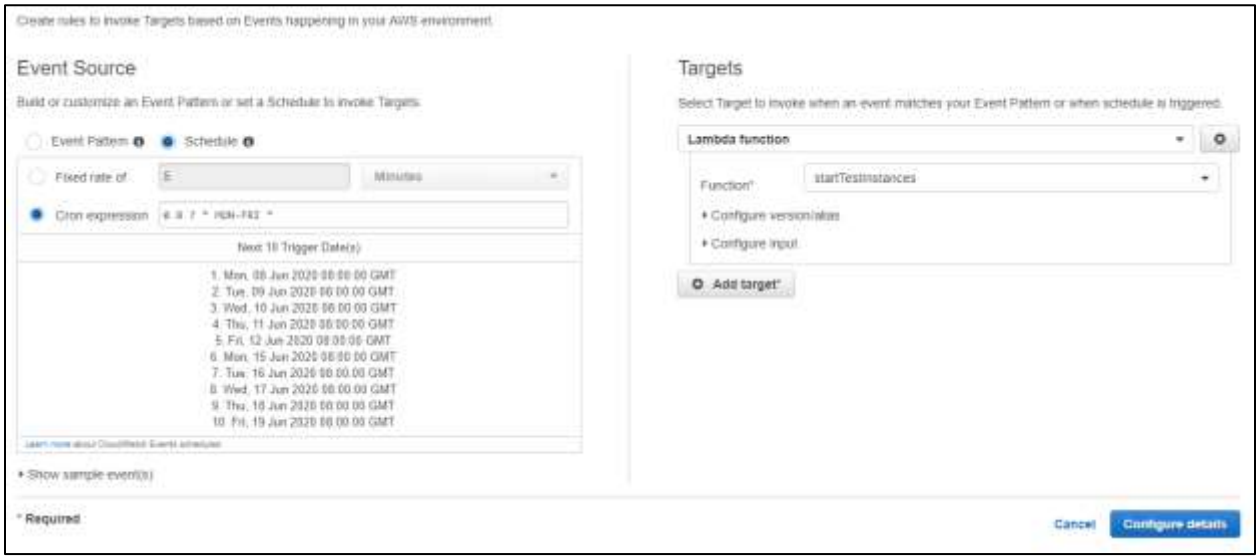

- ➔ After hitting generate, we will get the cron expression and detailed explanation about it.



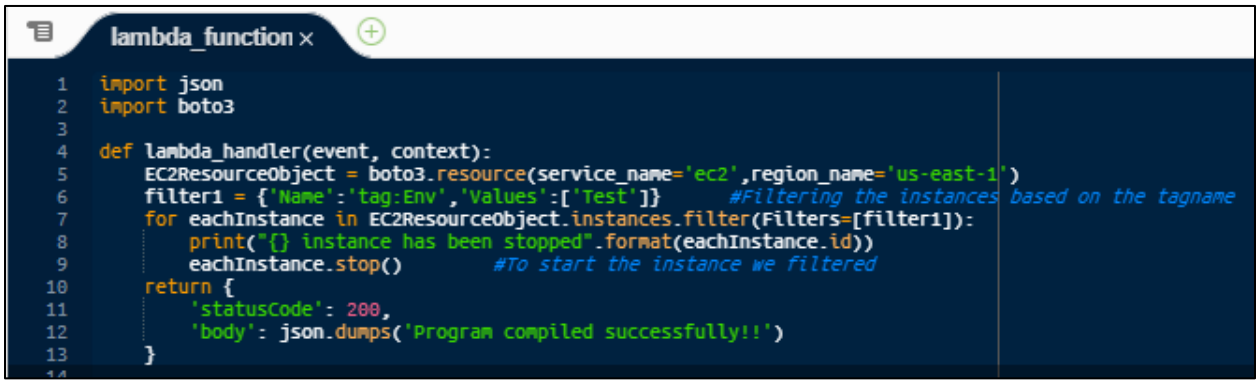

- ➔ After pasting the same in the crontab tab in cloudwatch, we will get the detail like this:



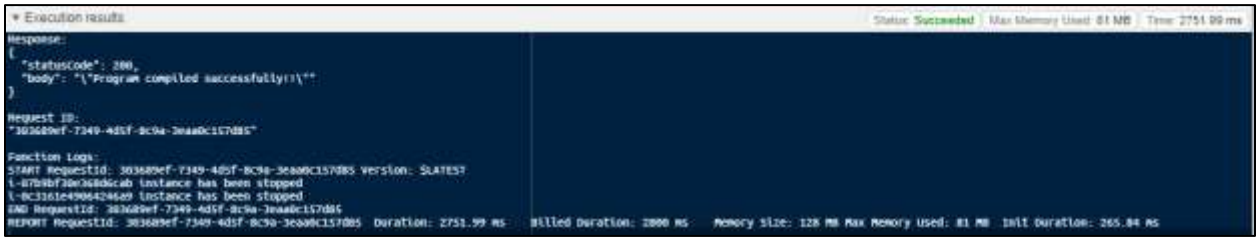
- ➔ Next, we need to select the target where we need to attach the Lambda function on which this cronjob must work.
- ➔ This must be our final screen, when creating the schedule:



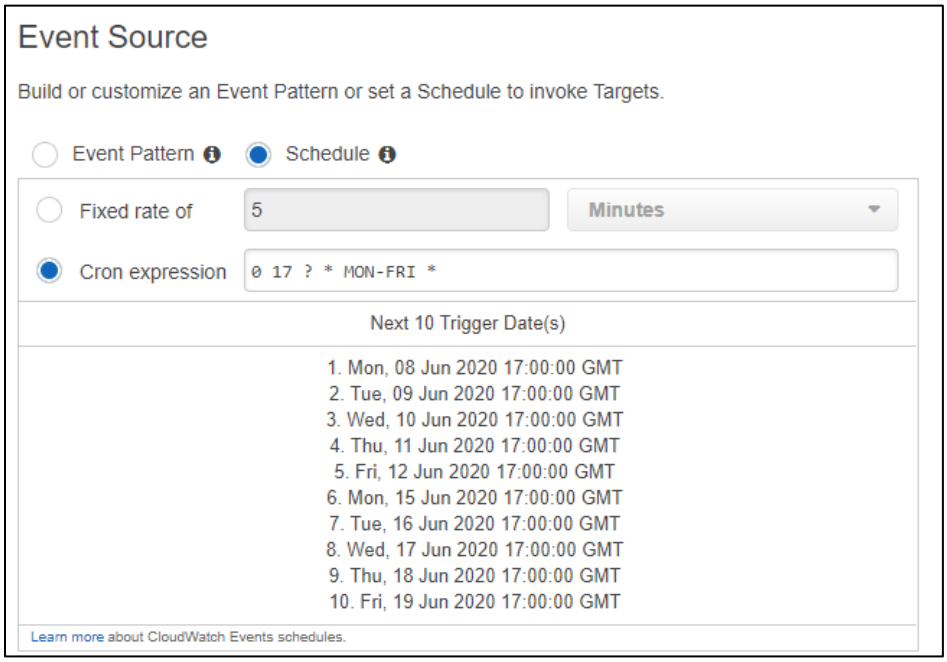

- ➔ Now click on Configure Details to go to next screen.
- ➔ And provide the Rule name and description and hit on create to create this new rule under CloudWatch.
- ➔ Now you can see that at provided specific time, the lambda script will start to execute.
- ➔ Now we will be writing the script for stopping the EC2 instances with Test Environment like this:




- ➔ The output would be like this:



➔ Next, we will be writing another cron job in CloudWatch to stop the scripts at 5 PM on every weekday like this:




➔ And adding the stop Lambda function to its target and creating this rule to execute at specified time.

➔ Remember to delete the function if not in use, and also to remove the log groups from CloudWatch > Log Groups if not in use as they may cost for usage.

**Automatic Mail Alert when Instance state is reached to Stop state:**

➔ We will be using the SNS service to send the emails, make sure that the IAM role has EC2 and SNS access.

➔ We will be writing a new lambda function named "**SendSNSEmail**"

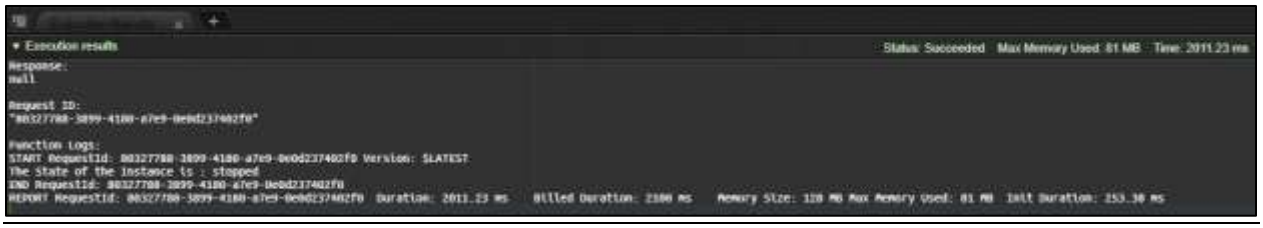
➔ We will be taking Resource object for Ec2 and Client Object for SNS.

➔ First, we will be printing the current state of the instance like this:

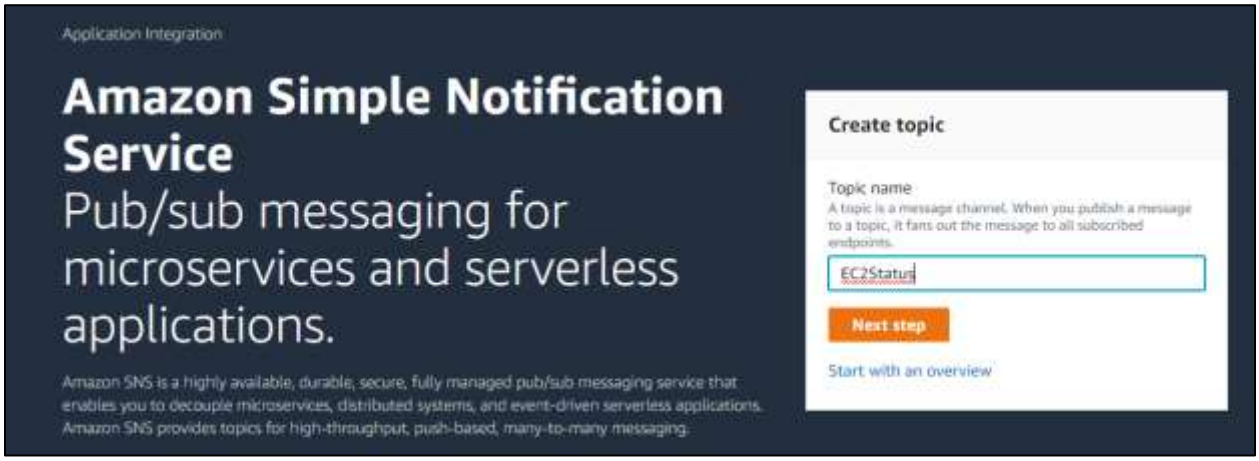
lambda_function.py

```
import json
import boto3

def lambda_handler(event, context):
    EC2_OBJECT = boto3.resource(service_name='ec2',region_name='us-east-1') #EC2 Resource Object
    SNS_OBJECT = boto3.client(service_name='sns',region_name='us-east-1')   #SNS Client Object
    ExampleInstance = EC2_OBJECT.Instance('i-0c3161e49064246a9')        #Taking our BOTO3 instance ID
    print("The State of the Instance is : {}".format(ExampleInstance.state['Name']))    #Printing the current state of the EC2 instance
```

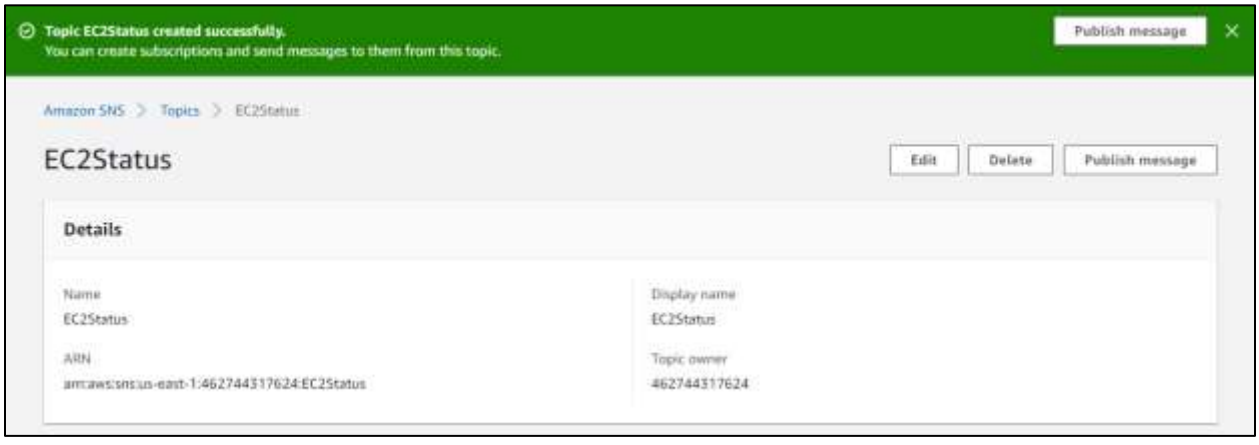
➔ The output will be like this:



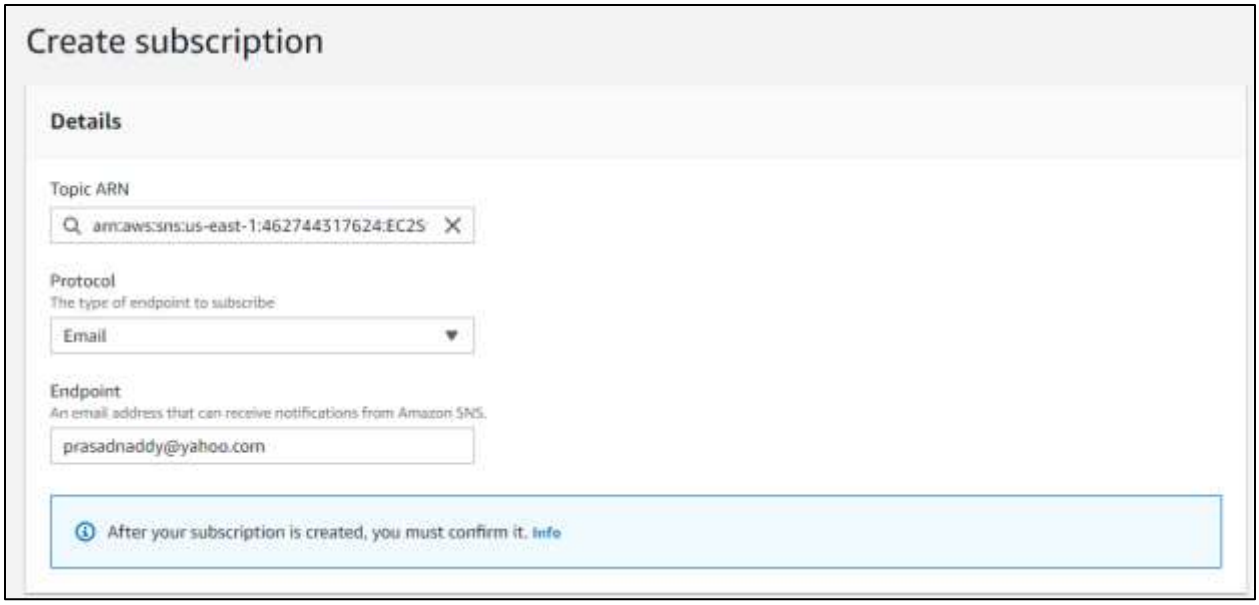

➔ Before sending the email, we need to make sure that the Topic and Subscriptions are done in SNS end also.
➔ Currently we don't have any topics and subscriptions created, so let's go ahead and create one.
➔ Search for SNS in AWS Service and navigate to the SNS page.
➔ If we don't have any topics created, we will be getting the screen like below, enter the topic name:



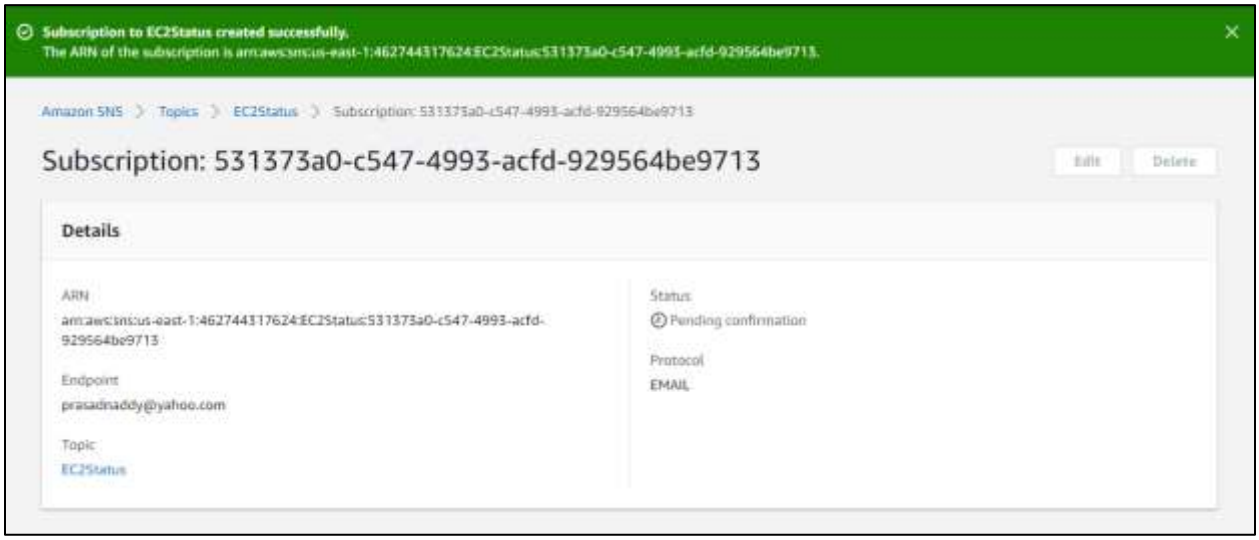

➔ Go to next page and enter the display and click on create, you will get the next screen like this:




➔ Now its time to create the subscriptions, navigate down and click on "Create Subscriptions"
➔ Now, enter the type and target email address like this:

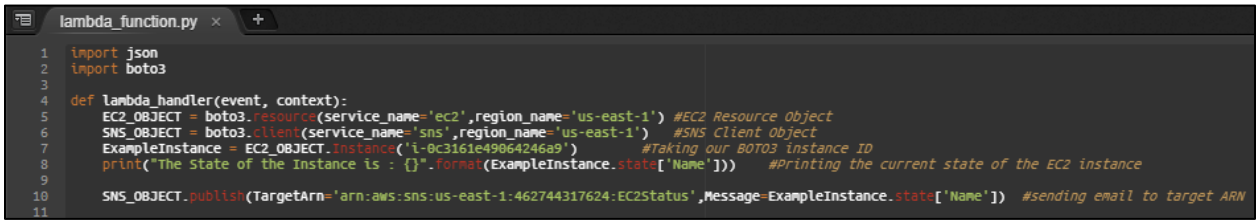



➔ You will get this page, and waiting for the completion:




➔ Now open your target email and click on the email from AWS and click on “Confirm Subscription” to activate the email service.

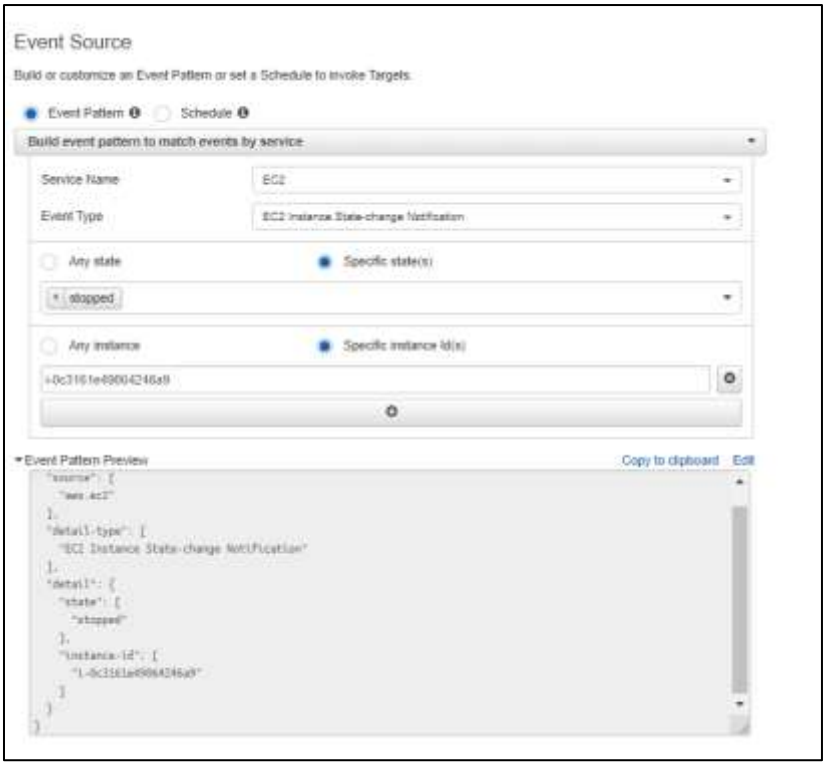
➔ After that, we will be adding a new line to the code for sending the email using SNS like this:



➔ Now, after running this script, if we check our target email then we will have an email regarding the state of the instance:


➔ Now we need to setup a rule such that we receive an email whenever our instance goes to stopped state, we will be doing it in CloudWatch Rules.

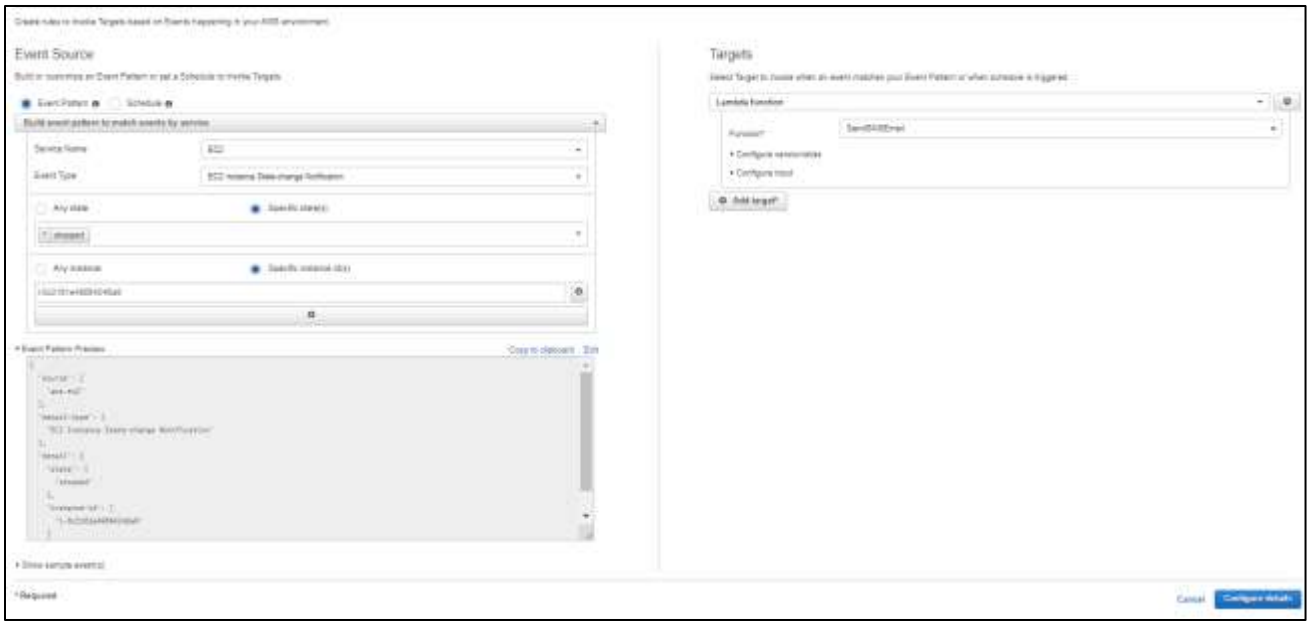
➔ We will be configuring the Event state rule like this:




➔ We will be specifying the service name as EC2, we need to set it for state changes to stopped state so we will be selecting “EC2 Instance State-Change Notification” and then selecting specific state to Stopped state and providing the instance ID on which this must apply.

➔ We get the Event Pattern Preview in JSON format, we can check that JSON to validate our requirement.

➔ Now add the respective Lambda Function in “Add Target”. Our final screen should look like this:




➔ Now click on Configure Details and add the Rule name and description. That should set up the rule in CloudWatch.

➔ Whenever our instance stops, the rule present in the JSON format will be sent as event parameter to our handler in lambda function that will process the next steps in the code.

➔ So, we no longer need to take the control of the EC2 instances, so we will be removing the EC2 instance logic from lambda function as event parameter will take care of everything. We will be adding only the SNS logic in the code like this:


➔ Now we can see that we will get a new email once our target ec2 instances go into stopped state.

**Paginators:**

➔ Suppose, we are having more than 1000 objects inside the EC2 or IAM and we are fetching the information by using client object from AWS Console, we have the limitations for listing the objects from client object to either 50 or 100.

➔ To overcome this limitation, we will be using the Paginators.

➔ But S3 will return max of 1000 results at once.

➔ But if we use the resource object, then we get all the results without any limitations.

➔ For example, if we make a client API call to the IAM service for listing all the users with their usernames, we will get only 100 users in the output, and we will be missing the other users.

- ➔ Each output will be considered as a "**page**", each page will be having results of either 50 or 100 based on the limitations of the service, by default we are going to get only first page as result.
- ➔ Suppose there are 204 users in our IAM console, first time Paginator will hit the API and fetch the 100 users in first page, it will hit the API again and fetch next 100 users in 2nd page and then it will hit the API again and fetch the remaining users in 3rd page.
- ➔ After the list of users is over, pagination will be stopped automatically and results are displayed on the screen.
- ➔ Paginators are not available for all of the services in AWS, we need to write a custom paginator if it doesn't exist.

**Steps for working with Paginators:**

1. **Create a paginator.**
2. **Navigate through Paginators to get the pages.**

- ➔ The sample script for listing the count of each users in a single page could be like this:

paginatorsScript.py > ...

```
import boto3        #module for working with boto3

AWS_MGMT_CONSOLE = boto3.session.Session(profile_name='root')   #configuring with root profile
IAM_CLIENT_OBJECT = AWS_MGMT_CONSOLE.client(service_name='iam') #running on IAM with client object

paginatorObject = IAM_CLIENT_OBJECT.get_paginator('list_users') #paginator object on list_users function

for eachPage in paginatorObject.paginate():     #iterating over the paginator object to get the pages
    print(len(eachPage['Users']))       #getting count of users on each page
```

- ➔ We get the count line by line based on the pages.
- ➔ Now we will be printing all the user names present in our IAM console using the paginators like this:

```
#function to get the usernames from the pages
def getUserNamesInPages():
    try:
        for eachPage in paginatorObject.paginate():
            for eachUser in eachPage['Users']:
                print(eachUser['UserName'])
    except Exception as e: print("Issue getting the usernames : {}".format(e))
```

- ➔ We will get all the names of the users present in IAM now without any restrictions.
- ➔ We will be writing another function to get all the instance id with help of paginators:

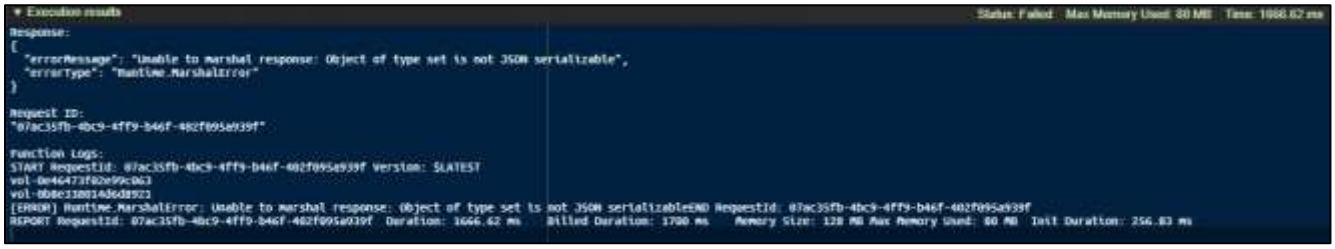
```python
#function to list all the instances id from the pages
def listInstancesFromPages():
    try:
        EC2PaginatorObject = EC2_CLIENT_OBJECT.get_paginator('describe_instances')  #ec2 paginator object
        for eachPage in EC2PaginatorObject.paginate():  #iterating over the paginator object to get pages
            for eachInstances in eachPage['Reservations']:  #iterating over the key 'Reservations'
                for eachInstance in eachInstances['Instances']: #iterating over the key 'instances'
                    print(eachInstance['InstanceId'])       #printing all the instance id from console
    except Exception as e: print("Issue listing the instance id : {}".format(e))
```

**Automate Snapshots for EBS Volumes using Lambda and CloudWatch:**

- ➔ Firstly, we would start with listing all the EBS Volumes based on requirement.
- ➔ This is only region specific and going forward we would be dealing with all regions.
- ➔ Consider that we have some volumes having tags: **“Key”: “Prod” and “Values”: “Backup”**, we want to take backups for only these volumes in our EBS.
- ➔ Now go to AWS Lambda, create a new function and assign the necessary roles to the Lambda.
- ➔ Set the timeout as required such as 5 mins or 10 mins. Max is 15 mins only.
- ➔ To save the Code press “**Ctrl + Shift + U**” and to run the code, type “**Ctrl + I**”
- ➔ We will be listing all the volume ID by filtering with the tag names as mentioned in the point number 3 like this:

```python
import json
import boto3

def lambda_handler(event, context):

    EC2_Object = boto3.client(service_name='ec2',region_name='us-east-1')   #setting the service name and region name for ec2
    FilterByTag = {'Name':'tag:Prod','Values':['Backup']}

    for eachVolume in EC2_Object.describe_volumes(Filters=[FilterByTag])['Volumes']:
        for volumeDetails in eachVolume['Attachments']:
            print(volumeDetails['VolumeId'])

    return {
        'Code Compiled Successfully!'
    }
```

- ➔ The output will be like this:




- ➔ We would be appending the volume ID to a list to have the information stored at a place.

```python
import json
import boto3

def lambda_handler(event, context):

    listOfVolumeIDs = []

    EC2_Object = boto3.client(service_name='ec2',region_name='us-east-1')   #setting the service name and region name for ec2
    FilterByTag = {'Name':'tag:Prod','Values':['Backup']}

    for eachVolume in EC2_Object.describe_volumes(Filters=[FilterByTag])['Volumes']:
        for volumeDetails in eachVolume['Attachments']:
            listOfVolumeIDs.append(volumeDetails['VolumeId'])

    print(listOfVolumeIDs)
```

➔ The output would be in a list format:

```
▼ Execution results
Response:
null

Request ID:
"cd332c73-bab4-4310-b5b0-03adb60089b0"

Function Logs:
START RequestId: cd332c73-bab4-4310-b5b0-03adb60089b0 Version: $LATEST
['vol-0e46473f02e99c063', 'vol-0b8e338014d6d8921']
END RequestId: cd332c73-bab4-4310-b5b0-03adb60089b0
REPORT RequestId: cd332c73-bab4-4310-b5b0-03adb60089b0  Duration: 1677.27 ms
```

➔ Beware that this is applicable for only 50 volumes, if there are more than 50 volumes in the region then we may not get all the volume ID, to overcome this situation we are using paginators.

➔ We would be taking the paginator for "**describe_volumes**".

➔ Make sure to add an extra loop, for fetching the pages first and then followed by logic.

➔ The updated code including paginators for listing the volumes based on tagnames would be like this:

```python
import json
import boto3

def lambda_handler(event, context):

    listOfVolumeIDs = []

    EC2_Object = boto3.client(service_name='ec2',region_name='us-east-1')   #setting the service name and region name for ec2
    FilterByTag = {'Name':'tag:Prod','Values':['Backup']}

    PaginatorObject = EC2_Object.get_paginator('describe_volumes')  #paginator object for listing the volumes

    for eachPage in PaginatorObject.paginate(Filters=[FilterByTag]):    #iterating on paginator object to get pages
        for eachVolume in eachPage['Volumes']:
            for volumeDetails in eachVolume['Attachments']:     #iterating over Volumes > Attachments to get volume ID
                listOfVolumeIDs.append(volumeDetails['VolumeId'])

    print(listOfVolumeIDs)
```

➔ We will get all the volume ID appended to a list.

- ➔ Now, we will be creating snapshot for each and every volume ID stored inside of the list by using "**create_snapshot()**" function of EC2 client object.
- ➔ Complete Lambda Script for creating the snapshot for tagged volume:

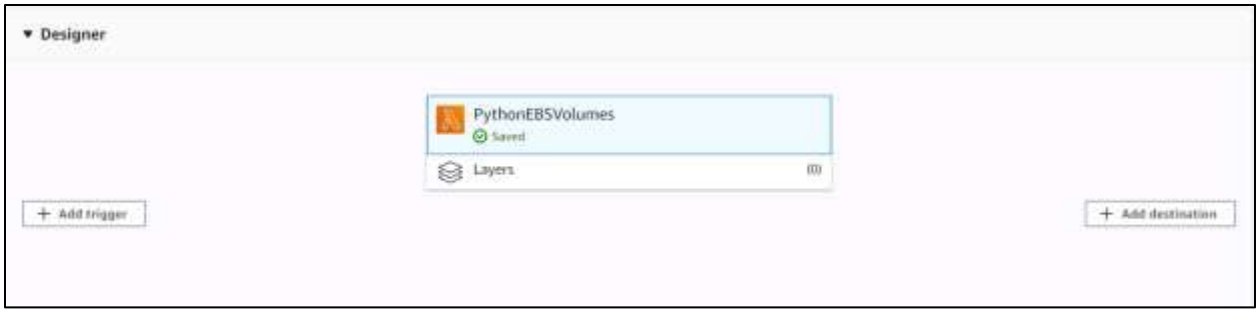
```
import json
import boto3

def lambda_handler(event, context):

    listOfVolumeIDs = []

    EC2_Object = boto3.client(service_name='ec2',region_name='us-east-1')   #setting the service name and region name for ec2
    FilterByTag = {'Name':'tag:Prod','Values':['Backup']}

    PaginatorObject = EC2_Object.get_paginator('describe_volumes')  #paginator object for listing the volumes

    for eachPage in PaginatorObject.paginate(Filters=[FilterByTag]):    #iterating on paginator object to get pages
        for eachVolume in eachPage['Volumes']:
            for volumeDetails in eachVolume['Attachments']:     #iterating over volumes > Attachments to get volume ID
                listOfVolumeIDs.append(volumeDetails['VolumeId'])

    for eachVolume in listOfVolumeIDs:      #iterating over the volume id stored in a list
        EC2_Object.create_snapshot(Description='Taking the important snapshot for the day',VolumeId=eachVolume,TagSpecifications=[{'ResourceType':'snapshot',
            'Tags':[{'Key':'Delete-On','Value':'90 Days'}]}])
```

- ➔ Now if we login to Snapshots, we would find a snapshot getting created like this:

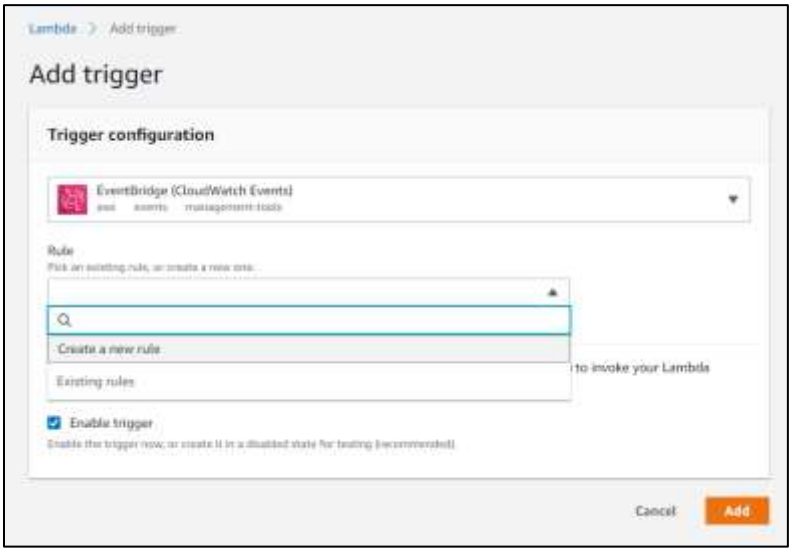

- ➔ Suppose, our volume is too large and our code gets exited without waiting for the completion, in that case we need to include the waiters for our snapshots.
- ➔ Since, the waiter for snapshots requires the snapshot ID to be passed, we need to capture the snapshot ID created back from our create_snapshot function.
- ➔ The final script for waiting till snapshot creation and stopping the script would be like this:

```
import json
import boto3

def lambda_handler(event, context):

    listOfVolumeIDs = []
    createdSnapIDList = []

    EC2_Object = boto3.client(service_name='ec2',region_name='us-east-1')   #setting the service name and region name for ec2
    FilterByTag = {'Name':'tag:Prod','Values':['Backup']}

    PaginatorObject = EC2_Object.get_paginator('describe_volumes')  #paginator object for listing the volumes
    WaiterObject = EC2_Object.get_waiter('snapshot_completed')      #waiter object for waiting until snapshot is finished

    for eachPage in PaginatorObject.paginate(Filters=[FilterByTag]):    #iterating on paginator object to get pages
        for eachVolume in eachPage['Volumes']:
            for volumeDetails in eachVolume['Attachments']:     #iterating over volumes > Attachments to get volume ID
                listOfVolumeIDs.append(volumeDetails['VolumeId'])


    for eachVolume in listOfVolumeIDs:      #iterating over the volume id stored in a list
        response = EC2_Object.create_snapshot(Description='Taking the important snapshot for the day',VolumeId=eachVolume,TagSpecifications=[{'ResourceType':'snapshot',
            'Tags':[{'Key':'Delete-On','Value':'90 Days'}]}])
        createdSnapIDList.append(response['SnapshotId'])

    WaiterObject.wait(SnapshotIds = createdSnapIDList)      #passing the list of snapshot IDs
    print("Created snapshot for {}".format(createdSnapIDList))
```

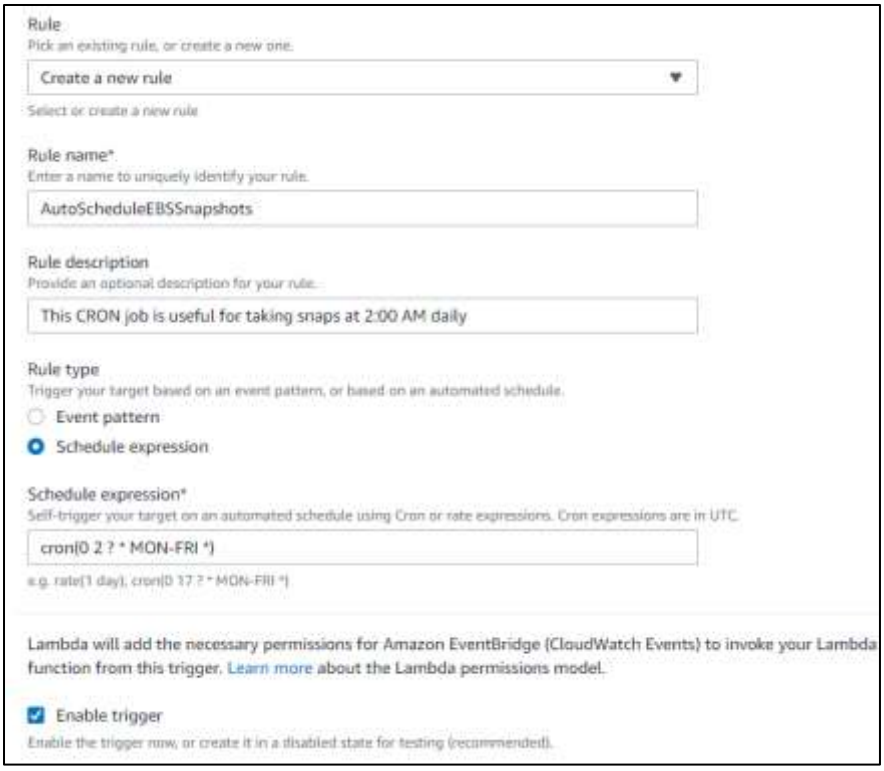
- ➔ We can use our old approach by going to cloudwatch and triggering the lambda script at specified time or else add it as a trigger in lambda console itself.
- ➔ We will be adding the cloudwatch rule as trigger to our lambda in this process.
- ➔ Click on "**Add Trigger**" button.




➔ And now select “EventBridge” for adding the rules for events.




➔ Now click on New rule and add the rule name and description. Since we need to schedule the rule, so we will be adding the CRON expression to schedule the job like this:




- ➔ This will schedule the job for every week day at 2 AM in the morning to take the snapshots.
- ➔ Click on Add to add the trigger to our lambda function.

**Automate Snapshots of EBS Volumes for all regions:**

- ➔ Previous script dealt with only the single region EBS Volumes, now we will be collecting the volumes from all the regions and applying the same logic as before.
- ➔ This program will be optimized version, so will be using generators and list comprehension wherever necessary.
- ➔ First, we will be listing all the regions available in our AWS EC2 console like this:

```
AutomateEBSVolumes.py > ...
1   import boto3            #module for working with boto3 scripts
2
3   AWS_MGMT_CONSOLE = boto3.session.Session(profile_name='root')   #Working with root profile
4   EC2_CLIENT_OBJECT = AWS_MGMT_CONSOLE.client(service_name='ec2',region_name='us-east-1') #working on ec2 client object
5
6   for eachRegion in EC2_CLIENT_OBJECT.describe_regions()['Regions']:  #iterating on 'regions' key in describe_regions
7       print(eachRegion['RegionName'])     #printing each region name
```

- ➔ The complete optimized script is named as '**AutomateEBSVolumes.py**'
- ➔ Making use of list and generator comprehension will make our code run efficiently and allows program to consume less memory.
- ➔ We can also make this code running in Lambda and schedule the job using CloudWatch.

**Automate Copy of EBS Snapshots between Regions:**

- ➔ We need to take the snapshots copy from one region to another because of three reasons:

a. **Geographic Expansion:** We will be able to launch our application from a new region.
b. **Migration:** We can migrate our application from one region to another.
c. **Disaster Recovery:** Taking backups in regular intervals allows us to recover the data in case of unexpected outages.

- ➔ We will be naming our backup snapshots with Tags such as "**Key**":"**Backup**", "**Values**":"**Yes**"
- ➔ Now we will be listing the snapshots based on this filter and taking the backups to the cross-regions.
- ➔ We will be listing the snapshots by using the describe_snapshots() function, that will by default list all the public, private and owned by me snapshots. That is nearly around 22k snapshots which is of no use for us.
- ➔ So, we will be passing the owner ID with our AWS Account ID to list only our snapshots.
- ➔ The sample code for listing all the snapshots in the source region would be like this:

```
import boto3            #importing the boto3 module

SOURCE_REGION = 'us-east-1'     #setting the source region to copy the snapshots
DEST_REGION = 'us-east-2'       #setting the destination region to migrate snapshots

AWS_MGMT_CONSOLE = boto3.session.Session(profile_name='root')   #working with root profile
EC2_CLIENT_OBJECT = AWS_MGMT_CONSOLE.client(service_name='ec2',region_name=SOURCE_REGION)   #ec2 client object
STS_CLIENT_OBJECT = AWS_MGMT_CONSOLE.client(service_name='sts',region_name=SOURCE_REGION)   #sts client object for account

accountID = STS_CLIENT_OBJECT.get_caller_identity()['Account']      #fetching our account ID

for eachSnap in EC2_CLIENT_OBJECT.describe_snapshots(OwnerIds=[accountID])['Snapshots']:
    print(eachSnap)
```

- ➔ After copying the snapshots, we are changing the tagname from backup=yes to backup=complete to make sure that duplicate copies are not made at destination.
- ➔ The complete script is listed as "**CrossRegionsSnapshot.py**" in our scripts. Please go through the code.
- ➔ We need to deploy the same code to AWS Lambda now and then schedule the event using Cloud Watch Events at every weekday early morning 6 AM.
- ➔ Please refer our above sections, it has complete description on how to do this in AWS Lambda.
- ➔ If you experience any issue in execution of the scheduled job, then visit the cloudwatch logs that contains the detailed information on our current Lambda function.

**Listing our S3 buckets using BOTO3:**

- ➔ Firstly, let us create the custom session with root profile and then starting working on S3 with client objects.
- ➔ We can even work with resource objects, but here we prefer to work with S3 client objects, because we can make almost any task in client objects.
- ➔ We will be listing all the buckets using the "**list_buckets()**" function from s3 client object.

➔ The response we get from using the list_buckets is as follows:

**Response Syntax**

```
{
    'Buckets': [
        {
            'Name': 'string',
            'CreationDate': datetime(2015, 1, 1)
        },
    ],
    'Owner': {
        'DisplayName': 'string',
        'ID': 'string'
    }
}
```

➔ We need to iterate over the '**Buckets**' key to get the required information on buckets.
➔ Iterating over '**Owner**' key to get the user details.
➔ Our script for listing all buckets will be like this:

```
#function to list all the buckets using client object
def listBuckets():
    try:
        for eachBucket in S3_CLIENT_OBJECT.list_buckets()['Buckets']:   #iterating over 'Buckets' key
            print("Bucket Name : {}\tCreation Date: {}".format(
                eachBucket['Name'],
                eachBucket['CreationDate'].strftime('%Y-%m-%d'))
            )
    except Exception as e: print("Issue listing buckets: {}".format(e))
```

➔ Make sure to include the pagination concept to include all the buckets in your result.

**Paginators for S3:**

➔ Let's consider that we have more than 1000 objects in a bucket in our API result, we will need a paginator to get all the results.
➔ We will be listing the objects that are present in the S3 buckets using pagination.
➔ We need to create our bucket object to fetch the objects from the s3 bucket.
➔ We will be using the '**list_objects()**' function to list all the objects from the bucket.
➔ The script would be like this:

```
def listObjectsInBucket():
    try:
        BucketName = 'cf-templates-4u351hjm9psg-us-east-1'
        for eachObject in S3_CLIENT_OBJECT.list_objects(Bucket=BucketName)['Contents']:   #iterating over 'contents'
            print("Object Name: {}".format(eachObject['Key']))
    except Exception as e: print("Issue listing objects: {}".format(e))
```

➔ The code after adding the pagination will be like this:

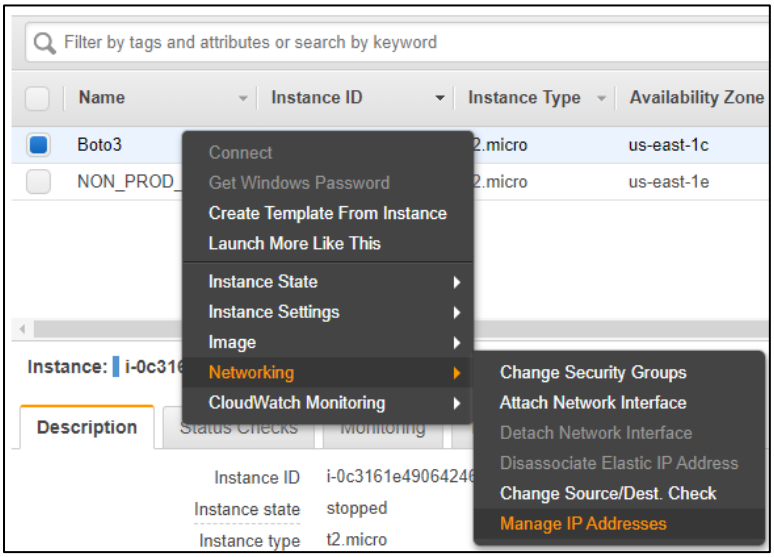
```
paginatorObject =  S3_CLIENT_OBJECT.get_paginator('list_objects')    #paginator object
```

```
def listObjectsInBucket():
    try:
        BucketName = 'cf-templates-4u351hjm9psg-us-east-1'
        for eachPage in paginatorObject.paginate(Bucket=BucketName):    #paginating over objects in a bucket
            for eachObject in eachPage['Contents']:   #iterating over 'contents' key
                print("Object Name: {}".format(eachObject['Key']))
    except Exception as e: print("Issue listing objects: {}".format(e))
```

➔ We can have 10 buckets under a region, so we will not be needing the paginators for listing the buckets. But we need paginator for listing the objects inside a bucket.

**<u>Automatically Assign Secondary IP of Master to Slave when Master is down:</u>**

➔ This is applicable where an instance acts like a master node and an instance acts like a slave node.
➔ Whenever the primary master node fails, we need to assign the secondary IP to the slave node.
➔ Navigate to the EC2 instances and then go to Manage IP address and assign a new IP to the Master server.




➔ Now select Assign IP address.


➔ Now select on Yes, Update.

➔ Now, if you refresh the console and navigate down, you can see the secondary IP address assigned to the instance.

➔ Let's do this over AWS Lambda now by creating a function from scratch.

➔ Fetch the master and slave Instance ID from EC2 console and also the secondary IP you assigned for the master node.

➔ We will be working on EC2 resource object for dealing this situation.

➔ We will create the master server object by using **Instance()** function and passing the master server instance ID as parameter.

➔ Now, we will check if master server is running or not, if it is running then we will not do anything.

➔ If master server is not running, then we will perform the assignment of IP to slave server.

➔ Now for else case, create the secondary server object.

➔ Now fetch the primary and secondary server network interface information.

➔ Now fetch the primary and secondary server network interface ID.

➔ Now we will be creating the EC2 client object.

➔ Now we will unassign the private IP (Secondary IP) for master node and assign it to slave node.

➔ The final script would be like this:

```
import json
import boto3

def lambda_handler(event, context):
    masterNode = "i-0c3161e49064246a9"
    slaveNode = "i-07b9bf30e368d6cab"
    secondaryIP = "172.31.89.168"

    ec2Object = boto3.resource('ec2','us-east-1')   #Resource object
    masterServer = ec2Object.Instance(masterNode)   #Master instance object

    if masterServer.state['Name']=="running":   #check if master is down
        print("Master server is up and running..")

    else:
        slaveServer = ec2Object.Instance(slaveNode) #slave server object
        masterNetInterface = masterServer.network_interfaces_attribute[0]
        slaveNetInterface = slaveServer.network_interfaces_attribute[0]
        masterNetIntID = masterNetInterface['NetworkInterfaceId']
        slaveNetIntID = slaveNetInterface['NetworkInterfaceId']
        ec2ClientObject = boto3.client('ec2','us-east-1')   #client object
        ec2ClientObject.unassign_private_ip_addresses(
            NetworkInterfaceId=masterNetIntID,   #master network interface ID
            PrivateIpAddresses = [secondaryIP]  #Secondary IP of master
            )
        ec2ClientObject.assign_private_ip_addresses(
            AllowReassignment = True,
            NetworkInterfaceId = slaveNetIntID,  #slave network interface ID
            PrivateIpAddresses = [secondaryIP]  #secondary IP of master to slave
            )
    return {
        'statusCode': 200,
        'body': json.dumps('Program executed successfully!')
    }
```

➔ You can also refer to this script under '**failOverScript.py**'

➔ Now if we stop the master server, and then start the script then we see that our secondary IP gets assigned to slave server.

➔ Make sure that IP address of Master and slave fall under same Subnet that is for
Eg: if master = '172.20.20.100' then slave must be '172.20.20.xxx' where xxx be any number for slave.

➔ If they are not in same subnet, we cannot assign the IP address to slave.

➔ We can automate this code by going to CloudWatch and then selecting events and triggering this action when instance state of an instance ID is changed to stopped state like this:


➔ On the next screen, we need to choose the Lambda function that needs to get triggered for this rule and hit ok create rule.

➔ That should schedule this rule when our master server goes to stopped state.