NATIONAL UNIVERSITY OF SINGAPORE

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EE5111 Special Topics in Industrial Control and Automation Systems

Simulation using Jet Engine Data to implement an Internet of Things (IoT) Setup using Amazon Web Services (AWS)

By

Name: PENG JINGMING (A0195042J)

Submission Date: 21 Sept 2019

Outline

1. Project Description
2. AWS IoT Concept
3. AWS IoT Realization
4. Visualization Concept
5. Visualization Realization
6. Task1– Publish Engine Data into DynamoDB through laptop and do visualization
7. Task 2 – Publish rainfall data through raspberry pi to DynamoDB and do visualization
8. Task3 – Self-monitor raspberry pi's cpu temperature and establish email alert on abnormal high temperature (temperature > 47.00DegC)
9. Discussion
10. Appendixes

1. Project Description

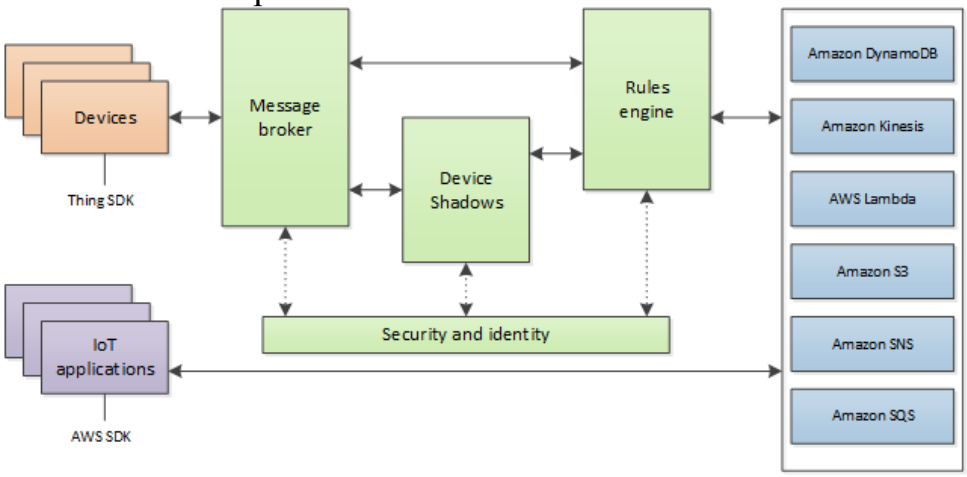
1.1) To implement a simple IoT pipeline with AWS Cloud platform and visualize the data.

1.2) To write a guideline on how to set up the pipeline with pictures showing matric No. inside.

1.3) 3 IoT tasks are discussed and covered within this report:

- Based on a laptop - the endpoint device platform, to simulate 2 small IoT setups that record and push data from 2 jet engines.
- To expand the idea, based on a Raspberry Pi 3 model B board - a different endpoint device platform, to simulate 1 IoT setup that pushes environmental data downloaded from Data.gov into AWS cloud.
- Relying on the same Raspberry Pi platform, capture its self-monitoring real-time data (temperature) and push it up to IoT cloud, and send pre-defined email alert on any abnormality (overtemperature).

2. AWS IoT Concept



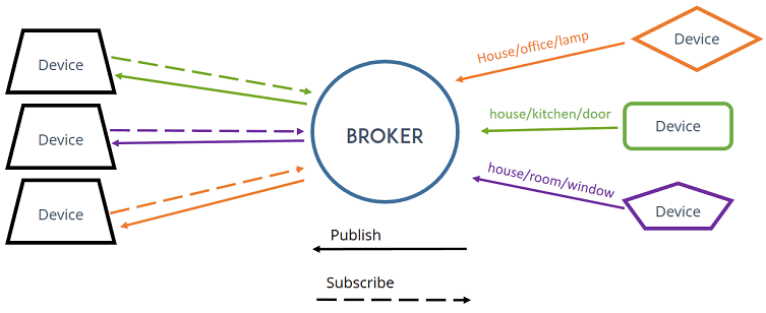
Within a general architecture of AWS IoT (shown above), it enables endpoint devices to connect to the AWS Cloud and lets applications in the cloud interact with these devices.

Main components:

2.1) Devices with policies - All devices that connect to AWS IoT need to have an entry in the registry, where information about a device and the certificates are stored. Within the registry, these devices are represented by “Things”, or the so-called “device simulator”. For example, the “Thing” can be your laptop or Raspberry Pi Board.

2.2) The AWS IoT policies, attached to the “Things”, are used to authorize devices to perform AWS IoT operations, such as subscribing or publishing to MQTT topics.

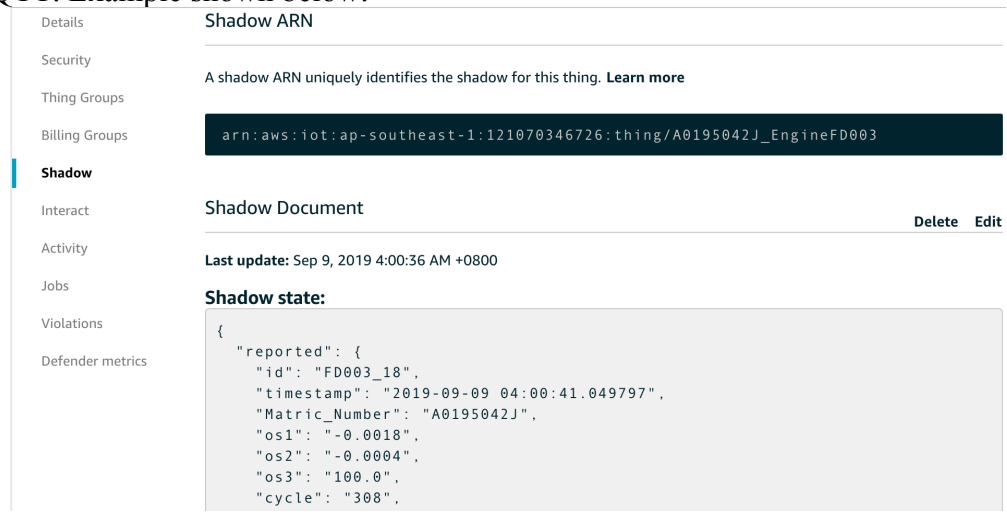
2.3) Message broker - It is responsible for sending all messages published on an MQTT topic to all clients subscribed to that topic.



Data exchange in between the device and cloud is through MQTT protocol. In a MQTT message broker, it uses topic (an UTF-8 string) to filter messages for each connected client.

Devices report their state by publishing messages, in JSON format, on MQTT topics. Each MQTT topic can trace the device when it gets updated.

2.4) Device shadow – It's a JSON document that is used to store and retrieve current state information for a device. You can use the shadow to get and set the state of a device over MQTT. Example shown below:



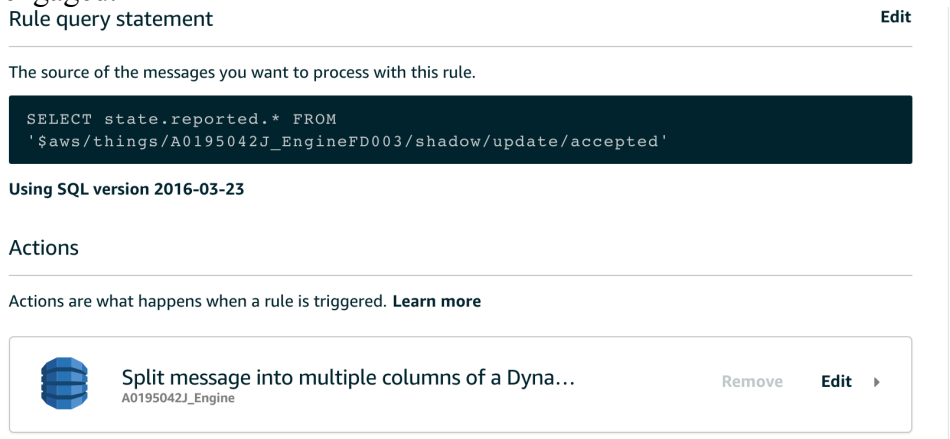
2.5) Rule engine – It's where rules are created. The rules define corresponding actions to perform based on the data in a message. When a rule matches a message, the rules engine triggers the action using the selected properties. Rules also contain an IAM role that grants AWS IoT permission to the AWS resources used to perform the action. An AWS IoT rule consists of an SQL SELECT statement, a topic filter, and a rule action:

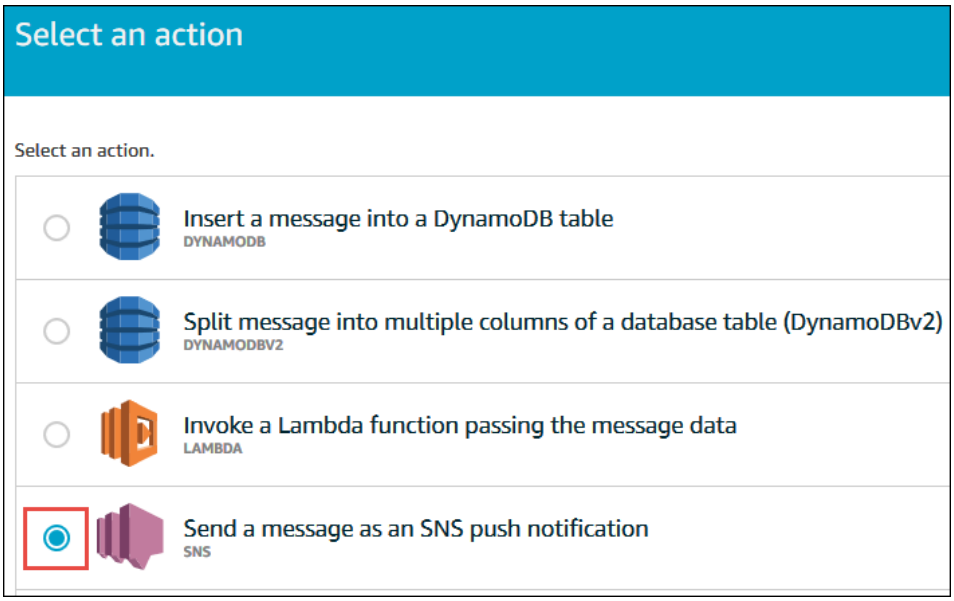
The SQL SELECT statement allows you to extract data from an incoming MQTT message;

The topic filter specifies one or more MQTT topics. The rule is triggered when an MQTT

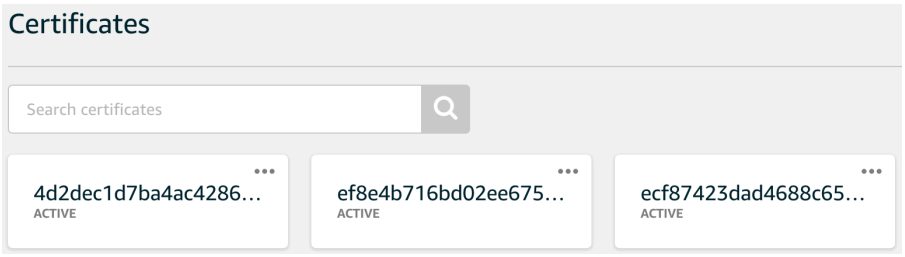
message is received on a topic that matches the topic filter;

Rule actions allow you to take the information extracted from an MQTT message and send it to another AWS service. In this report, DynamoDB database (for storing the published data from device) and AWS SNS (for pushing email alert on abnormality) are engaged.





2.6) Security and Identity - AWS IoT can generate a certificate (X.509) as the credential to secure communication between the device and AWS IoT. The certificate must be registered and activated with AWS IoT, and then copied onto your device. Hence it is 1-to-1 uniquely paired with the device or "Thing".



2.7) AWS services - DynamoDB, Lambda, SNS, S3, and etc.

Rules will be created to define the interactions:

f.1) Publish data into DynamoDB table.

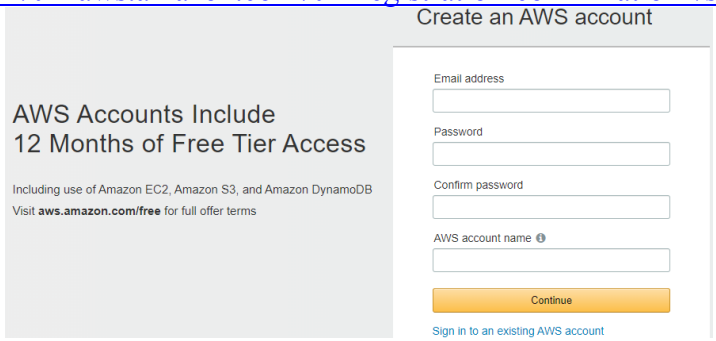
f.2) Push email alert using SNS.

3. AWS IoT Realization

Have known the concept of AWS IoT, the rest is to implement it. I start with creating an AWS free tier account as well as the IAM role.

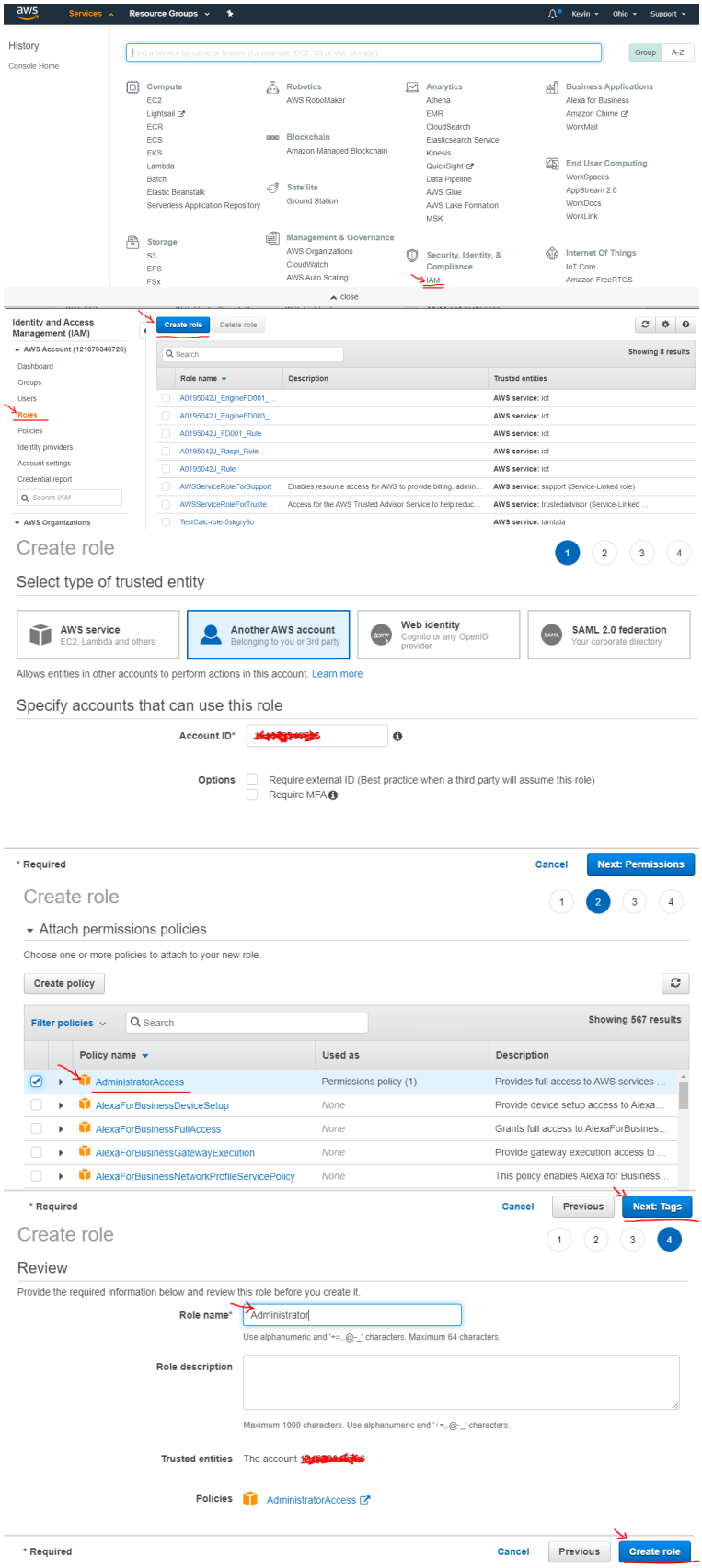
3.1) Log in to url below and create the free tier account first:

https://portal.aws.amazon.com/billing/signup?refid=em_127222&redirect_url=https%3A%2F%2Faws.amazon.com%2Fregistration-confirmation#/start

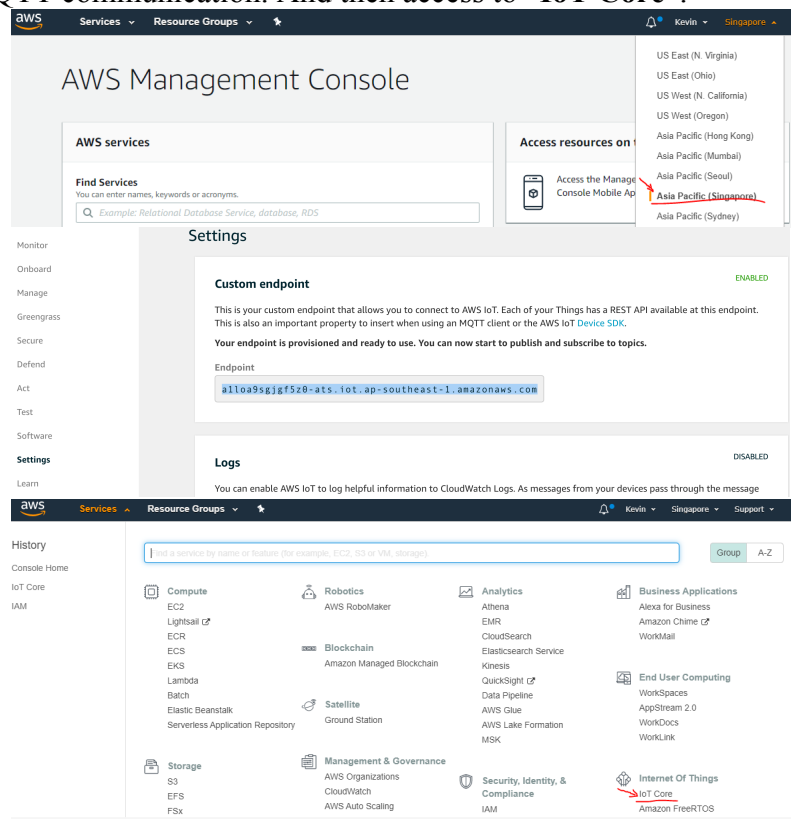


3.2) Within AWS management console

-> IAM under “security, identity, & compliance” -> Roles -> Create role -> Another AWS account -> Account ID -> next:Permission -> check policy name “Administrator Access” -> next:Tags -> skip “add tags” -> next:Review -> add “administrator” as Role name -> “create role”

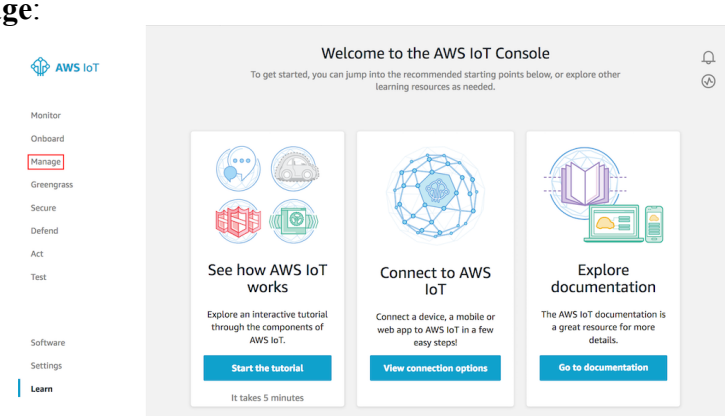

aws
Services
Resource Groups
Kevin
Ohio
Support
History
Console Home
Find a service by name or feature (for example, EC2, S3 or VM, storage).
Group
A-Z
Compute
EC2
Lightsail
ECR
ECS
EKS
Lambda
Batch
Elastic Beanstalk
Serverless Application Repository
Robotics
AWS RoboMaker
Blockchain
Amazon Managed Blockchain
Satellite
Ground Station
Analytics
Athena
EMR
CloudSearch
Elasticsearch Service
Kinesis
QuickSight
Data Pipeline
AWS Glue
AWS Lake Formation
MSK
Business Applications
Alexa for Business
Amazon Chime
WorkMail
End User Computing
WorkSpaces
AppStream 2.0
WorkDocs
WorkLink
Storage
S3
EFS
FSx
Management & Governance
AWS Organizations
CloudWatch
AWS Auto Scaling
Security, Identity, & Compliance
IAM
Internet Of Things
IoT Core
Amazon FreeRTOS
close
Identity and Access Management (IAM)
AWS Account (121070346726)
Dashboard
Groups
Users
Roles
Policies
Identity providers
Account settings
Credential report
Search IAM
AWS Organizations
Create role
Delete role
Search
Showing 8 results
Role name
Description
Trusted entities
A0195042J_EngineFD001_...
A0195042J_EngineFD003_...
A0195042J_FD001_Rule
A0195042J_Raspi_Rule
A0195042J_Rule
AWSServiceRoleForSupport
Enables resource access for AWS to provide billing, admin...
AWS service: support (Service-Linked role)
AWSServiceRoleForTruste...
Access for the AWS Trusted Advisor Service to help reduc...
AWS service: trustedadvisor (Service-Linked ...
TestCalc-role-5skgry6o
AWS service: lambda
AWS service: iot
Create role
1
2
3
4
Select type of trusted entity
AWS service
EC2, Lambda and others
Another AWS account
Belonging to you or 3rd party
Web identity
Cognito or any OpenID provider
SAML 2.0 federation
Your corporate directory
Allows entities in other accounts to perform actions in this account. Learn more
Specify accounts that can use this role
Account ID*
Options
Require external ID (Best practice when a third party will assume this role)
Require MFA
* Required
Cancel
Next: Permissions
Create role
Attach permissions policies
Choose one or more policies to attach to your new role.
Create policy
Filter policies
Search
Showing 567 results
Policy name
Used as
Description
AdministratorAccess
Permissions policy (1)
Provides full access to AWS services ...
AlexaForBusinessDeviceSetup
None
Provide device setup access to Alexa...
AlexaForBusinessFullAccess
None
Grants full access to AlexaForBusines...
AlexaForBusinessGatewayExecution
None
Provide gateway execution access to ...
AlexaForBusinessNetworkProfileServicePolicy
None
This policy enables Alexa for Business...
* Required
Cancel
Previous
Next: Tags
Create role
Review
Provide the required information below and review this role before you create it.
Role name*
Administrator
Use alphanumeric and '+=,.@-_' characters. Maximum 64 characters.
Role description
Maximum 1000 characters. Use alphanumeric and '+=,.@-_' characters.
Trusted entities
The account
Policies
AdministratorAccess
* Required
Cancel
Previous
Create role

3.3) Go back to AWS management console, choose region "**Asia Pacific (Singapore)**", linking to a unique endpoint **host name**, which is to be deployed later for establish MQTT communication. And then access to "**IoT Core**":

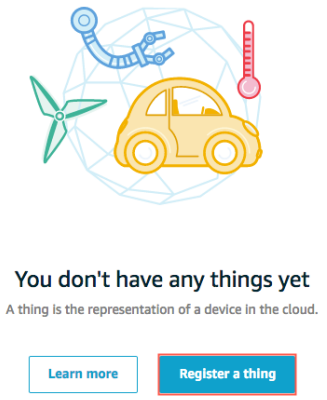


3.4) A device must be registered in the registry first, in order to grant the certificate, private key, and root CA certificate that are used to communicate with AWS IoT:

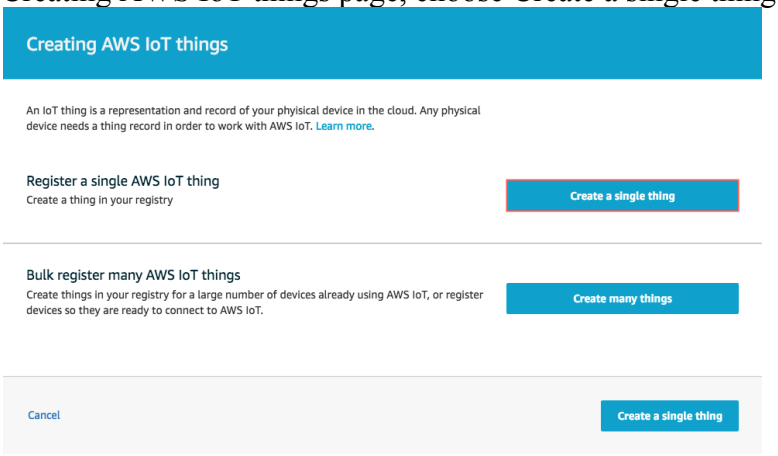
a. On the **Welcome to the AWS IoT Console** page, in the navigation pane, choose **Manage**:



b. On the **You don't have any things yet** page, choose **Register a thing**:



c. On the Creating AWS IoT things page, choose Create a single thing:



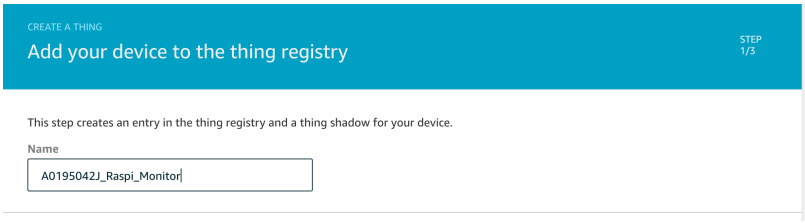
d. On the **Create a thing** page, in the **Name** field, enter a name for your thing;

For task 1 - publish 2 different engines' data into AWS IoT, name the thing as **A0195042J_EngineFD001** & **A0195042J_EngineFD003**.

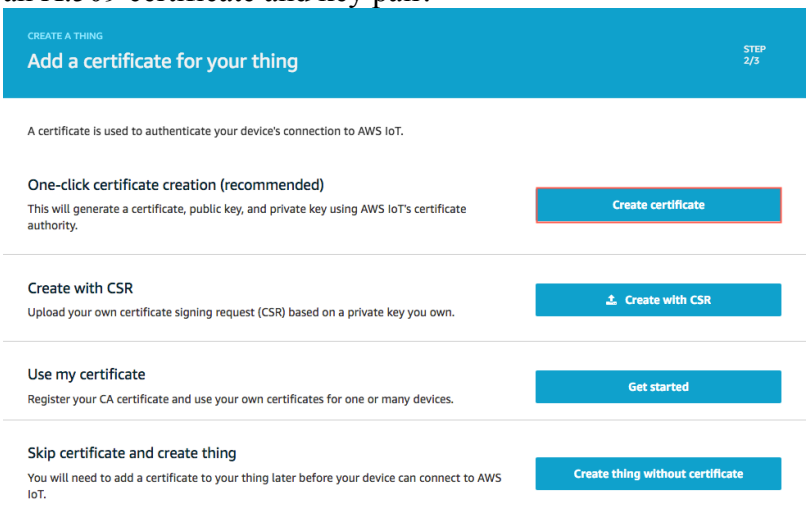
For task 2 – publish downloaded rainfall data into AWS IoT through Raspberry Pi board, name the thing as **A0195042J_Raspi**.

For task 3 – publish Raspberry Pi self-monitoring temperature data, name the thing as **A0195042J_Raspi_Monitor**.

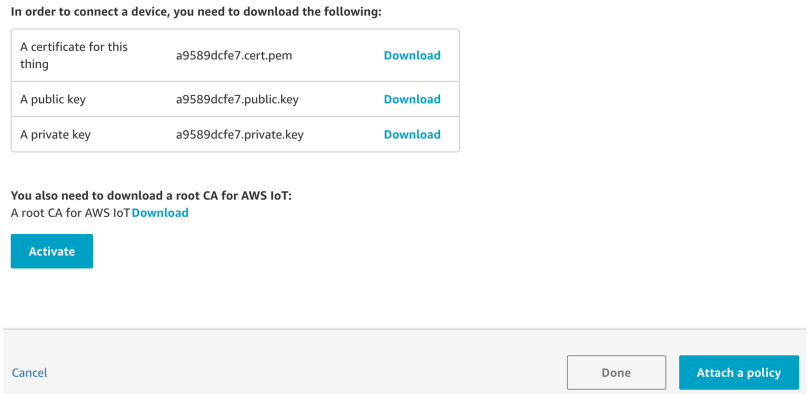
Choose **Next**.



e. On the **Add a certificate for your thing** page, choose **Create certificate**. This generates an X.509 certificate and key pair:



f. On the **Certificate created!** page, download your public and private keys, certificate, and root certificate authority (CA):
f.1) Choose **Download** for your certificate.
f.2) Choose Download for your private key.
f.3) Choose Download for the Amazon root CA. A new webpage is displayed.
Choose **RSA 2048 bit key: Amazon Root CA 1**. This opens another webpage with the text of the root CA certificate. Copy this text and paste it into a file named Amazon_Root_CA_1.pem.
Keep these files to a different directory.
Choose **Activate** to activate the X.509 certificate, and then choose **Attach a policy**.



Server Authentication

Server certificates allow your devices to verify that they're communicating with AWS IoT and not another server impersonating AWS IoT. Service certificates must be copied onto your device and referenced when devices connect to AWS IoT. For more information, see the AWS IoT Device SDKs.

AWS IoT server certificates are signed by one of the following CA certificates:

VeriSign Endpoints (legacy)

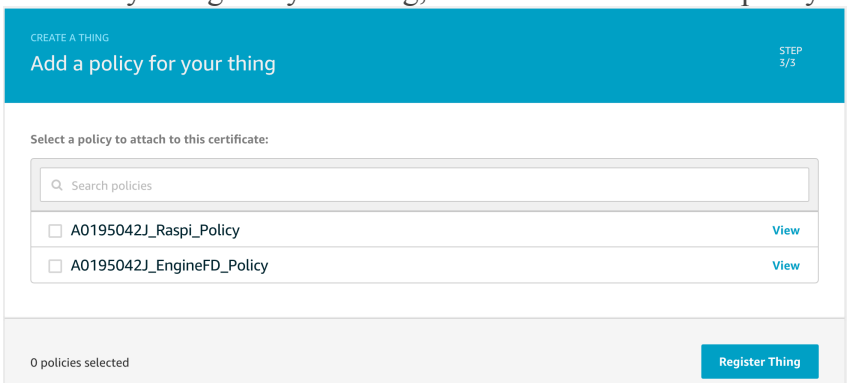
- RSA 2048 bit key: VeriSign Class 3 Public Primary G5 root CA certificate

Amazon Trust Services Endpoints (preferred)

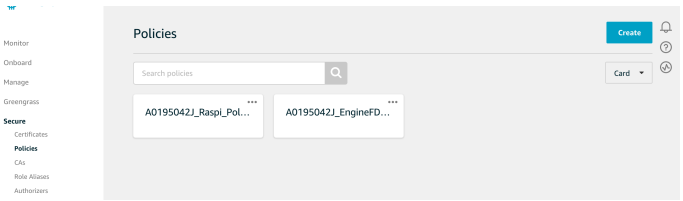
- RSA 2048 bit key: Amazon Root CA 1.
- RSA 4096 bit key: Amazon Root CA 2 - Reserved for future use.
- ECC 256 bit key: Amazon Root CA 3.
- ECC 384 bit key: Amazon Root CA 4 - Reserved for future use.

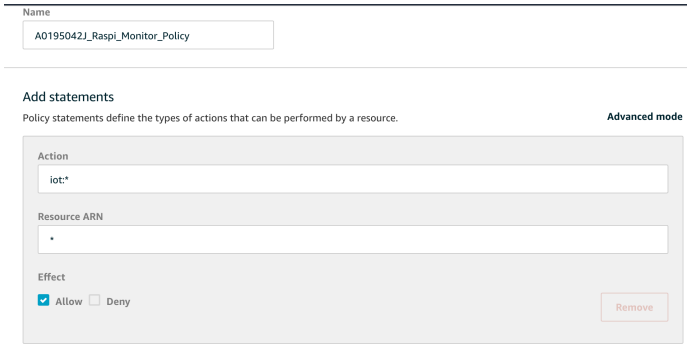
3.5) Add a policy to the thing, and attach this policy to the certificate.
This policy allows your device to perform all AWS IoT actions on all AWS IoT resources.

a. On the **Add a policy for your thing** page, choose **Register Thing**.
After you register your thing, create and attach a new policy to the certificate.

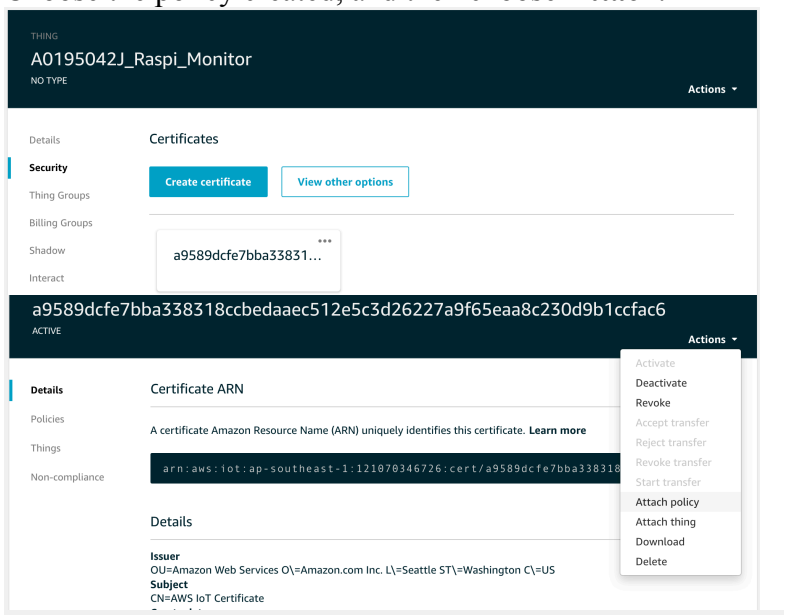


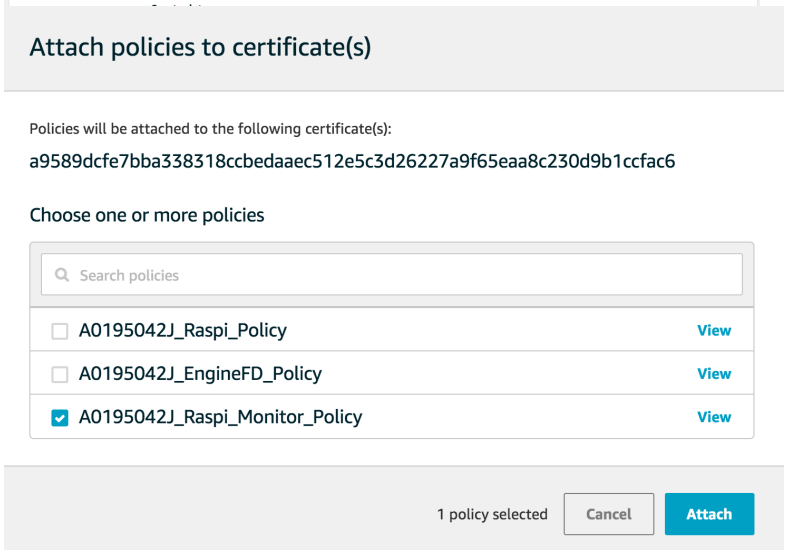
b. On the AWS IoT console, in the navigation pane, choose **Secure**, and then choose **Policies -> Create**.
On the **Create a policy** page:
b.1) Enter a **Name** for the policy:
For task1 - **A0195042J_EngineFD_Policy**
For task2 - **A0195042J_Raspi_Policy**
For task3 - **A0195042J_Raspi_Monitor_Policy**
b.2) For **Action**, enter **iot:***. For **Resource ARN**, enter *.
b.3) Under **Effect**, choose **Allow**, and then choose **Create**.





c. Choose the previous created AWS IoT thing -> **Security** -> choose linked certificate. In the certificate detail page, choose **Actions**, and then choose **Attach policy**. Choose the policy created, and then choose **Attach**.





3.6) Configure and test the rules for publish data into DynamoDB table

a. To publish the data from the device/Thing into DynamoDB table, the rule splits the data into columns of a DynamoDB table.

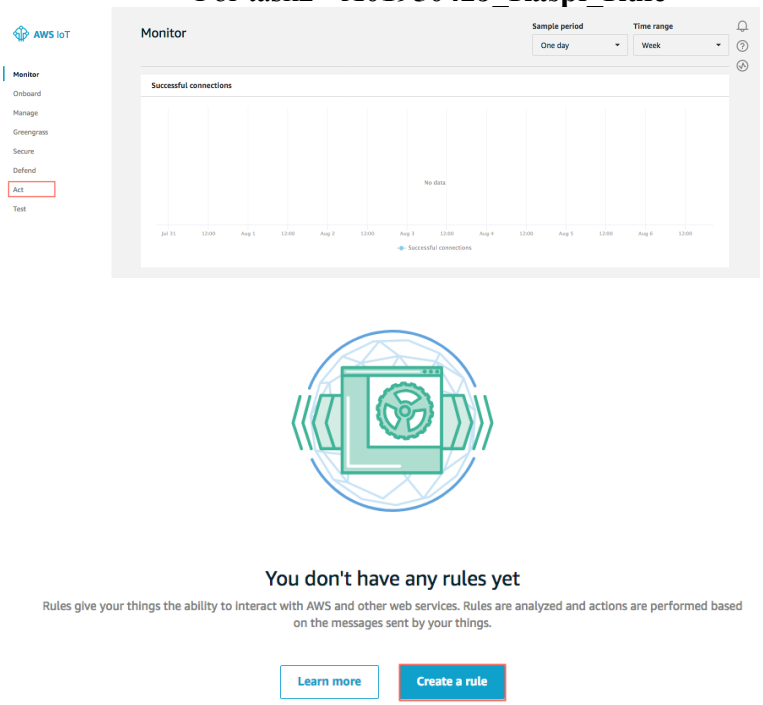
a.1) In the AWS IoT console, in the left navigation pane, choose **Act**.

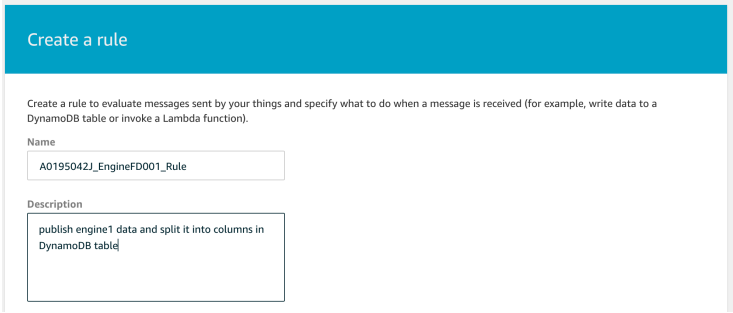
a.2) On the **Act** page, choose **Create a rule.**

a.3) On the **Create a rule** page, in the **Name** field, enter a name for your rule

For task1 - **A0195042J_EngineFD001_Rule & A0195042J_EngineFD003_Rule**

For task2 - **A0195042J_Raspi_Rule**





b. Scroll down to **Rule query statement**. Choose the latest version from the **Using SQL version** drop-down list (2016-03-23).

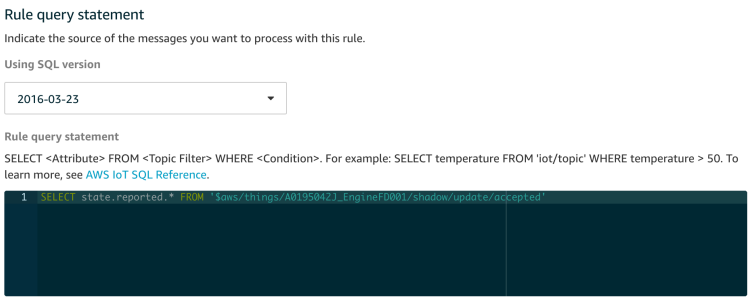
In the **Rule query statement** field,

For task1, enter:

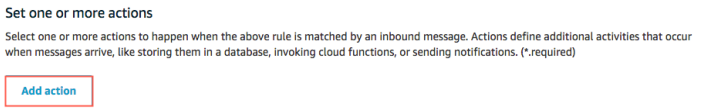
SELECT state.reported.* FROM '$aws/things/A0195042J_EngineFD001/shadow/update/accepted'
SELECT state.reported.* FROM '$aws/things/A0195042J_EngineFD003/shadow/update/accepted'

For task 2, enter:

SELECT state.reported.* FROM '$aws/things/A0195042J_Raspi/shadow/update/accepted'

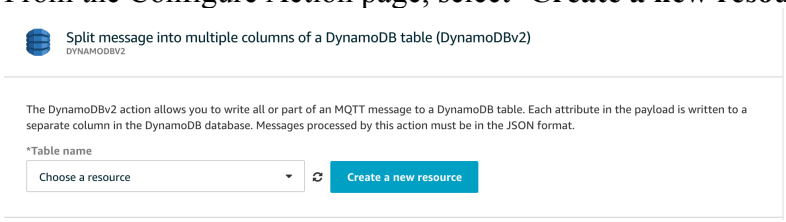


c. In **Set one or more actions**, choose **Add action**

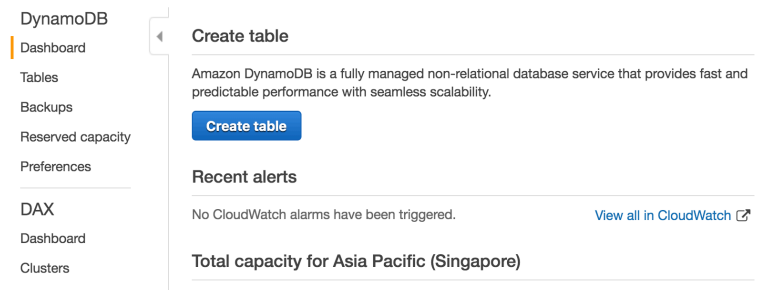


d. On the **Select an action** page, choose **Split message into multiple columns of a DynamoDB table, configure action**

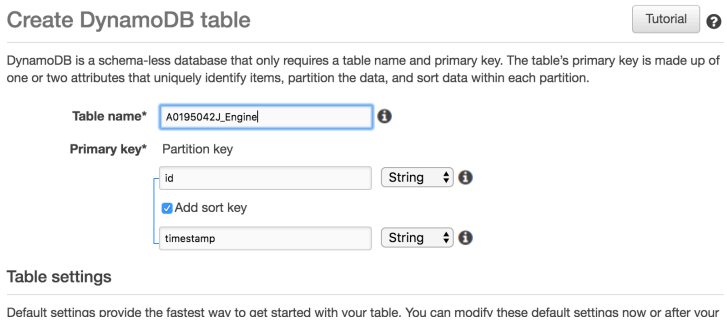
e. From the Configure Action page, select '**Create a new resource**'.



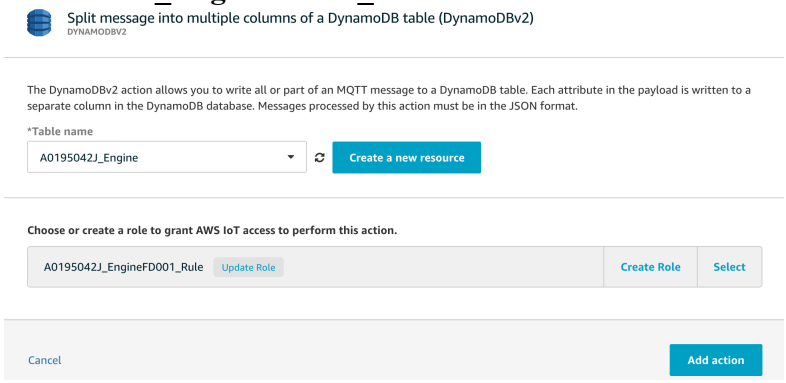
f. In the newly pop-out window, select "**Create table**"



g. Provide the table name, primary key, and sort key, then select "**Create**"
For task1, table name – **A0195042J_Engine**
For task 2, table name – **A0195042J_Raspi**
Primary key is common – **id**
Sort key is also common – **timestamp**



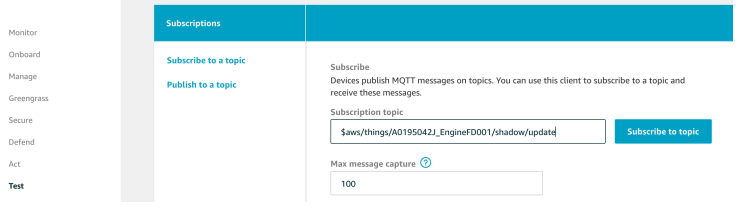
h. Go back to previous window, from scroll-down list select just created table name - e.g. **A0195042J_Engine,** and also select the rule created earlier - e.g. **A0195042J_EngineFD001_Rule**. Then select "**Add action**":



i. In order for the shadow document to authenticate and receive the actual data sent from device by the python script, we need to first simulate the MQTT communication by configuring the shadow document in JSON format, then test it based on what we have established the IoT configuration - Thing attached with certificate, policy, and the defined rule action.
Devices publish MQTT messages on topics. Hence we use the AWS IoT MQTT client to subscribe to these topics to see these messages.

Take engine1 (thing) as an example, it is from this Shadow Document that the device stores its information payload to either publish or subscribe to a Topic. Topics are communication channels in which the devices connect and communicate with each other. For this verification, we make use of the following Topics as listed below:
In general, they are:
$aws/things/A0195042J_EngineFD001/shadow/update
$aws/things/A0195042J_EngineFD001/shadow/update/accepted
$aws/things/A0195042J_EngineFD001/shadow/get
$aws/things/A0195042J_EngineFD001/shadow/get/accepted

Go to **Test**, enter "$aws/things/**A0195042J_EngineFD001**/shadow/update" in the subscription topic field, then click "**subscribe to topic**":

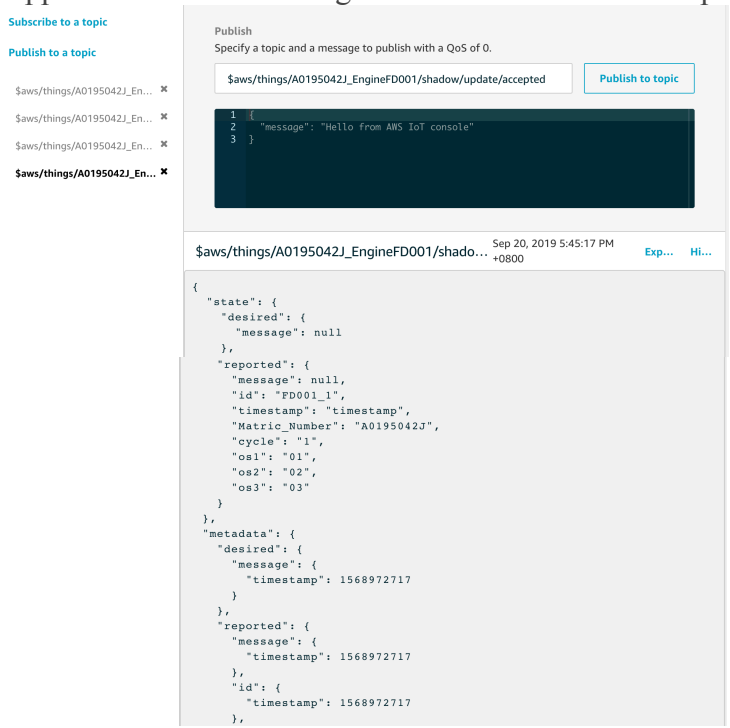


Do a similar "subscribe a topic" for the rest 3 topics:

Select topic of "$aws/things/A0195042J_EngineFD001/shadow/update", edit JSON message as below, and click "publish to topic":

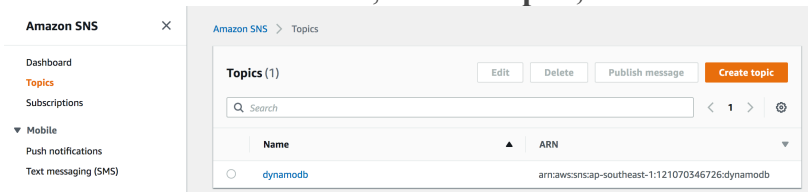
```
{
 "state": {
  "desired": {
   "message": null
   },
  "reported": {
   "message": null,
   "id": "FD001_1",
   "timestamp": "timestamp",
   "Matric_Number": "A0195042J",
   "cycle": "1",
   "os1": "01",
   "os2": "02",
   "os3": "03"
  }
 }
}
```

In the topic “$aws/things/A0195042J_EngineFD001/shadow/update/accepted”, it is supposed to receive message like below for successful publishing:

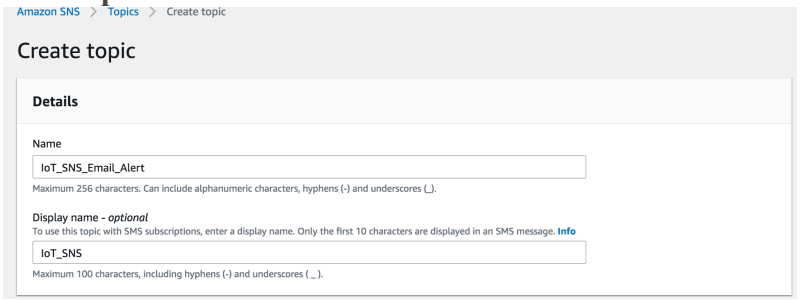


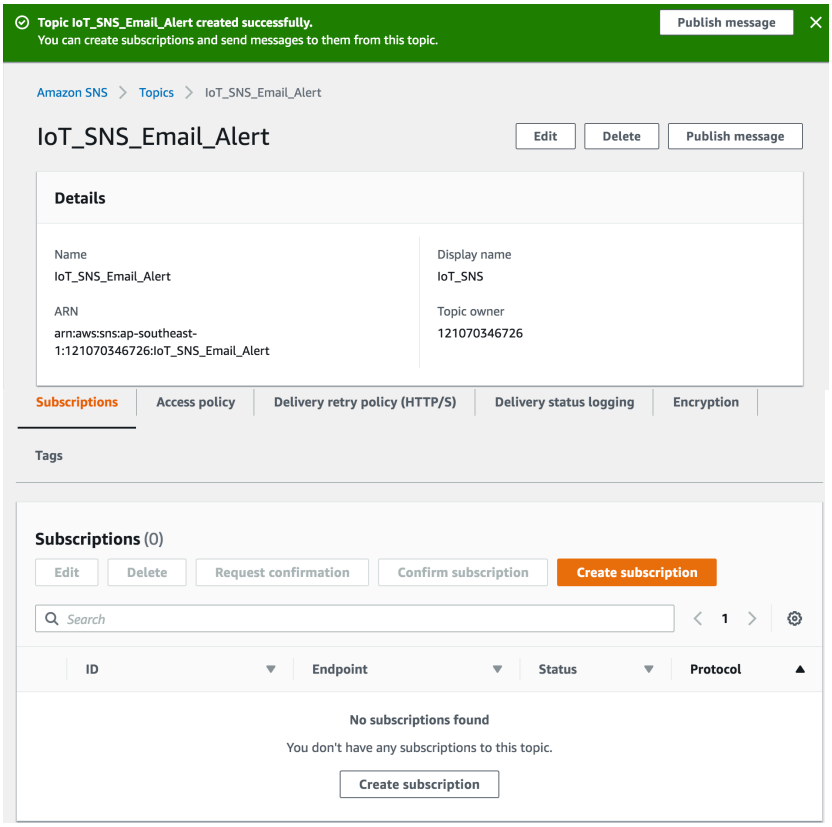
3.7) To receive email alert on abnormal reading, rule needs to be created to engage with AWS SNS service. In task3, rule is configured so that the abnormal temperature will be sent from raspberry pi board to an Amazon SNS topic.

a. From the AWS **SNS** console, choose **Topics,** and then choose **Create topic**

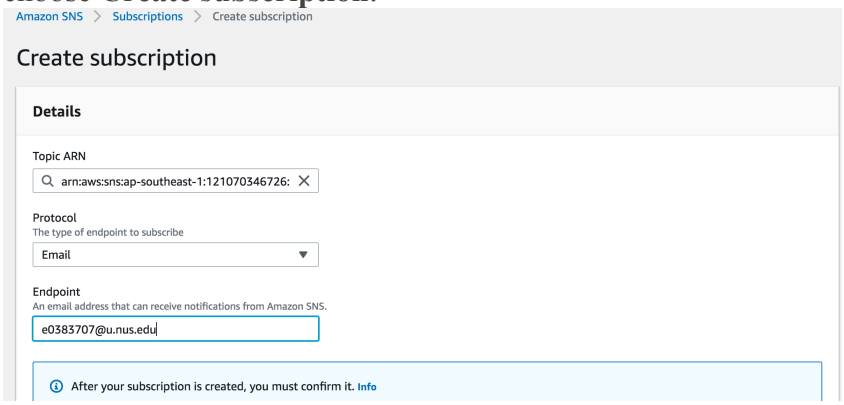


b. Enter a name for the topic - **IoT_SNS_Email_Alert**
Enter a display name for the topic – **IoT_SNS**
Then choose **Create topic** and under subscriptions tab, choose **Create subscription**:

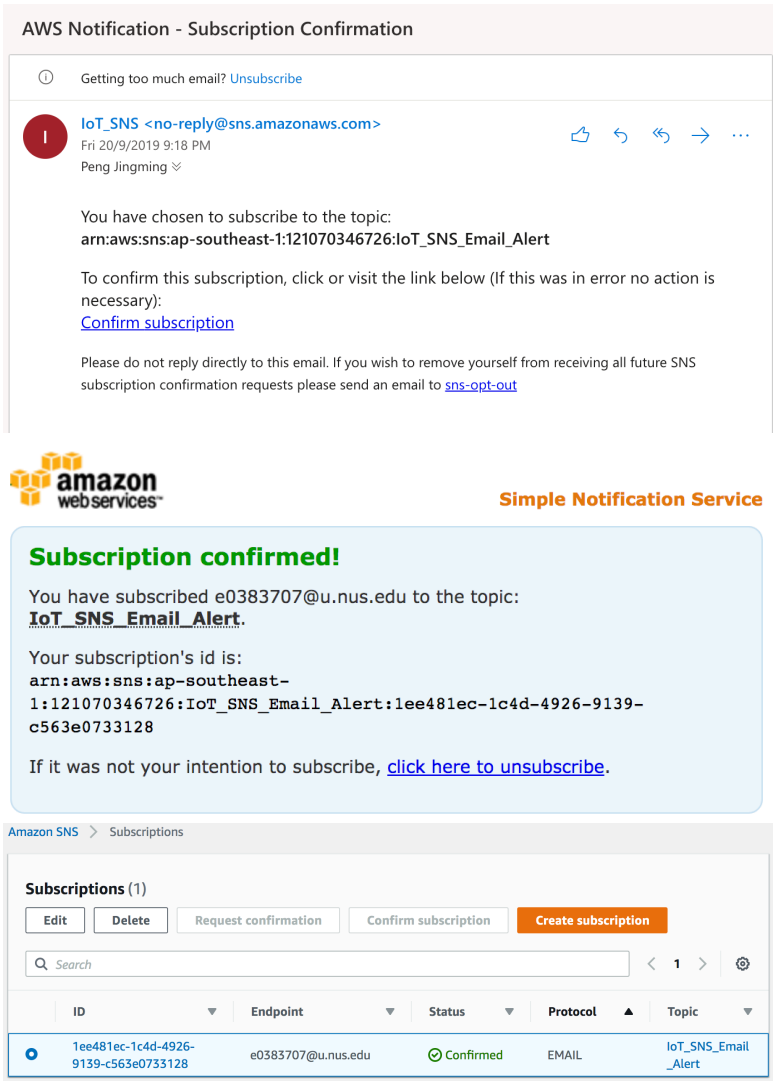




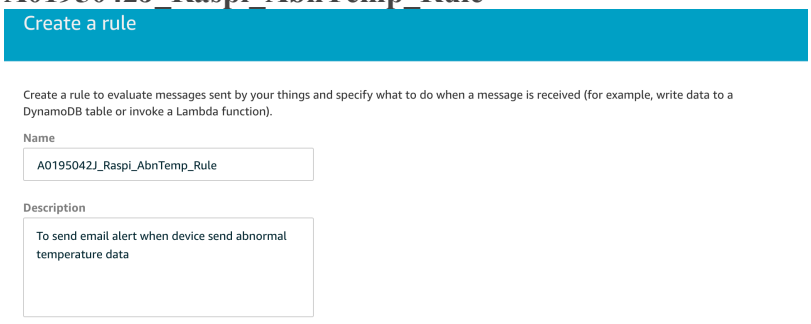
c. On **Create subscription**, from the **Protocol** drop-down list, choose **Email**. In the **Endpoint** field, enter school email – **e0383707@u.nus.edu**, and then choose **Create subscription**:



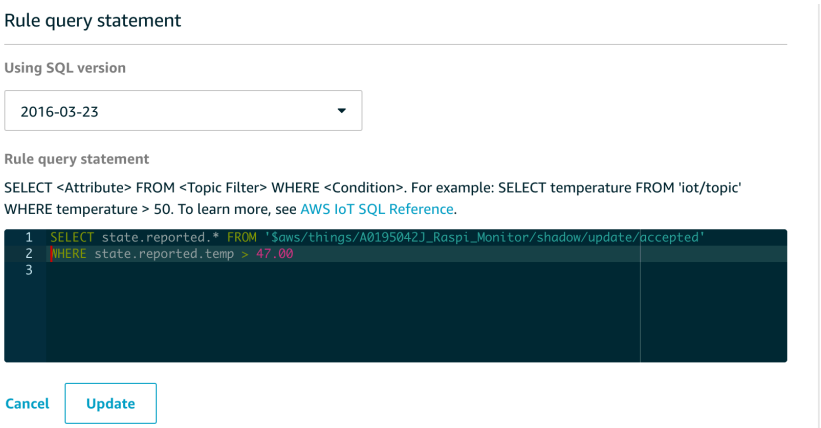
After your subscription is created, you must confirm it. Amazon SNS doesn't send messages to an endpoint until the subscription is confirmed. Hence, access into the email and click the url link to confirm this subscription:



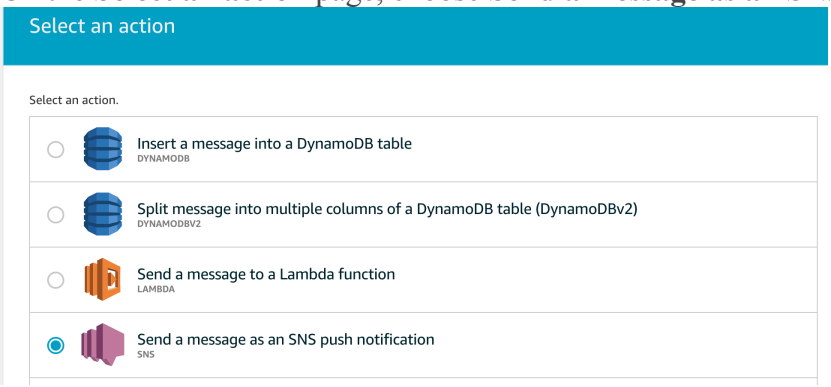
d. Create corresponding rules to trigger this SNS action, enter a name for it - **A0195042J_Raspi_AbnTemp_Rule**



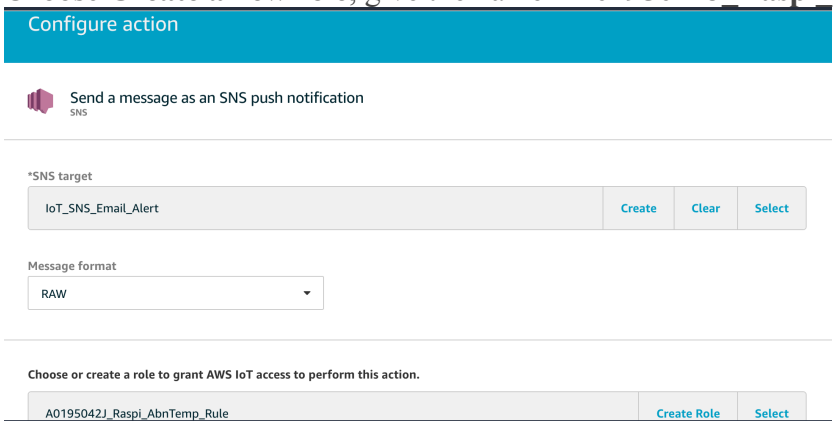
Under **Rule query statement**, enter the following AWS IoT SQL query statement:
SELECT state.reported.* FROM '$aws/things/A0195042J_Raspi_Monitor/shadow/update/accepted' WHERE state.reported.temp > 47.00



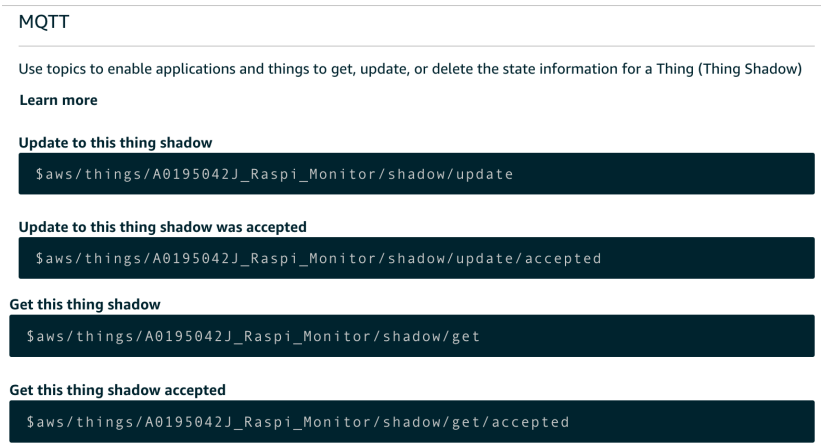
On the **Select an action** page, choose **Send a message as an SNS push notification:**



On the **Configure action** page, under **SNS target**, choose **Select** to expand the SNS topic. Then choose **Select** next to the Amazon SNS topic you created earlier.
Under **Message format**, choose **RAW**.
Choose **Create a new role**, give the name - **A0195042J_Raspi_AbnTemp_Rule**:



e. Follow the similar steps in 1.11) to test the rule, this time the MQTT topics are the following:
$aws/things/A0195042J_Raspi_Monitor/shadow/update
$aws/things/A0195042J_Raspi_Monitor/shadow/update/accepted
$aws/things/A0195042J_Raspi_Monitor/shadow/get
$aws/things/A0195042J_Raspi_Monitor/shadow/get/accepted



Select topic of “$aws/things/A0195042J_Raspi_Monitor/shadow/update”, edit JSON message as below, and click “publish to topic”:

```
{
 "state": {
            "desired": {
              "message": null
              },
            "reported": {
              "message": null,
              "temp": "60",
              "timestamp": "timestamp",
              "Matric_Number": "A0195042J"
            }
          }
        }
```

In the topic “$aws/things/A0195042J_Raspi_Monitor/shadow/update/accepted”, it is supposed to receive message like below for successful publishing:

3.8) In a short summary, for the 3 tasks, what have been created for the Things, certificates, keys, policies, rules, tables:

	Task1	
Thing	A0195042J_EngineFD001	A0195042J_EngineFD003
X.509 Certificate	ecf87423da-certificate.pem.crt	ef8e4b716b-certificate.pem.crt
Private Key	ecf87423da-private.pem.key	ef8e4b716b-private.pem.key
Public Key	ecf87423da-public.pem.key	ef8e4b716b-public.pem.key
root CA	AmazonRootCA1.pem	AmazonRootCA1.pem
Policy	A0195042J_EngineFD_Policy	A0195042J_EngineFD_Policy
Rule	A0195042J_EngineFD001_Rule	A0195042J_EngineFD003_Rule
table	A0195042J_Engine	A0195042J_Engine

	Task2
Thing	A0195042J_Raspi
X.509 Certificate	4d2dec1d7b-certificate.pem.crt
Private Key	4d2dec1d7b-private.pem.key
Public Key	4d2dec1d7b-public.pem.key
root CA	AmazonRootCA1.pem
Policy	A0195042J_Raspi_Policy
Rule	A0195042J_Raspi_Rule
table	A0195042J_Raspi

	Task3
Thing	A0195042J_Raspi_Monitor
X.509 Certificate	a9589dcfe7-certificate.pem.crt
Private Key	a9589dcfe7-private.pem.key
Public Key	a9589dcfe7-public.pem.key
root CA	AmazonRootCA1.pem
Policy	A0195042J_Raspi_Monitor_Policy
Rule	A0195042J_Raspi_AbnTemp_Rule
table	N.A.

4. Visualization Concept

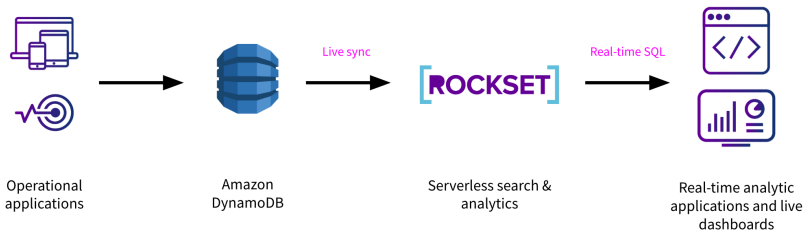
There're various ways of visualizing data that were transferred into the DynamoDB table:

4.1) By python – directly call matplotlib library to plot out the data;

4.2) By API gateway – create the API gateway to retrieve and present data in url;

4.3) By BI tools – establish connection between DynamoDB and BI tool, and visualize data inside BI tool.

In this report, I am demonstrating how to connect DynamoDB with Tableau through Rockset JDBC driver, and visualize the data inside Tableau:



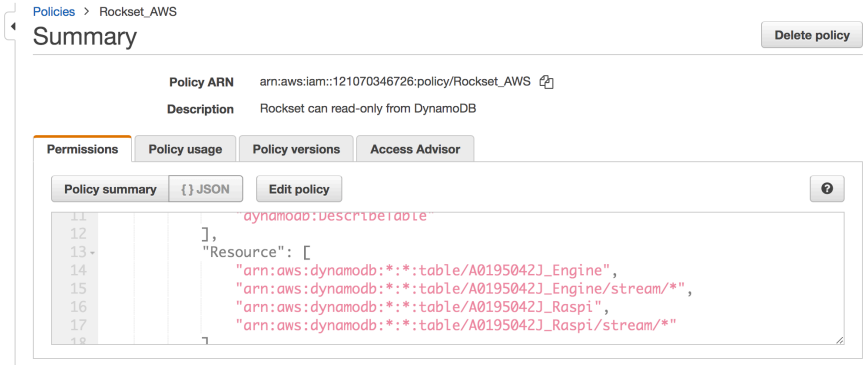
5. Visualization Realization

5.1) Configure AWS IAM policy

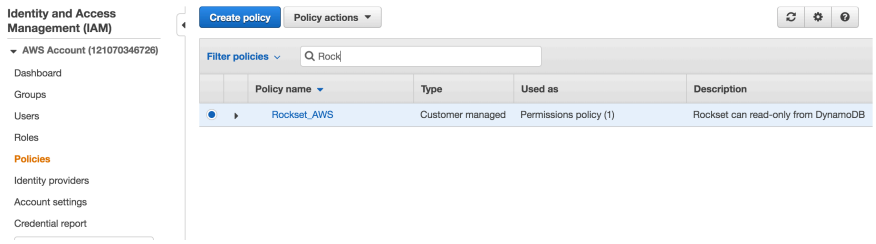
a. Navigate to the IAM Service in the AWS Management Console.
b. Set up a new policy by navigating to **Policies** and clicking **Create policy** – **Rockset_AWS**
c. Set up read-only access to your DynamoDB table, switch to the "JSON" tab and paste the policy shown below:

Table name are:
For task 1 - **A0195042J_Engine**;
For task 2 - **A0195042J_Raspi**;

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:Scan",
        "dynamodb:DescribeStream",
        "dynamodb:GetRecords",
        "dynamodb:GetShardIterator",
        "dynamodb:DescribeTable"
      ],
      "Resource": [
        "arn:aws:dynamodb:*:*:table/A0195042J_Engine",
        "arn:aws:dynamodb:*:*:table/A0195042J_Engine/stream/*",
        "arn:aws:dynamodb:*:*:table/A0195042J_Raspi",
        "arn:aws:dynamodb:*:*:table/A0195042J_Raspi/stream/*"
      ]
    }
  ]
}
```



d. Save this newly created policy:

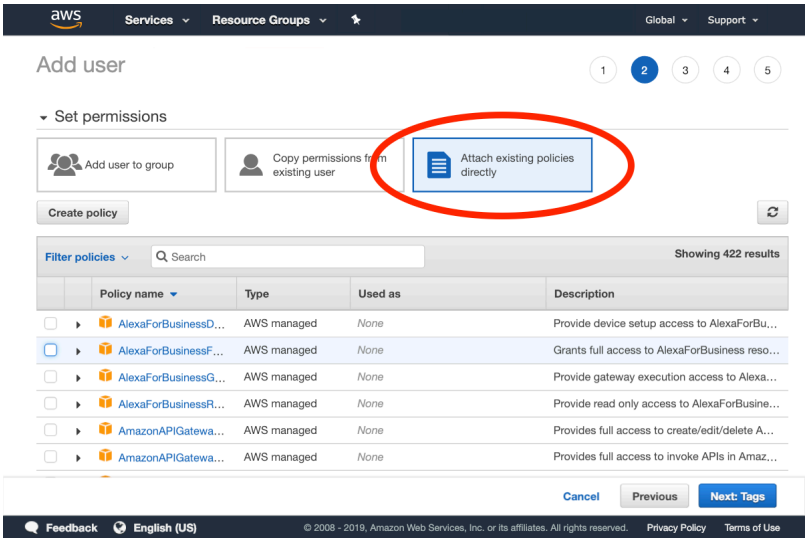


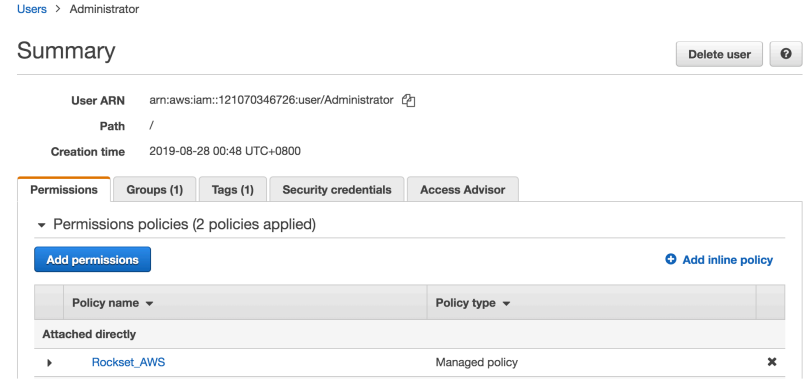
5.2) Grant Rockset permissions to access your AWS resource using AWS Access Keys.

a. Create a new user by navigating to **Users** and clicking **Add User - Administrator**

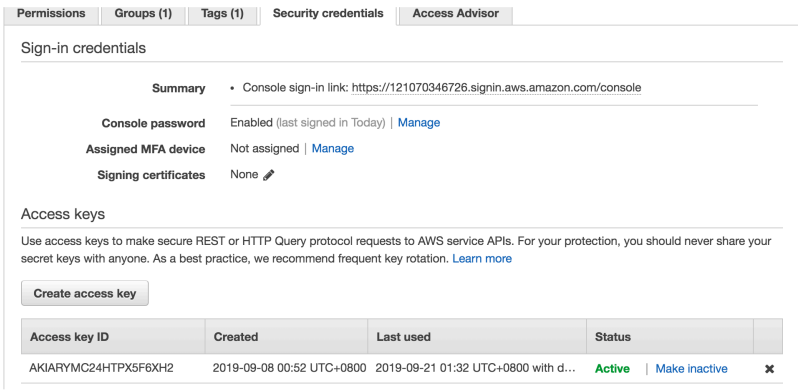
b. Enter a user name and check the **Programmatic access** option. Click to continue.

c. Choose **Attach existing policies directly** then select the policy you created in 5.1)





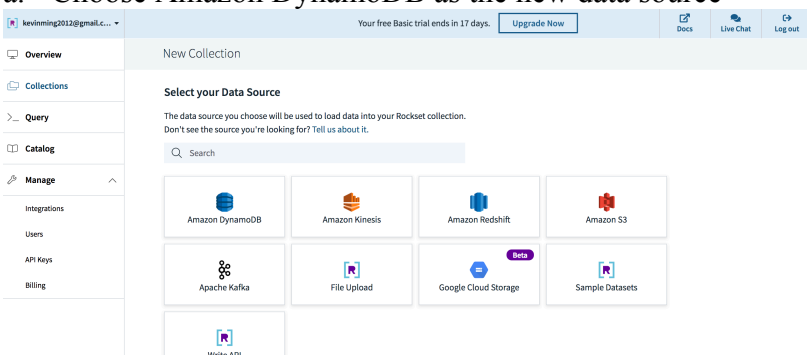
d. When the new user is successfully created you should see the **Access key ID** and **Secret access key** displayed on the screen.



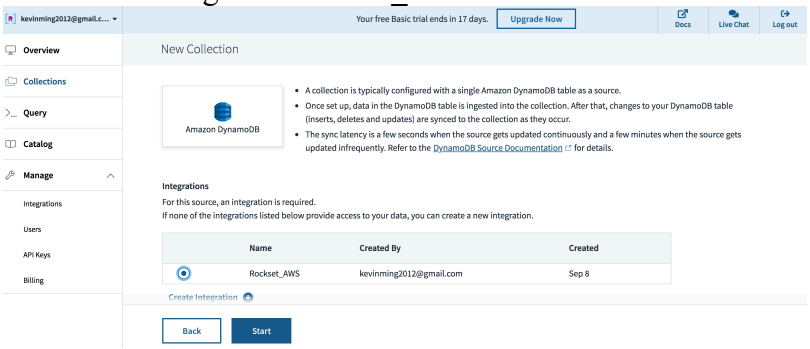
5.3) Create a collection backed by Amazon DynamoDB, Rockset scans the DynamoDB tables to continuously ingest and then subsequently uses the stream to update collections as new objects are added to the DynamoDB table. The sync latency is no more than 5 seconds under regular load.

In the Rockset Console, create a collection from Workspace > Collections > Create Collection:

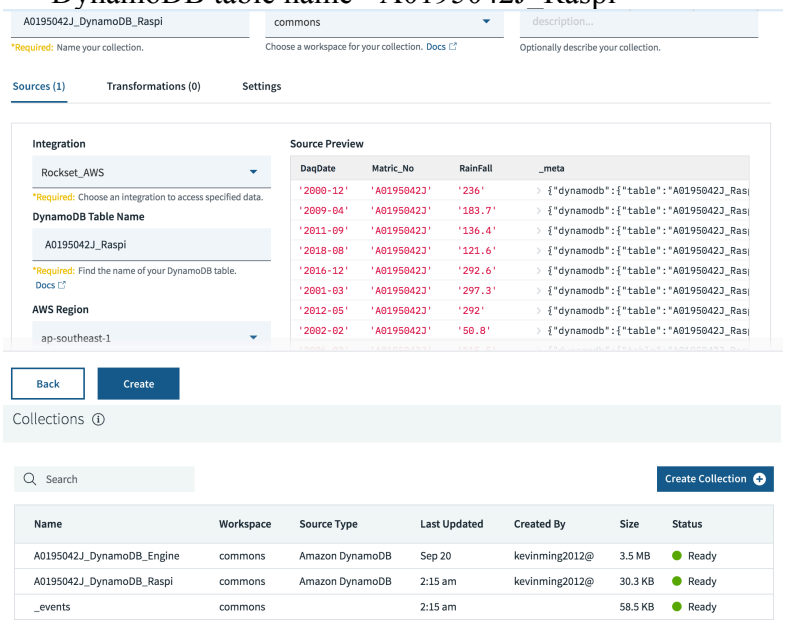
a. Choose Amazon DynamoDB as the new data source



b. Choose integration - **Rock_AWS** and then **Start**

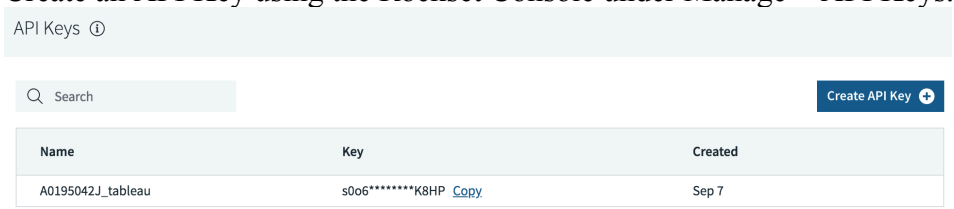


c. Choose Collection name, DynamoDB table name, and AWS Region, then **create**
For task1:
Collection name - A0195042J_DynamoDB_Engine
DynamoDB table name - A0195042J_Engine
For task2:
Collection name - A0195042J_DynamoDB_Raspi
DynamoDB table name - A0195042J_Raspi

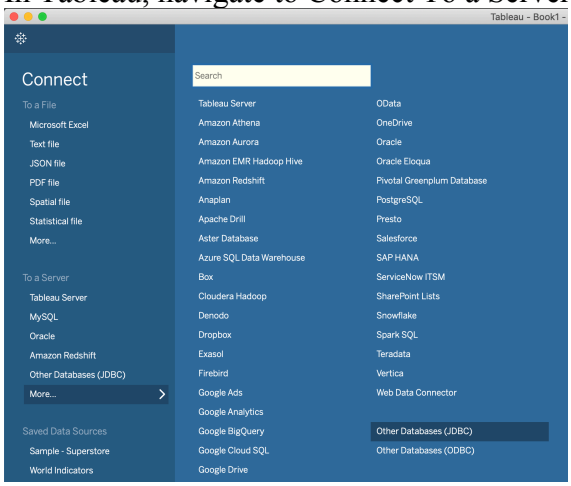


5.4) Set up Tableau Desktop and configure it for use with Rockset.
Tableau Desktop uses JDBC driver to perform operations.
JDBC support is available in Tableau Desktop version 2019.1 (or later) and Rockset Java Client version 0.5.11 (or later).

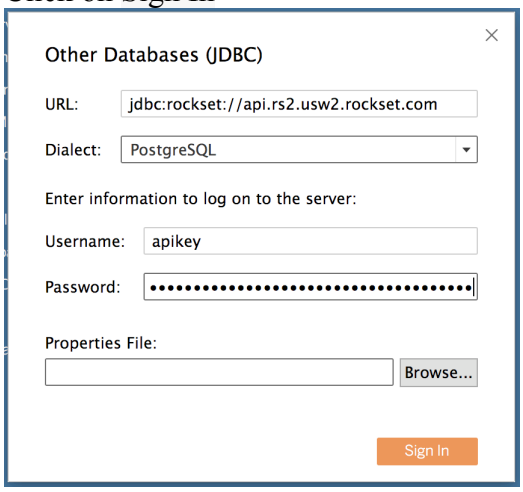
a. Download the jar (rockset-java-$version.jar) from https://oss.sonatype.org/#nexus-search;quick~rockset
b. Place the Rockset Java Client jar in the following folder:
   Mac: ~/Library/Tableau/Drivers
c. Create an API Key using the Rockset Console under Manage > API Keys.



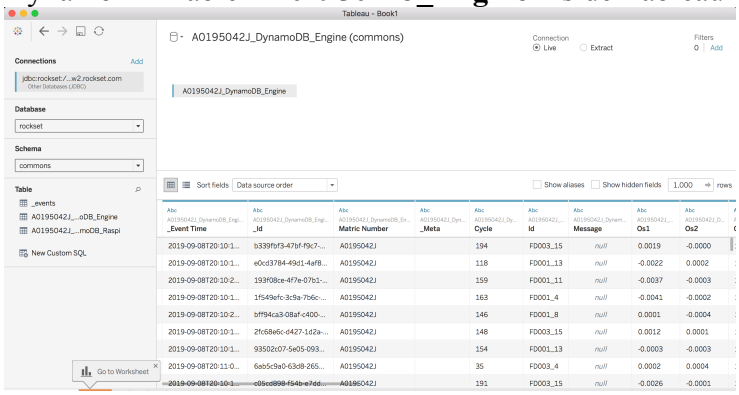
d. In Tableau, navigate to Connect To a Server and select Other Databases (JDBC):



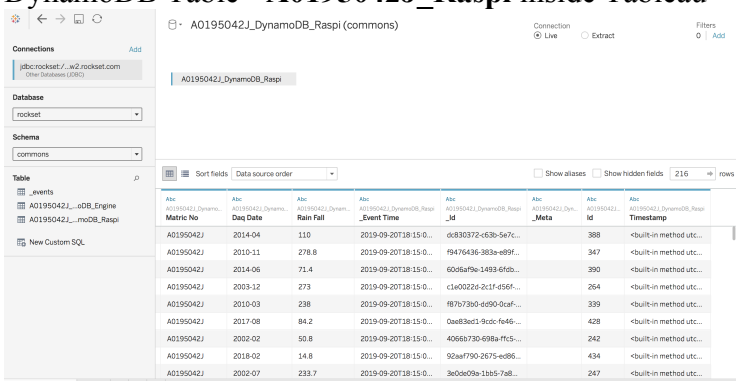
e. Configure the connection as follows:
   Use jdbc:rockset://api.rs2.usw2.rockset.com for URL
   Select PostgreSQL as dialect.
   Enter apikey as Username and Rockset API Key as the password.
   Click on Sign In



DynamoDB Table - **A0195042J_Engine** inside Tableau



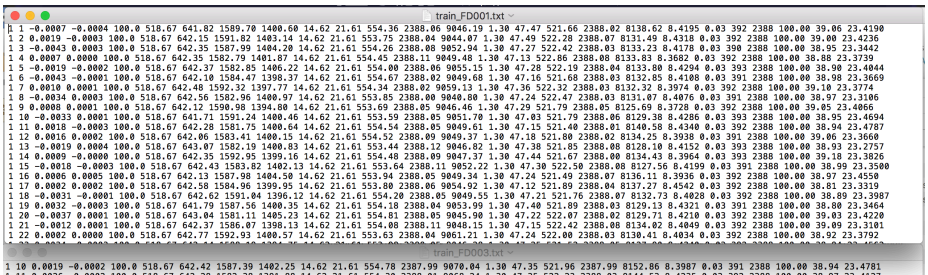
DynamoDB Table - **A0195042J_Raspi** inside Tableau



6. Task 1 – Publish Engine Data into DynamoDB through laptop and do visualization

6.1) Background

We are given jet engine data in four different text files which correspond to each of the jet engines in a propulsion system (with space delimiters to separate the fields) which are shown as follows:



I am using engine1 and engine3 data for this task.

There is already a total of 26 sets of data per line, which is given in the following format:

a. ID
b. OS1, OS2, OS3
c. Sensors 1 to 22

Required to add more field as columns in DynamoDB table:

a. Cycle
b. Timestamp
c. Matric_Number

6.2) Configure the endpoint device – laptop

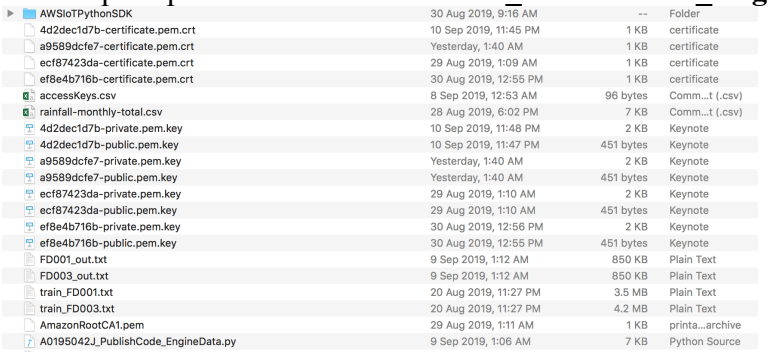
a. Download AWS_IoT_Python_SDK and install into the laptop – Mac OS
   Within the terminal:
   Install brew by running:
   /usr/bin/ruby -e "$(curl -fsSL
   https://raw.githubusercontent.com/Homebrew/install/master/install)"
   Install openssl and python3:
   brew update
   brew install openssl
   brew install python@3
   Check version:
   python3 --version
   import ssl
   print(ssl.OPENSSL_VERSION)
   exit()
   Note that, the OpenSSL version need to be 1.0.1 or later; Python version need to be Python 3.3+.
   Install the AWS IoT Device SDK for Python by running:
   cd ~
   git clone https://github.com/aws/aws-iot-device-sdk-python.git
   cd aws-iot-device-sdk-python
   python setup.py install
b. Put all related files into a common directory
   The files consist of:
   b.1 Downloaded AWS IoT device SDK (consists of MQTT library)

AWSIoTPythonSDK	30 Aug 2019, 9:16 AM	--	Folder
__pycache__	30 Aug 2019, 9:30 AM	--	Folder
core	30 Aug 2019, 9:30 AM	--	Folder
exception	30 Aug 2019, 9:30 AM	--	Folder
__init__.py	30 Aug 2019, 9:16 AM	24 bytes	Python Source
MQTTLib.py	30 Aug 2019, 9:16 AM	61 KB	Python Source

   b.2 Engine1 and engine3 data

train_FD001.txt	20 Aug 2019, 11:27 PM	3.5 MB	Plain Text
train_FD003.txt	20 Aug 2019, 11:27 PM	4.2 MB	Plain Text

   b.3 Thing's certificate, key, CA
   b.4 Script to publish data – **A0195042J_PublishCode_EngineData.py**

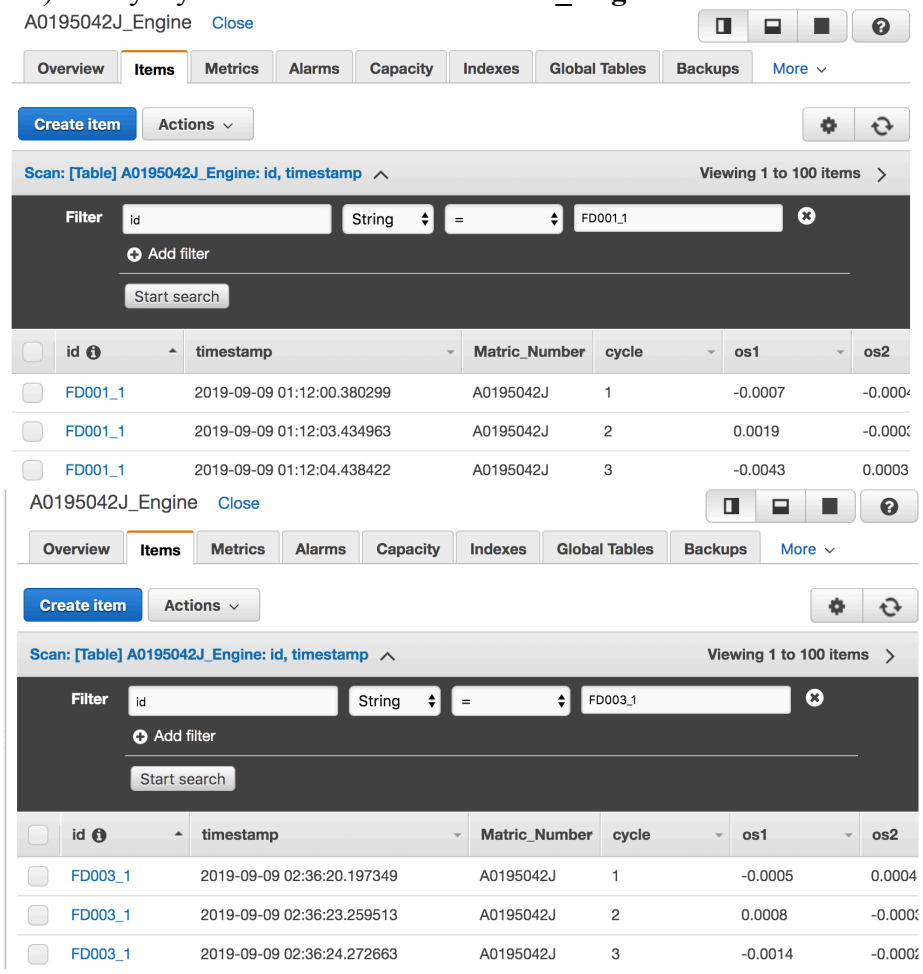


AWSIoTPythonSDK	30 Aug 2019, 9:16 AM	--	Folder
4d2dec1d7b-certificate.pem.crt	10 Sep 2019, 11:45 PM	1 KB	certificate
a9589dcfe7-certificate.pem.crt	Yesterday, 1:40 AM	1 KB	certificate
ecf87423da-certificate.pem.crt	29 Aug 2019, 1:09 AM	1 KB	certificate
ef8e4b716b-certificate.pem.crt	30 Aug 2019, 12:55 PM	1 KB	certificate
accessKeys.csv	8 Sep 2019, 12:53 AM	96 bytes	Comm...t (.csv)
rainfall-monthly-total.csv	28 Aug 2019, 6:02 PM	7 KB	Comm...t (.csv)
4d2dec1d7b-private.pem.key	10 Sep 2019, 11:48 PM	2 KB	Keynote
4d2dec1d7b-public.pem.key	10 Sep 2019, 11:47 PM	451 bytes	Keynote
a9589dcfe7-private.pem.key	Yesterday, 1:40 AM	2 KB	Keynote
a9589dcfe7-public.pem.key	Yesterday, 1:40 AM	451 bytes	Keynote
ecf87423da-private.pem.key	29 Aug 2019, 1:10 AM	2 KB	Keynote
ecf87423da-public.pem.key	29 Aug 2019, 1:10 AM	451 bytes	Keynote
ef8e4b716b-private.pem.key	30 Aug 2019, 12:56 PM	2 KB	Keynote
ef8e4b716b-public.pem.key	30 Aug 2019, 12:55 PM	451 bytes	Keynote
FD001_out.txt	9 Sep 2019, 1:12 AM	850 KB	Plain Text
FD003_out.txt	9 Sep 2019, 1:12 AM	850 KB	Plain Text
train_FD001.txt	20 Aug 2019, 11:27 PM	3.5 MB	Plain Text
train_FD003.txt	20 Aug 2019, 11:27 PM	4.2 MB	Plain Text
AmazonRootCA1.pem	29 Aug 2019, 1:11 AM	1 KB	printa...archive
A0195042J_PublishCode_EngineData.py	9 Sep 2019, 1:06 AM	7 KB	Python Source

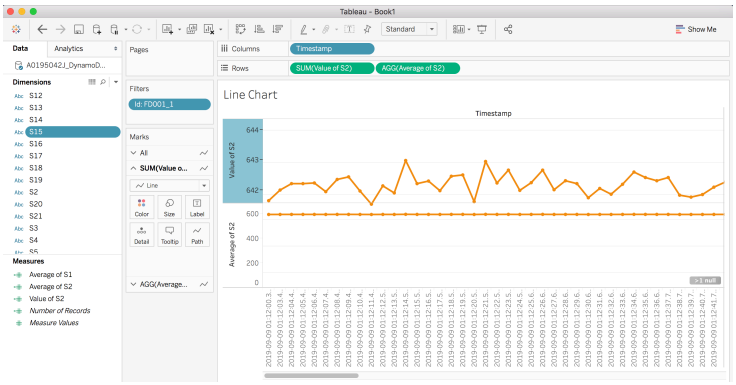
6.3) Run the script.
Due to process time issue, I only upload the first 5000 lines of data from engine1 and engine3 txt file. So the total size of DynamoDB table - **A0195042J_Engine**, will be 10000 lines.

```
33 with open('train_FD001.txt','r') as infile_1:
34   with open('FD001_out.txt','a') as outfile_1:
35     # Due to process time issue, only read the 1st 5000 lines of data
36     for line in infile_1.readlines()[0:5000]:
37       outfile_1.write(line)
38
39 with open('train_FD003.txt','r') as infile_2:
40   with open('FD003_out.txt','a') as outfile_2:
41     # Due to process time issue, only read the 1st 5000 lines of data
42     for line in infile_2.readlines()[0:5000]:
43       outfile_2.write(line)
```

6.4) Verify DynamoDB table - **A0195042J_Engine**



6.5) Visualization inside tableau
Taking the FD001, sensor 2 data – average vs distinct as an example



7. Task 2 – Publish rainfall data through raspberry pi to DynamoDB and do visualization

7.1) Background

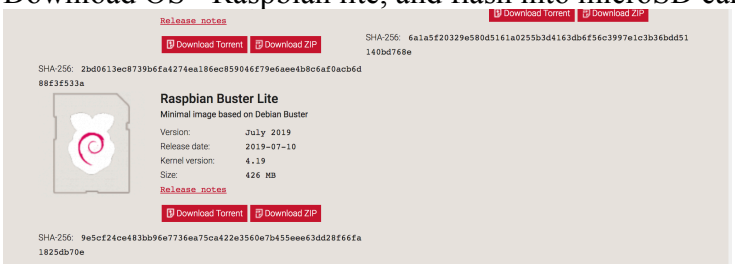
Prepare dataset by downloading the rain fall data - **rainfall-monthly-total.csv** from https://data.gov.sg/

7.2) Configure endpoint device – raspberry pi

Hardware preparation:

a. raspberry pi model 3 B
b. microSD card, 8GB
c. Keyboard
d. Monitor with HDMI cable
e. WiFi Dongle

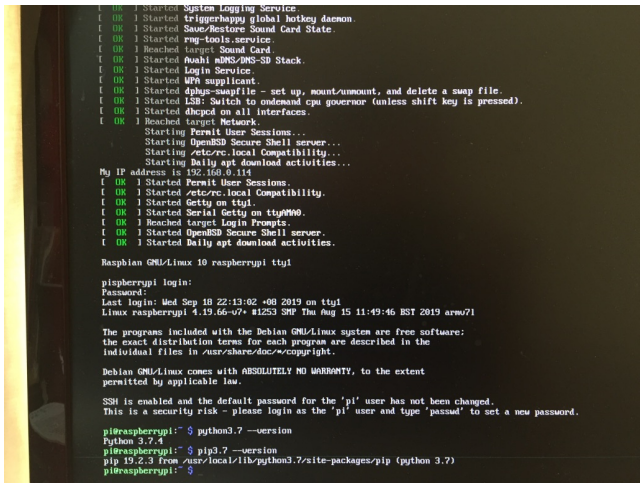
Download OS - Raspbian lite, and flash into microSD card:



f. Insert microSD card onto raspberry pi, power up and login using default credential:
   username: pi
   password: raspberry
g. Set up WiFi connection
   `sudo nano /etc/wpa_supplicant/wpa_supplicant.conf`
   network={
   ssid="WiFi name"
   psk="WiFi Password"
   }
   See details in link below:
   https://www.raspberrypi.org/documentation/configuration/wireless/wireless-cli.md
h. Auto-mount USB flash onto raspbian
   Follow steps in link below:
   https://www.raspberrypi-spy.co.uk/2014/05/how-to-mount-a-usb-flash-disk-on-the-raspberry-pi/

Note that, the USB flash is the median to store all the file (script, certificate, key, etc.) and to transfer into raspberry pi.

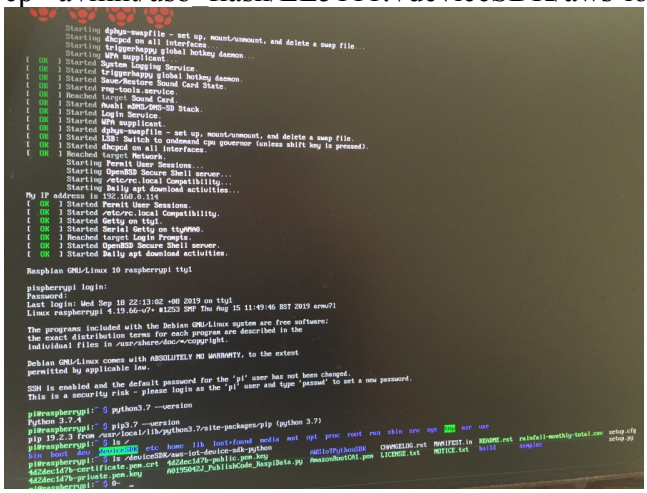
i. Download Python3 and Pip into raspberry pi
Follow steps in link below:
https://gist.github.com/SeppPenner/6a5a30ebc8f79936fa136c524417761d
Check the versions:



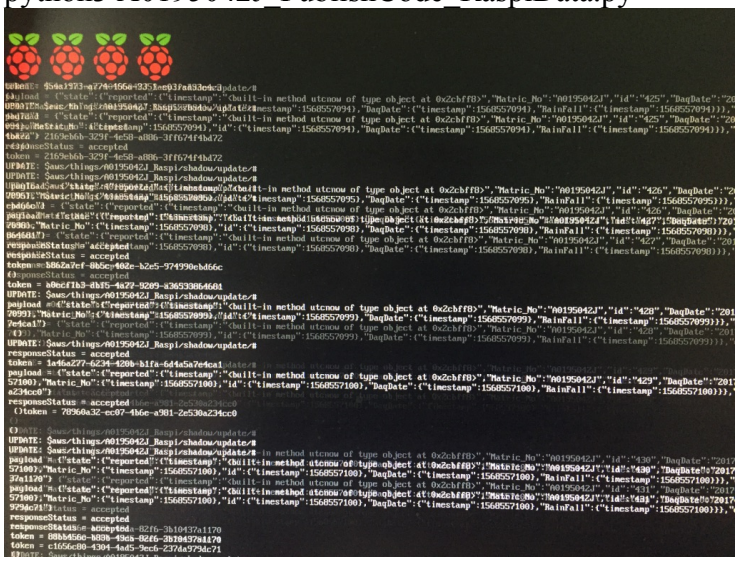
j. Download AWS_IoT_Device_Python_SDK into raspberry pi
Inside raspberry, follow steps (unix-like system) in link below:
https://docs.aws.amazon.com/greengrass/latest/developerguide/IoT-SDK.html
The SDK will be created into a folder named "aws-iot-device-sdk-python" under the root directory.

k. In order to get sufficient permission, create "deviceSDK" directory with both read and write access under raspberry root directory. Move the whole folder "aws-iot-device-sdk-python" to this "deviceSDK" folder

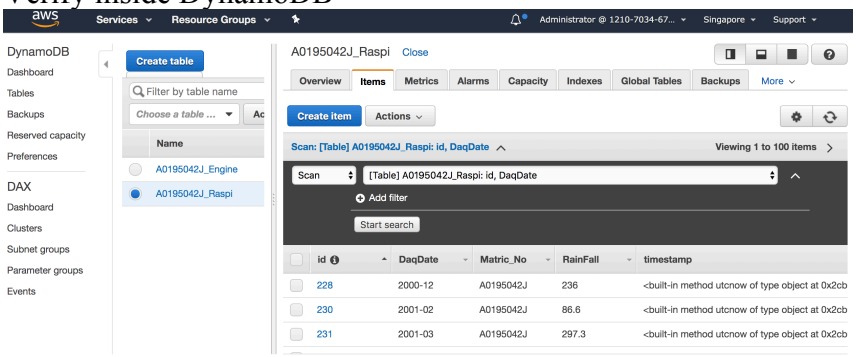
l. Copy the downloaded rainfall dataset, unique certificate, key, CA, and script - **A0195042J_PublishCode_RaspiData.py**, into folder (EE5111) within USB flash, connect the USB flash to raspberry, and copy all these files into the "deviceSDK" folder by command:
cp –a /mnt/usb_flash/EE5111. /deviceSDK/aws-iot-device-sdk-python



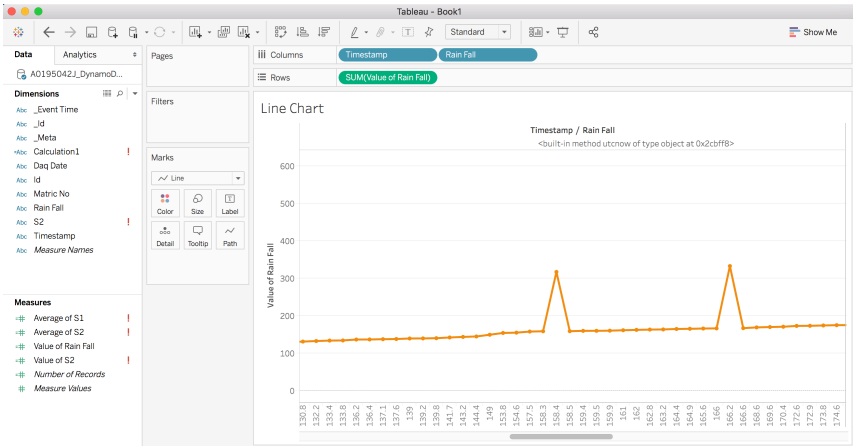
m. Run the script - **A0195042J_PublishCode_RaspiData.py**, and start publishing the rain fall data into DynamoDB table by command:
cd /deviceSDK/aws-iot-device-sdk-python
python3 A0195042J_PublishCode_RaspiData.py



7.3) Verify inside DynamoDB

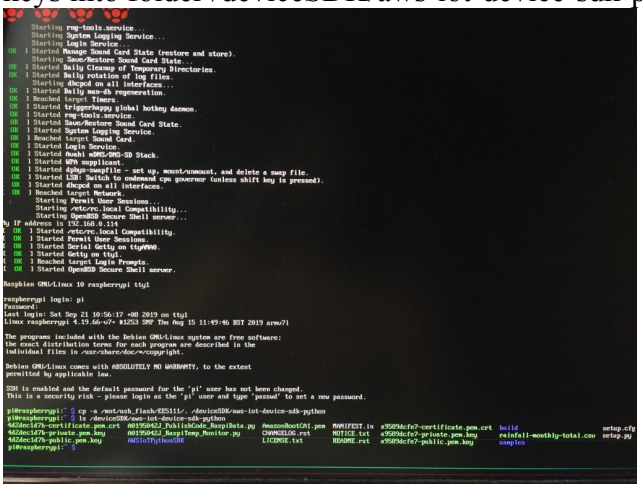


7.4) Visualization at Tableau



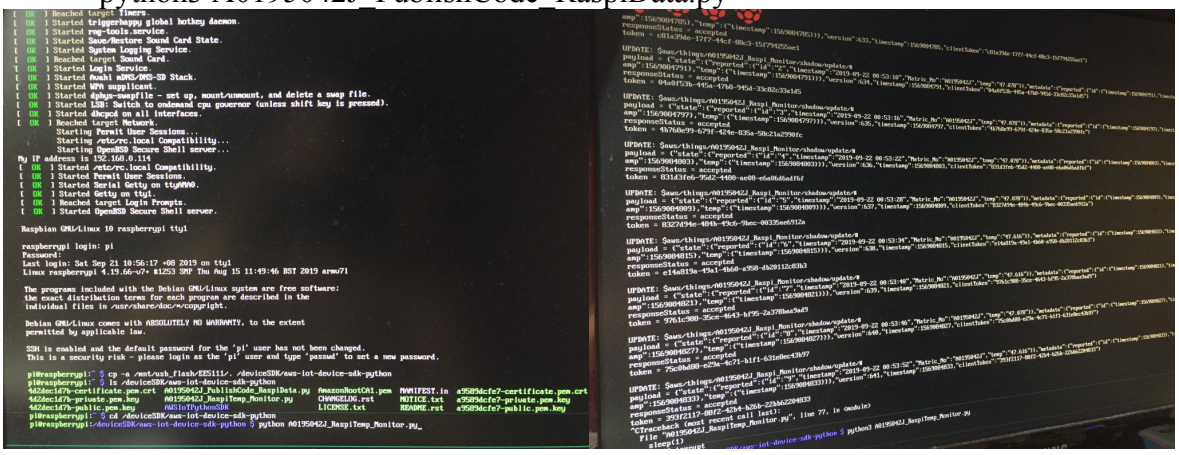
8. Task3 – Self-monitor raspberry pi's cpu temperature and establish email alert on abnormal high temperature (temperature > 47.00DegC)

8.1) Similar to 7.2)-l, copy the script – A0195042J_RaspiTemp_Monitor.py and certificate, keys into folder /deviceSDK/aws-iot-device-sdk-python

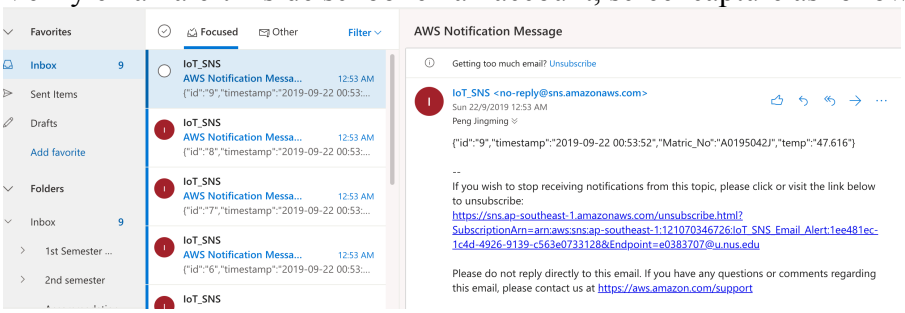


8.2) Install library – gpiozero by command:
sudo apt install python3-gpiozero

8.3) Direct to the folder and Run the script by command:
cd /deviceSDK/aws-iot-device-sdk-python
python3 A0195042J_PublishCode_RaspiData.py



8.4) Verify email alert inside school email account, screencapture as follow:



9. Discussion

The targeted AWS IoT functions (publish in and query out data from DynamoDB and establish SNS email alert) are realized by these 3 tasks.

There're more AWS services that could help to enhance user's IoT experience, and it's good worth of explore.

Beyond AWS platform, there're alternative IoT solution from other providers, such as Oracle, google, etc. Users are never limited within 1 solo IoT solution. However, cost, easiness, and effectiveness need to be carefully judged before deploying these technologies.

Another domain is about visualization. In this report, I am realizing it from tableau. AWS itself has similar solution, like QuickSight. To deploy this, no need to set up Rockset JDBC driver but have to go through S3 bucket. However, since it's not free, eventually I didn't choose it.

Tableau, as a powerful BI software, provides various ways of data visualization. And in this report, for illustration purpose, I only use very basic line-chart plot to show partial data visualization.

10. Appendixes:

Task1 Script – Publish Engine Data into DynamoDB table

```
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTShadowClient
import random, time, datetime

# The unique hostname (can be found in setting) AWS IoT generates for this device.
HOST_NAME = "a1loa9sgjgf5z0-ats.iot.ap-southeast-1.amazonaws.com"

# The relative path to the correct root CA file for AWS IoT,
ROOT_CA = "AmazonRootCA1.pem"

# A random programmatic shadow client ID for both Things (engine1 and engine3)
SHADOW_CLIENT_1 = "A0195042J_EngineFD001"
SHADOW_CLIENT_2 = "A0195042J_EngineFD003"

# The relative path to your private key file that AWS IoT generates for this
device
PRIVATE_KEY_1 = "ecf87423da-private.pem.key"
PRIVATE_KEY_2 = "ef8e4b716b-private.pem.key"

# The relative path to your certificate file that AWS IoT generates for this
device
CERT_FILE_1 = "ecf87423da-certificate.pem.crt"
CERT_FILE_2 = "ef8e4b716b-certificate.pem.crt"

# A programmatic shadow handler name prefix for both Things (engine1 and engine3)
SHADOW_HANDLER_1 = "A0195042J_EngineFD001"
SHADOW_HANDLER_2 = "A0195042J_EngineFD003"

# **************************************************
# Main script runs from here onwards.
# To stop running this script, press Ctrl+C.
# **************************************************

with open('train_FD001.txt','r') as infile_1:
  with open('FD001_out.txt','a') as outfile_1:
    # Due to process time issue, only read the 1st 5000 lines of data
    for line in infile_1.readlines()[0:5000]:
        outfile_1.write(line)

with open('train_FD003.txt','r') as infile_2:
  with open('FD003_out.txt','a') as outfile_2:
    # Due to process time issue, only read the 1st 5000 lines of data
    for line in infile_2.readlines()[0:5000]:
        outfile_2.write(line)

# Get the Data labels, later will be used as header of table inside DynamoDB
```

```
sensor_name = ['s'+ str(i) for i in range(1,22)]
dataLabels = ['id', 'timestamp', 'Matric_Number', 'cycle', 'os1', 'os2', 'os3'] +
    sensor_name

Matric_Number = 'A0195042J'

for i in range(0,len(dataLabels)):
    dataLabels[i] = '\"' + dataLabels[i] + '\"'

process_1 = open("FD001_out.txt",'r')
process_2 = open("FD003_out.txt",'r')

dataString_1 = []
dataString_2 = []
modifiedData_1 = []
modifiedData_2 = []

head = '{"state":{"reported":{'
tail = '}}}'

# Automatically called whenever the shadow is updated.
def myShadowUpdateCallback_1(payload, responseStatus, token):
  print()
  print('UPDATE: $aws/things/' + SHADOW_HANDLER_1 +
    '/shadow/update/#')
  print("payload = " + payload)
  print("responseStatus = " + responseStatus)
  print("token = " + token)

# Create, configure, and connect a shadow client.
myShadowClient_1 = AWSIoTMQTTShadowClient(SHADOW_CLIENT_1)
myShadowClient_1.configureEndpoint(HOST_NAME, 8883)
myShadowClient_1.configureCredentials(ROOT_CA, PRIVATE_KEY_1,
  CERT_FILE_1)
myShadowClient_1.configureConnectDisconnectTimeout(10)
myShadowClient_1.configureMQTTOperationTimeout(5)
myShadowClient_1.connect()

# Create a programmatic representation of the shadow.
myDeviceShadow_1 = myShadowClient_1.createShadowHandlerWithName(
  SHADOW_HANDLER_1, True)

for x in process_1.readlines():
      newData_1 = x.split(" ")
      modifiedData_1 = []
      modifiedData_1.append(str('FD001_' + newData_1[0]))
      modifiedData_1.append(str(datetime.datetime.utcnow()))
         modifiedData_1.append(Matric_Number)
```

```
        for j in range(2,len(sensor_name)):
          modifiedData_1.append(newData_1[j])

        ColumnLabels = []
        ColumnLabels.append(str(dataLabels[0] + ':'))
        ColumnLabels.append(str('"' + modifiedData_1[0] + '",'))
        ColumnLabels.append(str(dataLabels[1] + ':'))
        ColumnLabels.append(str('"' + str(datetime.datetime.now()) + '",'))
        ColumnLabels.append(str(dataLabels[2] + ':'))
        ColumnLabels.append(str('"' + Matric_Number + '",'))


        for i in range(3,len(dataLabels)):
          ColumnLabels.append(str(dataLabels[i] + ':'))
          ColumnLabels.append(str('"' + newData_1[i-2] + '",'))

        string = ''.join(ColumnLabels)
        string = string[:-1]

        data = []
        data.append(head)
        data.append(string)
        data.append(tail)
        data.append('\n')
        dataString_1 = ''.join(data)
        print(dataString_1)

        myDeviceShadow_1.shadowUpdate(dataString_1,myShadowUpdateCallback_1, 5)
        # read the data each 1s
        time.sleep(1)

# duplication for the second thing (engineFD003)
# Automatically called whenever the shadow is updated.
def myShadowUpdateCallback_2(payload, responseStatus, token):
  print()
  print('UPDATE: $aws/things/' + SHADOW_HANDLER_2 +
    '/shadow/update/#')
  print("payload = " + payload)
  print("responseStatus = " + responseStatus)
  print("token = " + token)

# Create, configure, and connect a shadow client.
myShadowClient_2 = AWSIoTMQTTShadowClient(SHADOW_CLIENT_2)
myShadowClient_2.configureEndpoint(HOST_NAME, 8883)
myShadowClient_2.configureCredentials(ROOT_CA, PRIVATE_KEY_2,
  CERT_FILE_2)
myShadowClient_2.configureConnectDisconnectTimeout(10)
myShadowClient_2.configureMQTTOperationTimeout(5)
```

```
149.   myShadowClient_2.connect()
150.
151.   # Create a programmatic representation of the shadow.
152.   myDeviceShadow_2 = myShadowClient_2.createShadowHandlerWithName(
153.     SHADOW_HANDLER_2, True)
154.
155.   for y in process_2.readlines():
156.         newData_2 = y.split(" ")
157.         modifiedData_2 = []
158.         modifiedData_2.append(str('FD003_' + newData_2[0]))
159.         modifiedData_2.append(str(datetime.datetime.utcnow()))
160.         modifiedData_2.append(Matric_Number)
161.         for k in range(2,len(sensor_name)):
162.           modifiedData_2.append(newData_2[k])
163.
164.         ColumnLabels = []
165.         ColumnLabels.append(str(dataLabels[0] + ':'))
166.         ColumnLabels.append(str('"' + modifiedData_2[0] + '",'))
167.         ColumnLabels.append(str(dataLabels[1] + ':'))
168.         ColumnLabels.append(str('"' + str(datetime.datetime.now()) + '",'))
169.         ColumnLabels.append(str(dataLabels[2] + ':'))
170.         ColumnLabels.append(str('"' + Matric_Number + '",'))
171.
172.         for l in range(3,len(dataLabels)):
173.           ColumnLabels.append(str(dataLabels[l] + ':'))
174.           ColumnLabels.append(str('"' + newData_2[l-2] + '",'))
175.
176.         string = ''.join(ColumnLabels)
177.         string = string[:-1]
178.
179.         data = []
180.         data.append(head)
181.         data.append(string)
182.         data.append(tail)
183.         data.append('\n')
184.         dataString_2 = ''.join(data)
185.         print(dataString_2)
186.
187.         myDeviceShadow_2.shadowUpdate(dataString_2,myShadowUpdateCallback_2, 5)
188.         # read the data each 1s
189.         time.sleep(1)
```

Task2 Script – Publish Raspi rainfall data into DynamoDB table

```
190.  from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTShadowClient
191.  from datetime import datetime
192.  import csv
193.  import os
194.  import json
195.
196.  # A random programmatic shadow client ID.
197.  SHADOW_CLIENT = "A0195042J_Raspi"
198.
199.  # The unique hostname that &IoT; generated for
200.  # this device.
201.  HOST_NAME = "a1loa9sgjgf5z0-ats.iot.ap-southeast-1.amazonaws.com"
202.
203.  # The relative path to the correct root CA file for &IoT;,
204.  # which you have already saved onto this device.
205.  ROOT_CA = "AmazonRootCA1.pem"
206.
207.  # The relative path to your private key file that
208.  # &IoT; generated for this device, which you
209.  # have already saved onto this device.
210.  PRIVATE_KEY = "4d2dec1d7b-private.pem.key"
211.
212.  # The relative path to your certificate file that
213.  # &IoT; generated for this device, which you
214.  # have already saved onto this device.
215.  CERT_FILE = "4d2dec1d7b-certificate.pem.crt"
216.
217.  # A programmatic shadow handler name prefix.
218.  SHADOW_HANDLER = "A0195042J_Raspi"
219.
220.  # Automatically called whenever the shadow is updated.
221.  def myShadowUpdateCallback(payload, responseStatus, token):
222.    print()
223.    print('UPDATE: $aws/things/' + SHADOW_HANDLER +
224.      '/shadow/update/#')
225.    print("payload = " + payload)
226.    print("responseStatus = " + responseStatus)
227.    print("token = " + token)
228.
229.  # Create, configure, and connect a shadow client.
230.  myShadowClient = AWSIoTMQTTShadowClient(SHADOW_CLIENT)
231.  myShadowClient.configureEndpoint(HOST_NAME, 8883)
232.  myShadowClient.configureCredentials(ROOT_CA, PRIVATE_KEY,
233.    CERT_FILE)
234.  myShadowClient.configureConnectDisconnectTimeout(10)
235.  myShadowClient.configureMQTTOperationTimeout(5)
```

```
236.   myShadowClient.connect()
237.
238.   # Create a programmatic representation of the shadow.
239.   myDeviceShadow = myShadowClient.createShadowHandlerWithName(
240.     SHADOW_HANDLER, True)
241.
242.   #object of CSV file
243.   f = 'rainfall-monthly-total.csv'
244.
245.   #Initialize weather data with 2 keyword: 'daq_time' & 'rain'
246.   WeatherData = {'daq_time':[], 'rain':[]}
247.
248.   #open rain fall csv file from directory (same directory where this script is
    stored)
249.   with open(os.path.join(os.getcwd(),f)) as csvfile:
250.       readCSV = csv.reader(csvfile,delimiter=',')
251.       #skip header
252.       next(readCSV)
253.
254.       for row in readCSV:
255.           WeatherData['daq_time'].append(row[0])
256.           WeatherData['rain'].append(row[1])
257.
258.   #length of data
259.   Length = len(WeatherData['daq_time'])
260.   #number of digit in Length
261.   Num = len(str(abs(Length)))
262.
263.   #previous time
264.   pt = datetime.now()
265.   #hold time = 1s
266.   delay = 1.0
267.
268.   Matric_Number = 'A0195042J'
269.
270.   for i in range(int(Length/2),Length):
271.       #current time
272.       ct = datetime.now()
273.       #timestamp
274.       timestamp = str(datetime.utcnow)
275.       id = str(i+1).zfill(Num)
276.       temp = {
277.           "state": {
278.               "reported":{
279.                   "id": id,
280.                   "timestamp": timestamp,
281.                   "Matric_No": Matric_Number,
282.                   "DaqDate": str(WeatherData['daq_time'][i]),
```

```
                    "RainFall": str(WeatherData['rain'][i])
                }
            }
        }

        #create Json string
        msg = json.dumps(temp)

        #publish Json message to AWS IoT device shadow
        myDeviceShadow.shadowUpdate(msg,myShadowUpdateCallback,5)
        #wait for 5s

        while ((ct-pt).total_seconds() <= delay):
            #update current time
            ct = datetime.now()
        #update previous time
        pt = ct

```

Task3 script – raspberry pi self-monitoring temperature data for email alert

```
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTShadowClient
from gpiozero import CPUTemperature
from datetime import datetime
from time import sleep, strftime, time
import os
import csv
import json


# A random programmatic shadow client ID.
SHADOW_CLIENT = "A0195042J_Raspi_Monitor"

# The unique hostname that &IoT; generated for
# this device.
HOST_NAME = "a1loa9sgjgf5z0-ats.iot.ap-southeast-1.amazonaws.com"

# The relative path to the correct root CA file for &IoT;,
# which you have already saved onto this device.
ROOT_CA = "AmazonRootCA1.pem"

# The relative path to your private key file that
# &IoT; generated for this device, which you
# have already saved onto this device.
PRIVATE_KEY = "a9589dcfe7-private.pem.key"

# The relative path to your certificate file that
# &IoT; generated for this device, which you
# have already saved onto this device.
```

```
CERT_FILE = "a9589dcfe7-certificate.pem.crt"

# A programmatic shadow handler name prefix.
SHADOW_HANDLER = "A0195042J_Raspi_Monitor"

# Automatically called whenever the shadow is updated.
def myShadowUpdateCallback(payload, responseStatus, token):
  print()
  print('UPDATE: $aws/things/' + SHADOW_HANDLER +
    '/shadow/update/#')
  print("payload = " + payload)
  print("responseStatus = " + responseStatus)
  print("token = " + token)

# Create, configure, and connect a shadow client.
myShadowClient = AWSIoTMQTTShadowClient(SHADOW_CLIENT)
myShadowClient.configureEndpoint(HOST_NAME, 8883)
myShadowClient.configureCredentials(ROOT_CA, PRIVATE_KEY,
  CERT_FILE)
myShadowClient.configureConnectDisconnectTimeout(10)
myShadowClient.configureMQTTOperationTimeout(5)
myShadowClient.connect()

# Create a programmatic representation of the shadow.
myDeviceShadow = myShadowClient.createShadowHandlerWithName(
  SHADOW_HANDLER, True)


# create object - cpu
cpu = CPUTemperature()

# initialize count
count = 0
# matric number
Matric_Number = 'A0195042J'

#create the log csv file
#with open('/deviceSDK/aws-iot-device-sdk-python/Raspi_Temp.csv','a') as log:
with open('Raspi_Temp.csv','a') as log:
  while count < 20:
    count = count + 1

    temp = cpu.temperature

    now = datetime.now()
    date_time = now.strftime('%Y-%m-%d %H:%M:%S')
    log.write('{0},{1}\n'.format(date_time,str(temp)))
    sleep(1)
```

```
msg = {
  "state": {
      "reported": {
          "id": str(count),
          "timestamp": date_time,
          "Matric_No": Matric_Number,
          "temp": str(temp)
      }
  }
}
myDeviceShadow.shadowUpdate(json.dumps(msg),myShadowUpdateCallback,5)
# rest for 5s before next data collection
sleep(5)
```