Scrum Documentation

SkipQ DevOps – Proxima Centauri

Author:

Abdullah Zaman Babar abdullah.zaman.babar.s@skipq.org

Synopsis:

This document is a guide to the techniques and procedures followed for the conclusion of the DevOps training offered at SkipQ.
For the construction and operation of production-grade web applications running across numerous regions, Infrastructure as Code is made use of. RESTful Python Apis are employed on AWS to monitor CloudWatch metrics, along with the application of Continuous Integration and Continuous Deployment for automation.

Table of Contents

1. Orientation Session

1.1. Introduction to Cloud Computing

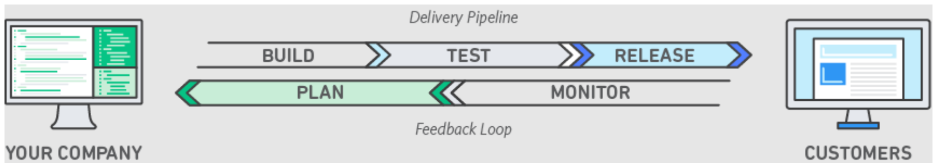
Cloud computing is the delivery of diverse services through the Internet. It is the on-demand availability of computer system resources. It uses the internet for storing and managing data on remote servers and then access data via the internet.

1.2. Introduction to AWS

AWS (Amazon Web Services) is an extensive and advancing distributed computing platform introduced by Amazon that incorporates a combination of infrastructure as a service (IaaS), platform as a service (PaaS) and packaged software as a service (SaaS).

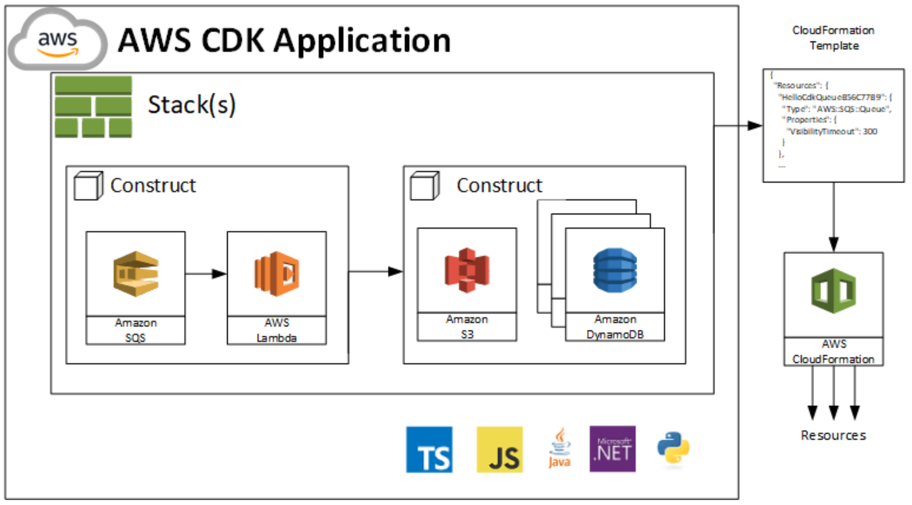
1.3. Introduction to DevOps

DevOps represents a change in IT culture, focusing on rapid IT service delivery through the adoption of agile, lean practices in the context of a system-oriented approach.



1.4. Introduction to AWS CDK

The AWS CDK is a new software development framework from AWS with the sole purpose of making it fun and easy to define cloud infrastructure in our favorite programming language and deploy it using AWS CloudFormation.



```
{
  "Resources": {
    "HelloCdkQueueB56C7789": {
      "Type": "AWS::SQS::Queue",
      "Properties": {
        "VisibilityTimeout": 300
      }
    },
    ...
```

2. Sprint 1

2.1. Objective

Using AWS CDK to measure the availability and latency(delay) of a custom list of websites and monitor the results on a CloudWatch. Then setting up alarms on metrics when the prescribed thresholds are breached. Each alarm is published to SNS notifications which triggers a lambda function that writes the alarm information into DynamoDB.

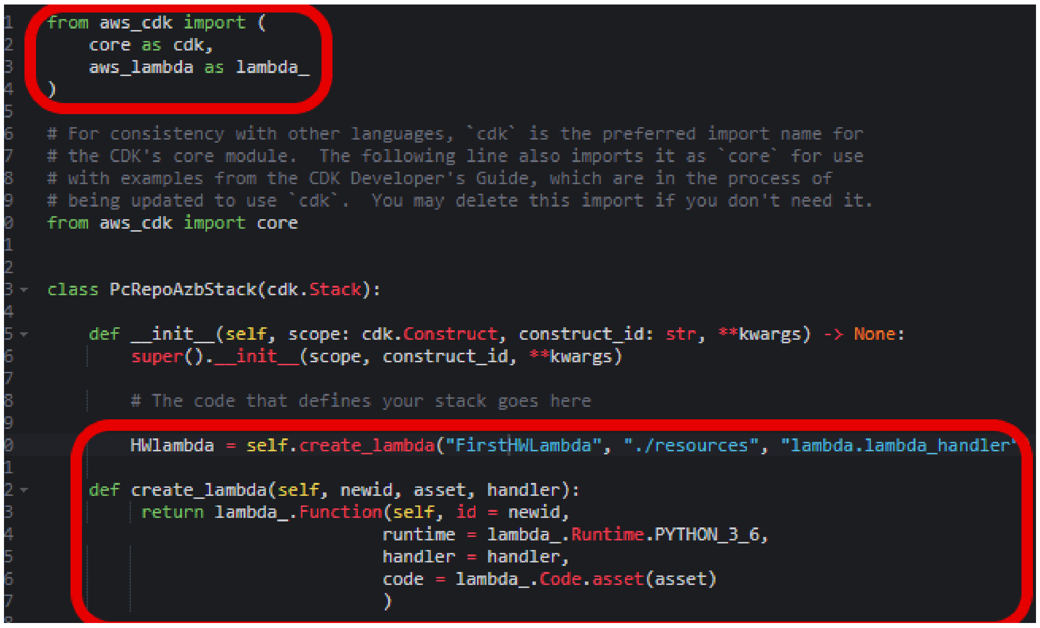
2.2. Implementation

Creating the Hello World Lambda Function:

AWS Lambda is a serverless compute service that runs your code in response to events and automatically manages the underlying compute resources for you.

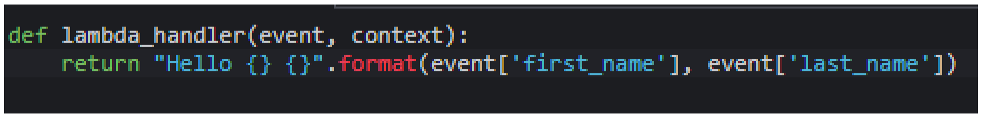
Against EC2, we now have Lambda to deploy or scale applications. It is a virtual function; it uploads and scales. It doesn't need to be provisioned, it only needs to provision code which is a function that will run on the cloud, and will auto provision resources file.

We are using the *Function* method of Lambda library. It defines the lambda functions.



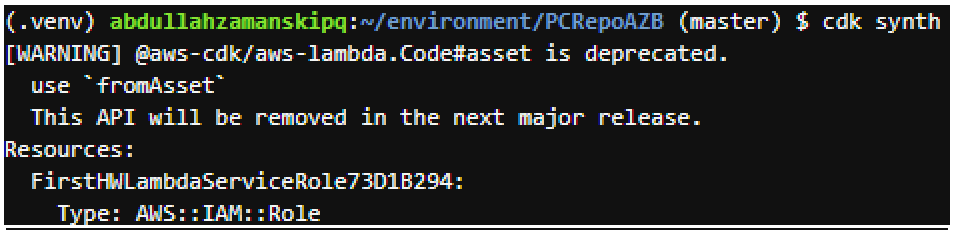
```python
from aws_cdk import (
    core as cdk,
    aws_lambda as lambda_
)

# For consistency with other languages, `cdk` is the preferred import name for
# the CDK's core module.  The following line also imports it as `core` for use
# with examples from the CDK Developer's Guide, which are in the process of
# being updated to use `cdk`.  You may delete this import if you don't need it.
from aws_cdk import core


class PcRepoAzbStack(cdk.Stack):

    def __init__(self, scope: cdk.Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # The code that defines your stack goes here

        HWlambda = self.create_lambda("FirstHWLambda", "./resources", "lambda.lambda_handler"

    def create_lambda(self, newid, asset, handler):
        return lambda_.Function(self, id = newid,
                                runtime = lambda_.Runtime.PYTHON_3_6,
                                handler = handler,
                                code = lambda_.Code.asset(asset)
                                )
```

Then we define our lambda handler that takes an *event* and *context* as parameters.

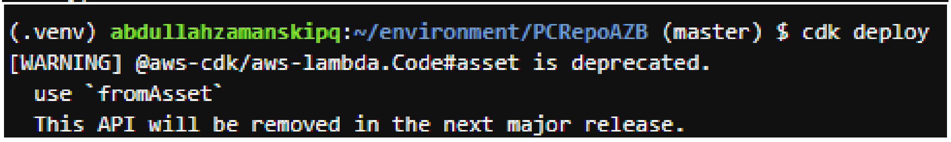


```python
def lambda_handler(event, context):
    return "Hello {} {}".format(event['first_name'], event['last_name'])
```

Now we use the command *cdk synth* to synthesize our code into CloudFormation template and they we use *cdk deploy* to deploy it in our S3 bucket, where an intermediary resource will run our code.

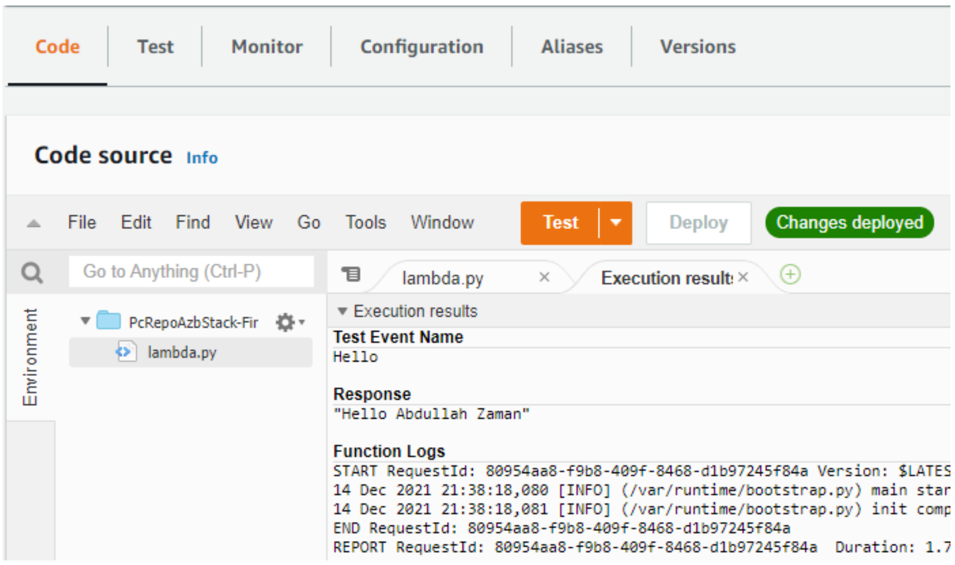


```
(.venv) abdullahzamanskipq:~/environment/PCRepoAZB (master) $ cdk synth
[WARNING] @aws-cdk/aws-lambda.Code#asset is deprecated.
  use `fromAsset`
  This API will be removed in the next major release.
Resources:
  FirstHWLambdaServiceRole73D1B294:
    Type: AWS::IAM::Role
```



```
(.venv) abdullahzamanskipq:~/environment/PCRepoAZB (master) $ cdk deploy
[WARNING] @aws-cdk/aws-lambda.Code#asset is deprecated.
  use `fromAsset`
  This API will be removed in the next major release.
```

Now we test our code. The *code source* is our compute resource that runs our code from s3 bucket.



Test Event Name
```
Hello
```

Response
```
"Hello Abdullah Zaman"
```

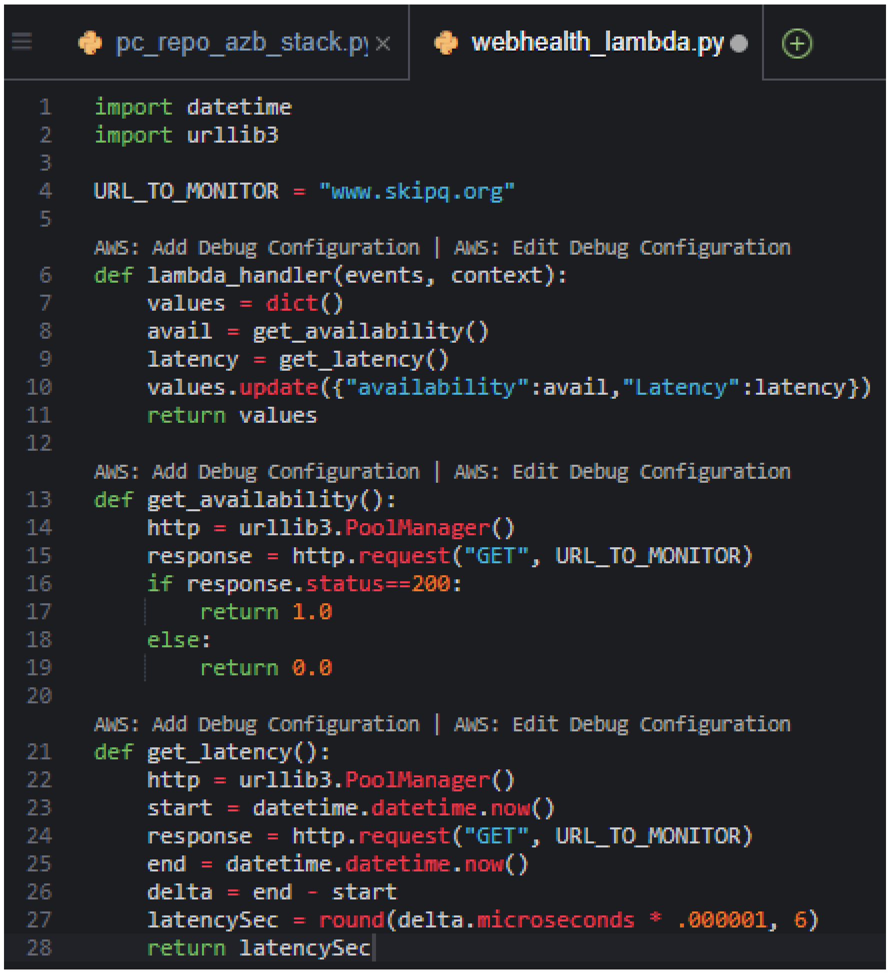
Function Logs
```
START RequestId: 80954aa8-f9b8-409f-8468-d1b97245f84a Version: $LATES
14 Dec 2021 21:38:18,080 [INFO] (/var/runtime/bootstrap.py) main star
14 Dec 2021 21:38:18,081 [INFO] (/var/runtime/bootstrap.py) init comp
END RequestId: 80954aa8-f9b8-409f-8468-d1b97245f84a
REPORT RequestId: 80954aa8-f9b8-409f-8468-d1b97245f84a  Duration: 1.7
```

Creating the WebHealth Lambda Function:

In this we are working to achieve two metrics of a url that will determine its availability and latency.

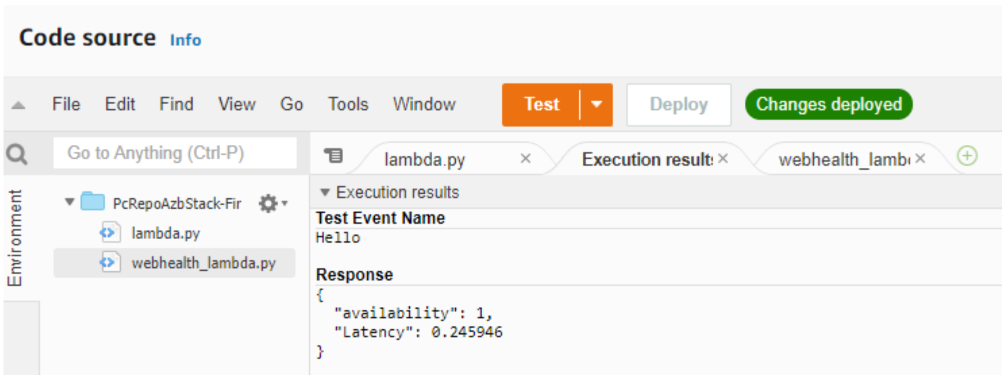
We are using the *urllib3* library that will catch the status of our url and will determine the availability metric.

The latency metric is determined by using the *datetime* module through which we measure the different of the time the url is called and the response we receive.



```
import datetime
import urllib3

URL_TO_MONITOR = "www.skipq.org"

AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def lambda_handler(events, context):
    values = dict()
    avail = get_availability()
    latency = get_latency()
    values.update({"availability":avail,"Latency":latency})
    return values

AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def get_availability():
    http = urllib3.PoolManager()
    response = http.request("GET", URL_TO_MONITOR)
    if response.status==200:
        return 1.0
    else:
        return 0.0

AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def get_latency():
    http = urllib3.PoolManager()
    start = datetime.datetime.now()
    response = http.request("GET", URL_TO_MONITOR)
    end = datetime.datetime.now()
    delta = end - start
    latencySec = round(delta.microseconds * .000001, 6)
    return latencySec
```

We can see the values returned for our metrics in *code source.*



Test Event Name
```
Hello
```

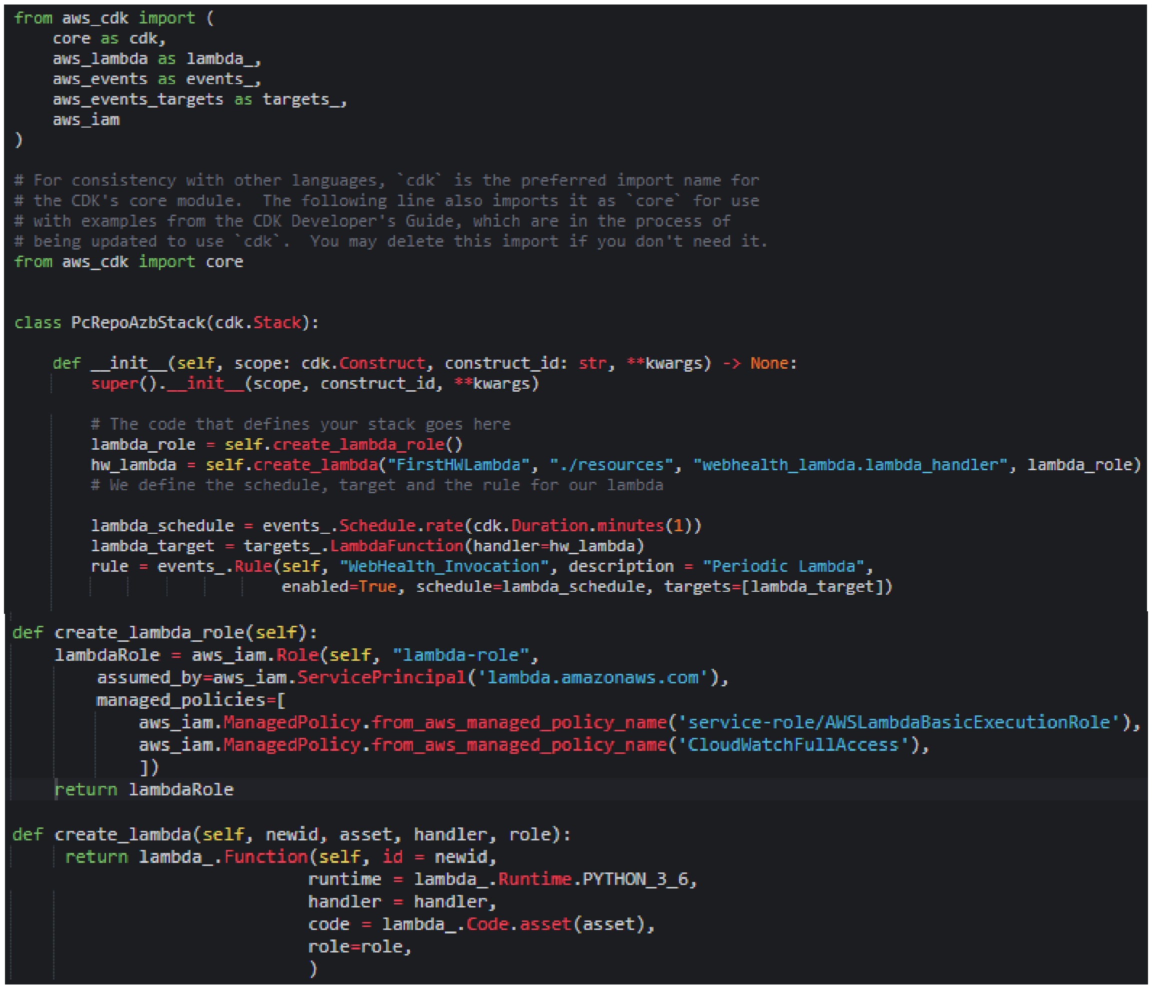
Response
```
{
  "availability": 1,
  "Latency": 0.245946
}
```

Periodically monitoring the Metrics:

Currently our WebHealth lambda function is currently a one-time invocation, but being a DevOps engineer we want to periodically monitor our metrics and we want to schedule it to run every one minute.

Amazon EventBridge delivers a near real-time stream of system events that describe changes in AWS resources.

First, we have to schedule an *event*, and then will specify a *target* (i.e., lambda in our case) for our event. Then we specify a *rule* for applying the schedule to the target. We want our lambda to be invoked periodically to show us the availability and latency of our url in order to visualize our metrics graphically.

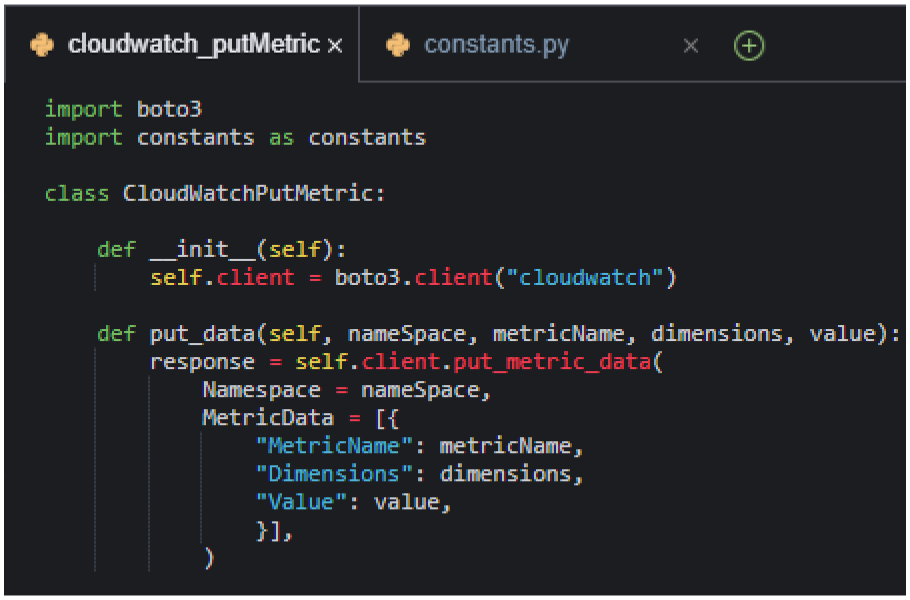


```
from aws_cdk import (
    core as cdk,
    aws_lambda as lambda_,
    aws_events as events_,
    aws_events_targets as targets_,
    aws_iam
)

# For consistency with other languages, `cdk` is the preferred import name for
# the CDK's core module.  The following line also imports it as `core` for use
# with examples from the CDK Developer's Guide, which are in the process of
# being updated to use `cdk`.  You may delete this import if you don't need it.
from aws_cdk import core


class PcRepoAzbStack(cdk.Stack):

    def __init__(self, scope: cdk.Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # The code that defines your stack goes here
        lambda_role = self.create_lambda_role()
        hw_lambda = self.create_lambda("FirstHWLambda", "./resources", "webhealth_lambda.lambda_handler", lambda_role)
        # We define the schedule, target and the rule for our lambda

        lambda_schedule = events_.Schedule.rate(cdk.Duration.minutes(1))
        lambda_target = targets_.LambdaFunction(handler=hw_lambda)
        rule = events_.Rule(self, "WebHealth_Invocation", description = "Periodic Lambda",
                            enabled=True, schedule=lambda_schedule, targets=[lambda_target])

def create_lambda_role(self):
    lambdaRole = aws_iam.Role(self, "lambda-role",
        assumed_by=aws_iam.ServicePrincipal('lambda.amazonaws.com'),
        managed_policies=[
            aws_iam.ManagedPolicy.from_aws_managed_policy_name('service-role/AWSLambdaBasicExecutionRole'),
            aws_iam.ManagedPolicy.from_aws_managed_policy_name('CloudWatchFullAccess'),
            ])
    return lambdaRole

def create_lambda(self, newid, asset, handler, role):
    return lambda_.Function(self, id = newid,
                            runtime = lambda_.Runtime.PYTHON_3_6,
                            handler = handler,
                            code = lambda_.Code.asset(asset),
                            role=role,
                            )
```

Publishing Metrics on CloudWatch:

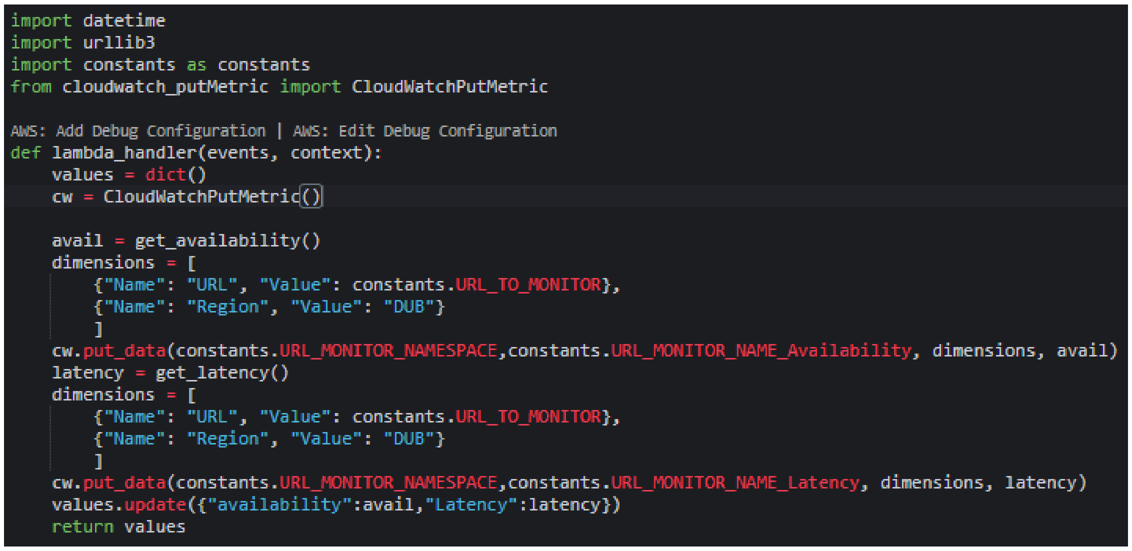
CloudWatch is a monitoring service, where we can use functions that will help up set alarms and we can set metrics e.g., availability and latency. We can also use cloudwatch to graphically monitor our metrics and other things

We use *boto3* module to use its *CloudWatch client.*



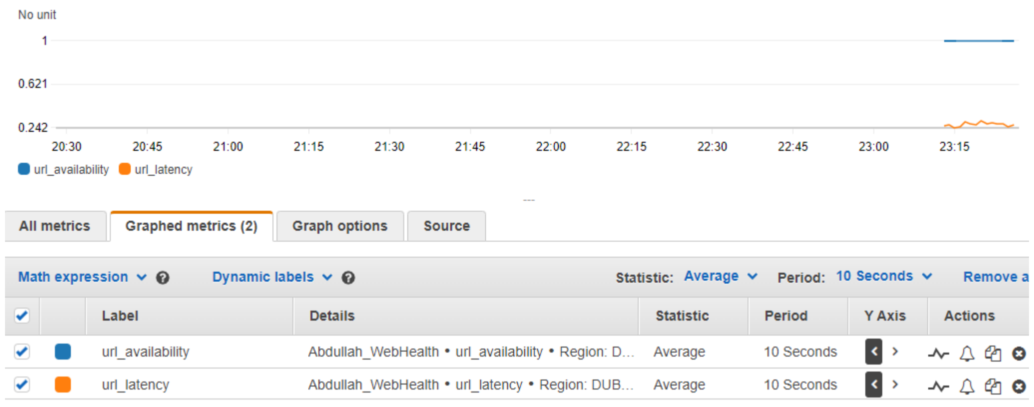
```
import boto3
import constants as constants

class CloudWatchPutMetric:

    def __init__(self):
        self.client = boto3.client("cloudwatch")

    def put_data(self, nameSpace, metricName, dimensions, value):
        response = self.client.put_metric_data(
            Namespace = nameSpace,
            MetricData = [{
                "MetricName": metricName,
                "Dimensions": dimensions,
                "Value": value,
                }],
            )
```

In the project stack we have an event that will schedule every one minute and every one minute it will call the *webhealth_lambda* function. Now every time *webhealth_lambda* function is called we want to take the metrics and put them on CloudWatch.



```
import datetime
import urllib3
import constants as constants
from cloudwatch_putMetric import CloudWatchPutMetric

AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def lambda_handler(events, context):
    values = dict()
    cw = CloudWatchPutMetric()

    avail = get_availability()
    dimensions = [
        {"Name": "URL", "Value": constants.URL_TO_MONITOR},
        {"Name": "Region", "Value": "DUB"}
        ]
    cw.put_data(constants.URL_MONITOR_NAMESPACE,constants.URL_MONITOR_NAME_Availability, dimensions, avail)
    latency = get_latency()
    dimensions = [
        {"Name": "URL", "Value": constants.URL_TO_MONITOR},
        {"Name": "Region", "Value": "DUB"}
        ]
    cw.put_data(constants.URL_MONITOR_NAMESPACE,constants.URL_MONITOR_NAME_Latency, dimensions, latency)
    values.update({"availability":avail,"Latency":latency})
    return values
```

The graphical results can be seen as follows in the CloudWatch console.



		Label	Details	Statistic	Period	Y Axis	Actions
✓	🟦	url_availability	Abdullah_WebHealth • url_availability • Region: D...	Average	10 Seconds		
✓	🟧	url_latency	Abdullah_WebHealth • url_latency • Region: DUB...	Average	10 Seconds		

Setting up Alarms:

In our project stack file, we want to directly use CloudWatch, so the metrics we used are able to be imported in our infrastructure.

So, when we invoke lambda, the metrics defined in are automatically published on CloudWatch, but to set alarms on these metrics we need to have these metrics in our infrastructure in stack file.

We can call *Metric* on an existing metric which will be used in our infrastructure to set an alarm on top of it. We can't set an alarm on lambda it has to be done in our infrastructure. Lambda gets invoked every one minute, so every minute our alarm will be created which we do not want. We want our alarm to be continuous so every one minute the metric gets updated, and if the metric goes up the specified threshold alarm will be raised.

The infrastructure that we are setting is done in stack, lambda is the function that gets invoked. So, first the stack will set up and then lambda will be invoked. All the setting is done in stack.

We define our metric and then set an alarm on it in our stack file.

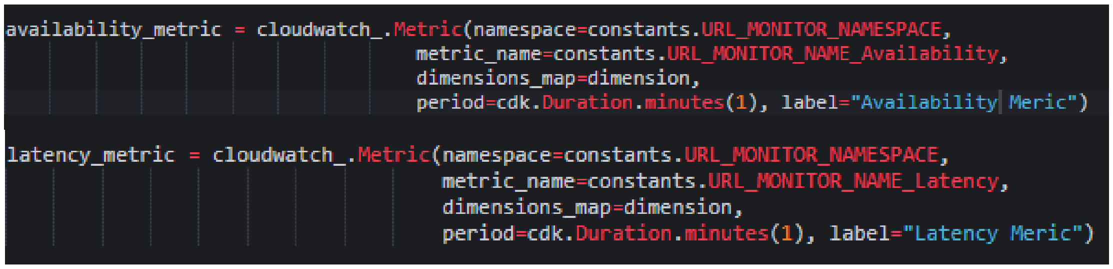
```
cloudwatch_.Metric()
```

```
cw.put_data(constants.URL_MONITOR_NAMESPACE,constants.URL_MONITOR_NAME_Availability, dimensions, avail)
cw.put_data(constants.URL_MONITOR_NAMESPACE,constants.URL_MONITOR_NAME_Latency, dimensions, latency)
```

We set up our dimensions as.

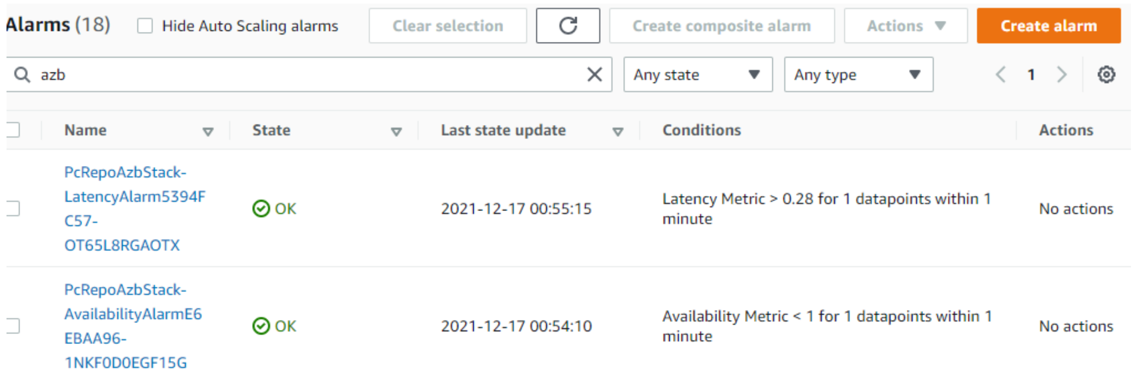
```
dimension = {"URL" : constants.URL_TO_MONITOR}
```

We then create the metrics.



```
availability_metric = cloudwatch_.Metric(namespace=constants.URL_MONITOR_NAMESPACE,
                                         metric_name=constants.URL_MONITOR_NAME_Availability,
                                         dimensions_map=dimension,
                                         period=cdk.Duration.minutes(1), label="Availability Meric")

latency_metric = cloudwatch_.Metric(namespace=constants.URL_MONITOR_NAMESPACE,
                                         metric_name=constants.URL_MONITOR_NAME_Latency,
                                         dimensions_map=dimension,
                                         period=cdk.Duration.minutes(1), label="Latency Meric")
```

We can see the results in CloudWatch console.



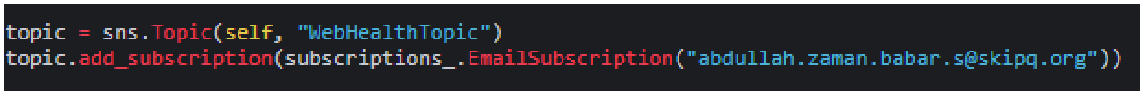
Name	State	Last state update	Conditions	Actions
PcRepoAzbStack-LatencyAlarm5394FC57-OT65L8RGAOTX	OK	2021-12-17 00:55:15	Latency Metric > 0.28 for 1 datapoints within 1 minute	No actions
PcRepoAzbStack-AvailabilityAlarmE6EBAA96-1NKFODOEGF15G	OK	2021-12-17 00:54:10	Availability Metric < 1 for 1 datapoints within 1 minute	No actions

SNS Topic – Email Subscription:

SNS is the *simple notification service* and it notifies the subscriber of a breach in threshold.

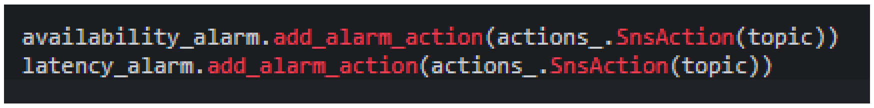
We will first set the *sns* and then we will add *sns-subscriptions* on it to subscribe to the sns topic.

We define a topic with *id* and the *scope* as parameters.



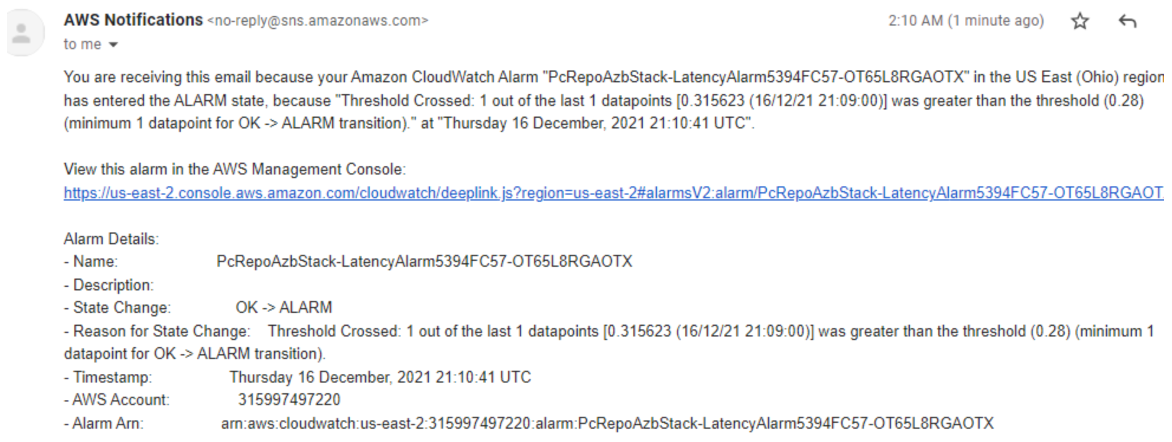
```
topic = sns.Topic(self, "WebHealthTopic")
topic.add_subscription(subscriptions_.EmailSubscription("abdullah.zaman.babar.s@skipq.org"))
```

We then add subscription to our alarms that we specified. These should be under the alarms defined.



```
availability_alarm.add_alarm_action(actions_.SnsAction(topic))
latency_alarm.add_alarm_action(actions_.SnsAction(topic))
```

We receive the alarm notification in our email.


AWS Notifications <no-reply@sns.amazonaws.com> 2:10 AM (1 minute ago)
to me

You are receiving this email because your Amazon CloudWatch Alarm "PcRepoAzbStack-LatencyAlarm5394FC57-OT65L8RGAOTX" in the US East (Ohio) region has entered the ALARM state, because "Threshold Crossed: 1 out of the last 1 datapoints [0.315623 (16/12/21 21:09:00)] was greater than the threshold (0.28) (minimum 1 datapoint for OK -> ALARM transition)." at "Thursday 16 December, 2021 21:10:41 UTC".

View this alarm in the AWS Management Console:
https://us-east-2.console.aws.amazon.com/cloudwatch/deeplink.js?region=us-east-2#alarmsV2:alarm/PcRepoAzbStack-LatencyAlarm5394FC57-OT65L8RGAOT

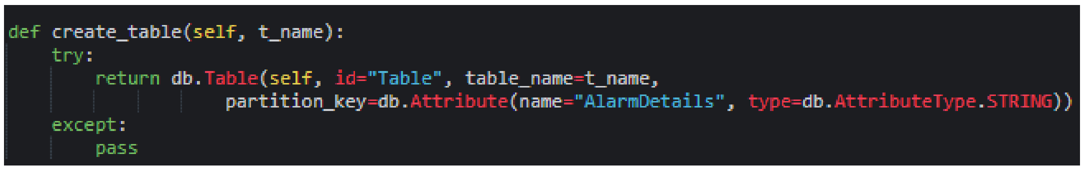
Alarm Details:
- Name: PcRepoAzbStack-LatencyAlarm5394FC57-OT65L8RGAOTX
- Description:
- State Change: OK -> ALARM
- Reason for State Change: Threshold Crossed: 1 out of the last 1 datapoints [0.315623 (16/12/21 21:09:00)] was greater than the threshold (0.28) (minimum 1 datapoint for OK -> ALARM transition).
- Timestamp: Thursday 16 December, 2021 21:10:41 UTC
- AWS Account: 315997497220
- Alarm Arn: arn:aws:cloudwatch:us-east-2:315997497220:alarm:PcRepoAzbStack-LatencyAlarm5394FC57-OT65L8RGAOTX

Lambda Subscription:

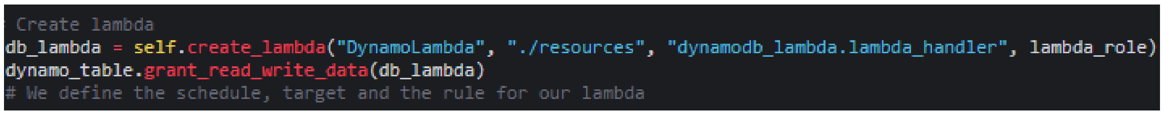
We want that when we get notified, we want another lambda function to be notified which happens when an alarm is raised.

1. We need to add lambda as a subscriber to the sns topic, and a sns notification which is an event goes to the lambda.
2. When the lambda is triggered, it takes the cloudwatch alarm information as an event.
3. Now parse the alarm event, and write it in DynamoDB. Every time an alarm is raised add a row to the table

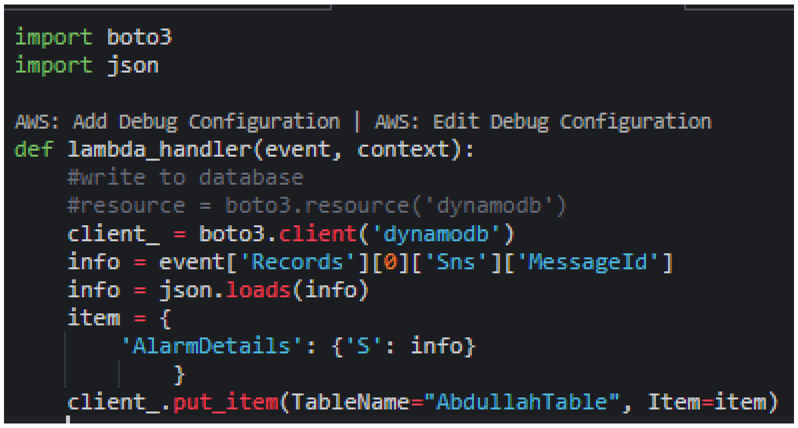
First, we create a table in our DynamoDB and make sure that once the table in created it doesn't not try to overwrite it.


```python
def create_table(self, t_name):
    try:
        return db.Table(self, id="Table", table_name=t_name,
                    partition_key=db.Attribute(name="AlarmDetails", type=db.AttributeType.STRING))
    except:
        pass
```

We the create a lambda function for our database.

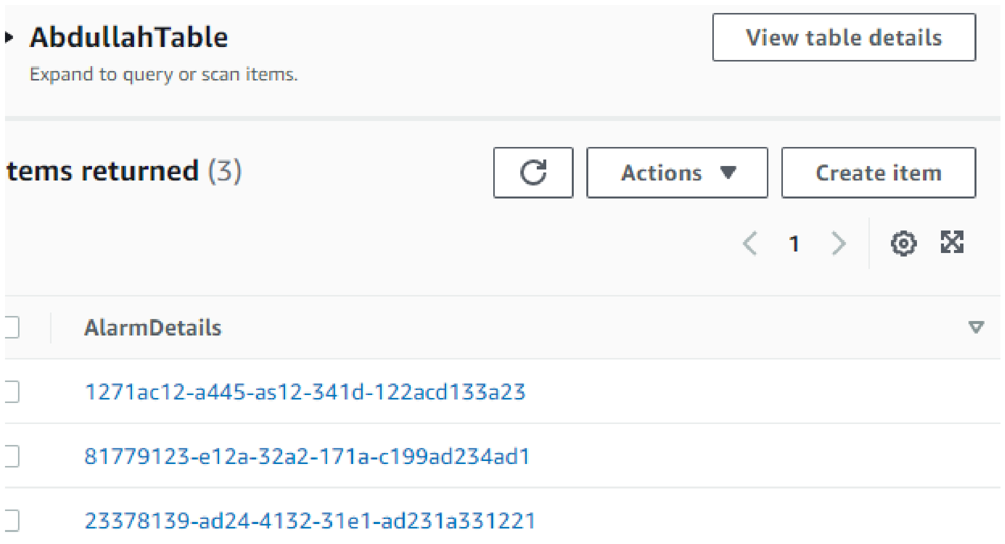

```python
 Create lambda
db_lambda = self.create_lambda("DynamoLambda", "./resources", "dynamodb_lambda.lambda_handler", lambda_role)
dynamo_table.grant_read_write_data(db_lambda)
# We define the schedule, target and the rule for our lambda
```

Then we create a function to parse through the event.



```
import boto3
import json

AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def lambda_handler(event, context):
    #write to database
    #resource = boto3.resource('dynamodb')
    client_ = boto3.client('dynamodb')
    info = event['Records'][0]['Sns']['MessageId']
    info = json.loads(info)
    item = {
        'AlarmDetails': {'S': info}
            }
    client_.put_item(TableName="AbdullahTable", Item=item)
```

The entries recorded are as follows.



AlarmDetails
1271ac12-a445-as12-341d-122acd133a23
81779123-e12a-32a2-171a-c199ad234ad1
23378139-ad24-4132-31e1-ad231a331221