Nimra Amin
skipQ trainee (cohort 2)
Batch: Proxima Centauri
Email: nimra.amin.s@skipq.org

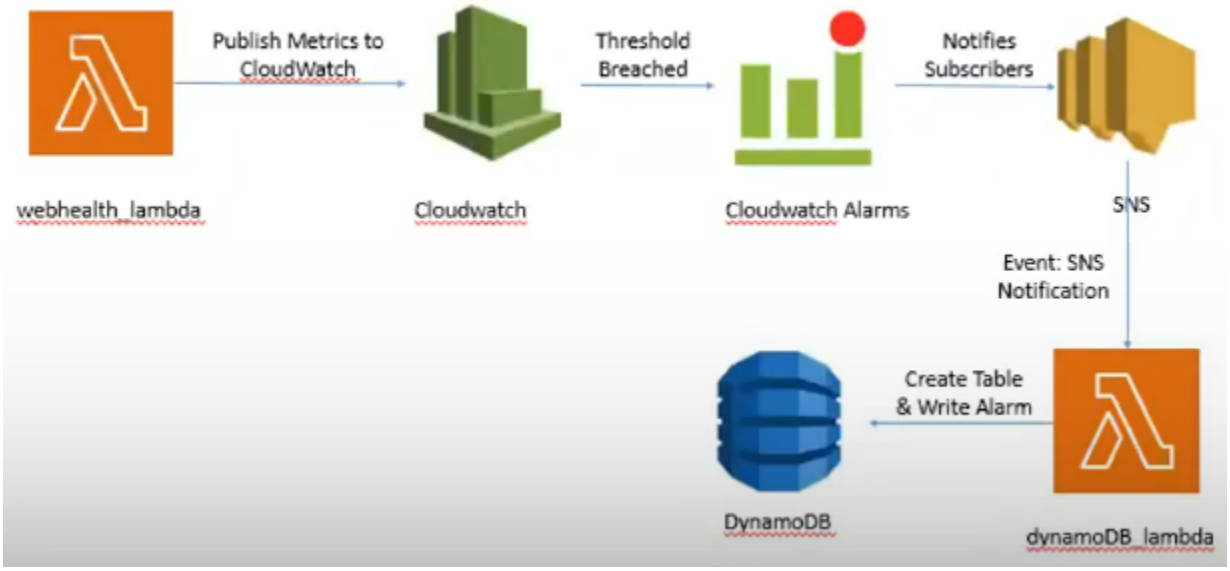
SPRINT 1: A Web Health Monitoring Application

Name: Nimra Amin
Demo Date: 20th Dec 2021

SPRINT 1: A Web Health Monitoring Application

20th December 2021

OVERVIEW:



GOALS

1. Create a periodic Lambda Function, that checks latency and availability of a specified URL at some predefined periodic time.
2. Connect that periodic lambda function with cloud watch, by publishing the latency and availability metric to CloudWatch.
3. Generate an alarm, if the CloudWatch's metric breaches a certain threshold.
4. Create an SNS Topic that notifies(emails) its subscribers of the breached threshold by sending them an alarm message.
5. Create A table in DynamoDB for keeping a record of the alarm messages.
6. Create another SNS subscription that triggers a lambda function to write into DynamoDB whenever an alarm is triggered.
7. Now, run this application for 4 URLs, by retrieving them from an S3 Bucket.

PROJECT SPECIFICATIONS:

- Aws-cdk (v2)
- Python 3.6
- Git for vcs

```
aws-cdk.core==1.135.0
aws-cdk.assertions==1.135.0
aws-cdk.aws.lambda==1.135.0
aws-cdk.core==1.135.0
aws-cdk.aws_events_targets==1.135.0
aws-cdk.aws_cloudwatch_actions==1.135.0
aws-cdk.aws_dynamodb==1.135.0
```

MILESTONES:

1. Periodic LAMBDA:

Create a lambda function that is invoked periodically, and calculates the latency and availability of a given URL. The following code has been used to create a periodic lambda function.

```
lambda_role = self.create_lambda_role()

# THis is a periodic lambda function that monitors webhealth
WH_lambda = self.create_lambda("WebHealthPeriodicLambda",
                               "./resources/",
                               "webHealth_lambda.lambda_handler",
                               lambda_role)
```

```
lambda_schedule = _events.Schedule.rate(cdk.Duration.minutes(1))
lambda_targets = _events_targets.LambdaFunction(handler=WH_lambda)
rule = _events.Rule(self, "webHealth_Invocation",
                    description="Periodic Lambda",
                    enabled=True,
                    schedule=lambda_schedule,
                    targets=[lambda_targets])
```

2. Publish Metrics to CloudWatch:

This stage publishes to the CloudWatch. Where, the metrics can be monitored in real time. The following is the code:

```
#  for puclishing metrics to cloued watch
dimensions = {'URL': constants.URL_TO_MONITOR}
availability_metric = _cloudwatch.Metric(namespace = constants.URL_MONITOR_NAMESPACE,
                metric_name=constants.URL_MONITOR_NAME_AVAILABILITY,
                dimensions_map = dimensions,
                period = cdk.Duration.minutes(1),
                label = 'AVAILABILITY ALARM METRIC')
```

```
#  for puclishing aws metrics to cloued watch

dimensions = {'URL': constants.URL_TO_MONITOR}
latency_metric = _cloudwatch.Metric(namespace = constants.URL_MONITOR_NAMESPACE,
                metric_name=constants.URL_MONITOR_NAME_LATENCY,
                dimensions_map = dimensions,
                period = cdk.Duration.minutes(1),
                label = 'LATENCY ALARM METRIC')
```

3. Publish Metrics to CloudWatch:

Generate an alarm, whenever the specified threshold is reached for both the latency and availability metrics. For latency, we have kept the threshold's range between .25-.3, where 0 is the threshold for the availability metric.

```
#Setting up the availability alarm
availability_alarm = _cloudwatch.Alarm(self,
                                id ='NimraAvailabilityAlarm',
                                metric = availability_metric,
                                comparison_operator =_cloudwatch.ComparisonOperator.LESS_THAN_THRESHOLD,
                                datapoints_to_alarm = 1,
                                evaluation_periods = 1,
                                threshold = 1,
                                )
```

```
#Setting up the availability alarm
latency_alarm = _cloudwatch.Alarm(self,
                                id ='NimraLatencyAlarm',
                                metric = latency_metric,
                                comparison_operator =_cloudwatch.ComparisonOperator.GREATER_THAN_THRESHOLD,
                                datapoints_to_alarm = 1,
                                evaluation_periods = 1,
                                threshold = .25 #.34
                                )
```

4. SNS Topic with Email Subscription:

Create an SNS topic, which is triggered whenever an email is breached. It sends a message to its subscriber.

```
#SNS TOPIC
topic = sns.Topic(self, "Nimra_webHealthAlarm_sprint1")
topic.add_subscription(subscriptions.EmailSubscription(
                                        email_address = "nimra.amin.s@skipq.org"))

topic.add_subscription(subscriptions.LambdaSubscription(
                                        fn=ddb_lambda_producer))
```

5. DynamoDB Table:

Create a DynamoDB table in the stack, that store alarm notifications in its table. Provide to it the correct read and write permissions with correct roles and policies, for it to be able to work.

```
def create_ddb_table(self, ):
    return ddb.Table(self,
                     id="Nimra_Table",
                     table_name=constants.TABLE_NAME,
                     partition_key=ddb.Attribute(name="MessageID", type=ddb.AttributeType.STRING))
```

```
#### CREATING A DYNAMODB TABLE
 #create table in dynamo db
try:
    ddb_alarm_table= self.create_ddb_table()
except: pass

#give read write permissions to our lambda
ddb_alarm_table.grant_read_write_data(ddb_lambda_producer)
###defining SNS service
ddb_lambda_producer.add_environment('table_name',constants.TABLE_NAME)
```

```
  aws_iam.ManagedPolicy.from_aws_managed_policy_name('AmazonDynamoDBFullAccess'),
```

6. SNS Triggered Lambda:

By Subscribing the SNS topic to the DynamoDB lambda, we can make it able to read and write into the DynamoDB table. A lambda is function is created which is then subscribed to the SNS topic.

```
#SNS TOPIC
topic = sns.Topic(self, "Nimra_webHealthAlarm_sprint1")
topic.add_subscription(subscriptions.EmailSubscription(
                                        email_address = "nimra.amin.s@skipq.org"))

topic.add_subscription(subscriptions.LambdaSubscription(
                                        fn=ddb_lambda_producer))
```

```
#### CREATING A DYNAMODB TABLE
 #create table in dynamo db
try:
    ddb_alarm_table= self.create_ddb_table()
except: pass

#give read write permissions to our lambda
ddb_alarm_table.grant_read_write_data(ddb_lambda_producer)
###defining SNS service
ddb_lambda_producer.add_environment('table_name',constants.TABLE_NAME)
```

a. The lambda handler:

```
import boto3,os
AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def lambda_handler(event, context):
    db=boto3.client('dynamodb')
    MessageID = event['Records'][0]['Sns']['Message']
    Timestamp = event['Records'][0]['Sns']['Timestamp']
    table_name=os.getenv('table_name')
    db.put_item(TableName=table_name,Item={
        'MessageID':{'S':MessageID},
        'TimeStamp':{'S':Timestamp},
    })
```

7. S3 Bucket:l

TO DO.

ISSUES:

1. No log groups for DynamoDB lambda:

- Was getting unpredictable behavior with incomplete resources, no log groups.
    - Need to manually delete the stack.

2. Lambda not being triggered through SNS:

SNS wasn't triggering the lambda. Due to which I was getting no log groups unless i pass the test case myself.

- Correctly specify the SNS roles and policies.
- Add a composite principal role, the lambda function.

3. DynamoDB Data Entry Issue:

Only test time deployment data was being entered into the table, not the alarm triggered data.

- Provide both the read write data execution permissions.
- Add environment for the table to get the table_name.

BACKLOG/TODO:

- Create an S3 Bucket resource, and retrieve URLs from it.