

Smart Parking IoT Solution based on AWS Web Services

Solutions Overview

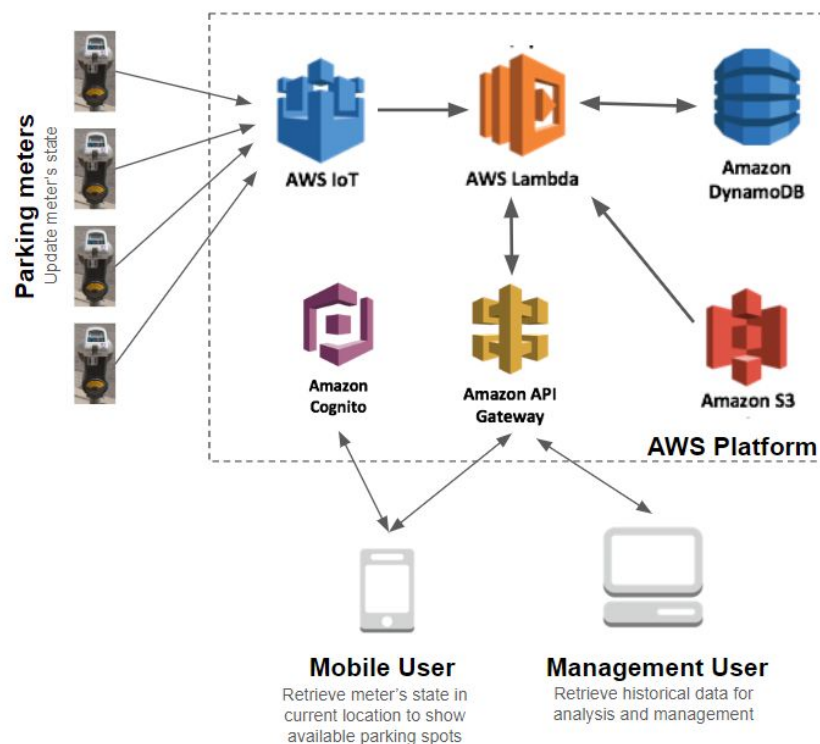
The city of Roundabout has decided to attack the problem associated with the street parking, where motorists struggle to find vacant parking spaces in the downtown area. They contacted AWS Professional Services for a solution using Amazon AWS Technologies.

Following is the Technical Solution proposal developed for this customer.

Architecture and Design description

The main idea of this solution consist on having the parking meters connected to the AWS Platform to update the current state of the parking slot. This detection is based on sensor technology that informs the meter when a car is parked or the spot is empty. With this information in the AWS Platform, the users (motorists) can use an Apps (from 3rd party developers) to query the platform for available spots in the current location. Additionally, the city parking management could gather useful information for management, analysis and control. Following is the proposed Architecture for the solution:

Smart Parking AWS Solution Architecture



In the above architecture diagram, the parking meters are connected to AWS IoT Core via an appropriate connectivity technology (see Connectivity technology options below), and sending an update to a mqtt topic including deviceId (unique device identifier), a timestamp and the state (Occupied/Free). Additional, other meter's parameters such as payment status, battery level, signal level (RSSI wireless signal indicator), etc., could also be sent for monitoring purposes. The deviceId is unique and it's associated with a location, address and meter number (any specific address may have many parking meters, so the number identify each one for that address). Since this association is in the datastore, there is no need to re-send for each packet minimizing transmission costs and power (shorter messages).

A Rule configured in AWS IoT Core will monitor this update topic and trigger a lambda function which will eventually update the information in the datastore, keeping the meter's information updated.

The datastore is implemented with DynamoDB so the solution can scale without problems. Since the intention is to provide motorists with information of available parking near to their current location, the datastore needs to process requests with the current location and a radius to indicate an area around this central point. For this particular geolocation problem, we use the Geo Library for DynamoDB (dynamodb-geo) which creates Geohash indexes for fast and efficient execution of location-based queries over DynamoDB items representing the parking meter location around the downtown.

There is an initial task to populate this datastore with the parking meter information and associated Geohashes, using a node.js code (***import.js*** - see appendices for the code) which imports the meter data from a json file provided by the city. This code can also be used to maintain (add or delete meters) in the datastore. These JSON files can be uploaded to S3 to maintain a reference, in case there is a need to re-create the datastore.

Once the datastore is in place, the ***parkingUpdate.js*** code (triggered by a Rule, as described above) will update the corresponding record to keep the meter state current (occupied/vacant). This code uses parameters such as deviceId, location, state and timestamp passed by the JSON file published by the device in the mqtt topic. The location is needed to calculate the hashKey (partition key of the table) while the deviceId is used as the rangeKey (sort.key).

An additional datastore can be created with the historical data for each device. This is implemented in DynamoDB with the deviceId as the primary partition-key and timestamp as sort-key. This datastore can be used for management or control purposes, providing an history of activity from each meter (every time was occupied and busy with the corresponding timestamp, as well as knowing if the user paid or not, while the car was occupying the spot).

At this point, the location datastore is up and current, so in order to retrieve the meters close to a specific location, a ***parkingQuery.js*** is used which uses location (latitude/longitude) and a Radius (in meters) to delimit the area of interest. This code is exposed through the API Gateway which is used by the Apps to retrieve the requested information.

In a typical scenario, a motorist looking for parking in the city, would use the Apps (from 3rd party developers) which will take the current location and suggest a Radius of search (the user can change this) and use Amazon Cognito to authenticate with API Gateway and send the request. This request is processed by API Gateway triggering and passing the parameters to ***parkingQuery.js*** which will retrieve the information. The Apps will use this information to show in a map, the available parking spots around the current location.

The following sections will provide detailed information for each specific topic of the solution.

Hardware/Sensor technology

In order to detect if the parking space is occupied (by a vehicle) or vacant, a sensor must be used. Different options exist with cost and accuracy differences;

- ***Magnetic sensor:*** it's a passive sensing technology that detects large ferrous objects (like a car) by measuring the change in the ambient magnetic field. These sensors are installed in the floor of the parking slot, so when a vehicle parks on it, it alters the magnetic field and the sensor detects those changes. Even when this type of sensor requires more labor for installation, accuracy is very good so it's one of the best options. Some sensors might have cellular connectivity so they can work independently of the meter, while others might have low power wireless (such as BLE) connectivity to connect them to the meter, being forwarded to the cloud using the meter cellular connectivity.



- ***Ultrasonic sensor:*** this sensor uses sound waves to detect objects. Although it's a better option to use indoor (mounted on the ceiling to detect a vehicle in the parking space below), it could be used as a cheap option mounted on the meter pole, or even better, in

the parking space floor; as in the first case, a person standing in front would block the sensor. This sensor is very cheap but accuracy is not as good as the magnetic sensor.



- *Video cameras* using object detection techniques could also be an option in the near future, as currently the cost of implementing could be much higher than the above options, while providing the same information (occupancy).

Connectivity technology options

Regarding connectivity suitable for this solution, following are the options;

- *Cellular data communication:* GSM modems for cellular (3G/2G) data communications are readily available and might be the best option considering most of the parking meters installed today are already using these channels for payment authorization. Unless the sensor includes this communication and operates independently, it would be convenient to connect the sensor to the meter to re-use this communication channel.
- *LoRA (Long-Range) communication:* as an option, this LPWAN technology is being used in many outdoor use cases, so it could be a good alternative to the cellular technology, in terms of cost and power consumption (as most of the meters operates from batteries with solar panels for rechargement).
- *Other technologies:* NB-IoT, LTE-M and 5G might be a good option in the near future.

Scalable Data Processing architecture

In order for this solution to deliver reliable performance at any scale, we chose DynamoDB - a fully managed, multi-region, multi-master database - as the datastore; and Lambda which allows to run code without provisioning or managing servers, paying only for the consumed compute time. AWS IoT Core service allows an easy and secure connection and management of billions of devices. API Gateway complements the solution allowing access to the data in a fast, reliable and secure way using Cognito service.

Geolocation technology

In order to support geospatial indexing on AWS DynamoDB datasets, we use Geo Library for Amazon DynamoDB, which takes care of managing Geohash indexes. Giving the library the user information (including localization) to populate into a table, the library will create a Geohash and Geohash-key and create the table with the appropriate setup, suitable for fast and efficient queries for records with location points close to the provided location.

Interface for 3rd Party Apps developers

The solution interfaces with 3rd Party Apps via AWS API Gateway, which uses Lambda to query and retrieve the requested information from the datastore. This datastore, the ***parkingMeters*** DynamoDB table, is configured to search the available parking meters in a radius and location specified in the Request. The Response will contain number, address and location of all available parking meters in the specified radius.

parkingMeters [Close](#)

Overview **Items** Metrics Alarms Capacity Indexes Global Tables Backups Triggers Access control Tags

Create item Actions

Scan: [Table] parkingMeters: hashKey, rangeKey Viewing 1 to 31 items

	hashKey	rangeKey	address	geoJson	geohash	isOccupied
<input type="checkbox"/>	-64	ABC115	3243 Bethany Loop, XYZ, AB	{"type":"POINT","coordinates":[[-146.2923],[-61.5722]]}	-6439579384445322285	false
<input type="checkbox"/>	-63	ABC109	15275 Elfrieda Street, XYZ, AB	{"type":"POINT","coordinates":[[-151.6866],[-48.8712]]}	-6398857278823319005	false
<input type="checkbox"/>	-63	ABC110	15275 Elfrieda Street, XYZ, AB	{"type":"POINT","coordinates":[[-151.6866],[-48.8712]]}	-6398857278823319005	false
<input type="checkbox"/>	-63	ABC111	15275 Elfrieda Street, XYZ, AB	{"type":"POINT","coordinates":[[-151.6866],[-48.8712]]}	-6398857278823319005	false
<input type="checkbox"/>	-63	ABC112	15275 Elfrieda Street, XYZ, AB	{"type":"POINT","coordinates":[[-151.6866],[-48.8712]]}	-6398857278823319005	false
<input type="checkbox"/>	-63	ABC113	15275 Elfrieda Street, XYZ, AB	{"type":"POINT","coordinates":[[-151.6866],[-48.8712]]}	-6398857278823319005	false
<input type="checkbox"/>	-51	ABC107	36590 Reanna Canyon, XYZ, AB	{"type":"POINT","coordinates":[[-4.1794],[-45.9565]]}	-5184789910831035103	false
<input type="checkbox"/>	-51	ABC108	36590 Reanna Canyon, XYZ, AB	{"type":"POINT","coordinates":[[-4.1794],[-45.9565]]}	-5184789910831035103	false
<input type="checkbox"/>	21	ABC104	093 Harris Parkway, XYZ, AB	{"type":"POINT","coordinates":[[25.9577],[-32.8645]]}	2189668027522342571	false

For security purposes, the access to the API Gateway requires AWS Credentials given to the App once it signs-in in AWS Cognito and exchanges tokens to get the credentials.

Access to Historical data

In order to keep track of the activity of a specific parking meter, a second DynamoDB table is created (***parkingActivity*** table), using the deviceId as partition key and the timestamp (provided in the JSON message of the update operation) as the sort key. Via API Gateway, after validating access through AWS Cognito, the user (mainly the parking system supervisor) can retrieve the requested information.

parkingActivity [Close](#)

Overview **Items** Metrics Alarms Capacity Index

[Create item](#) Actions ▾

Scan: [Table] parkingActivity: deviceId, timestamp ▾

<input type="checkbox"/>	deviceId ⓘ ▲	timestamp ▾	isOccupied ▾
<input type="checkbox"/>	ABC105	1538875399	true
<input type="checkbox"/>	ABC105	1538876200	false

Platform Security and Data Protection

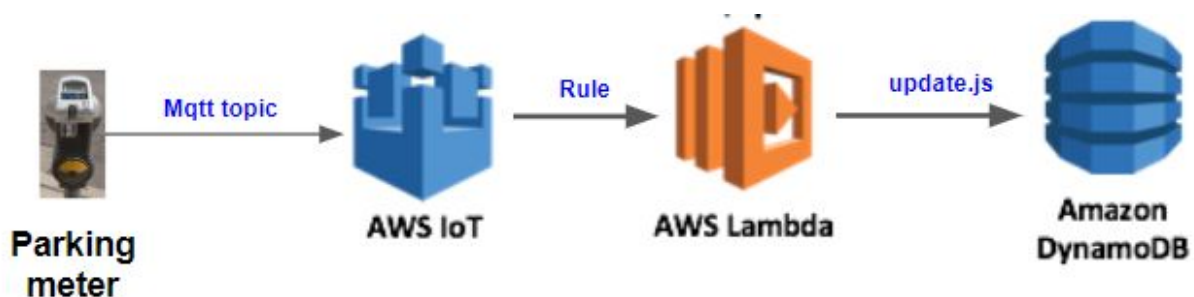
The security is ensured in the following ways;

- All communication from the parking meters to the AWS IoT Core is secure with TLS 1.2 (or greater) with mutual authentication (meaning the device - the client - needs to present it's digital certificate to the server to be validated).
- Communications between Mobile Apps and AWS API Gateway is via HTTPS (TLS), and Authentication is made via AWS Cognito.
- The Data Protection in the datastore can be achieved using encryption provided by DynamoDB.

Implementation details

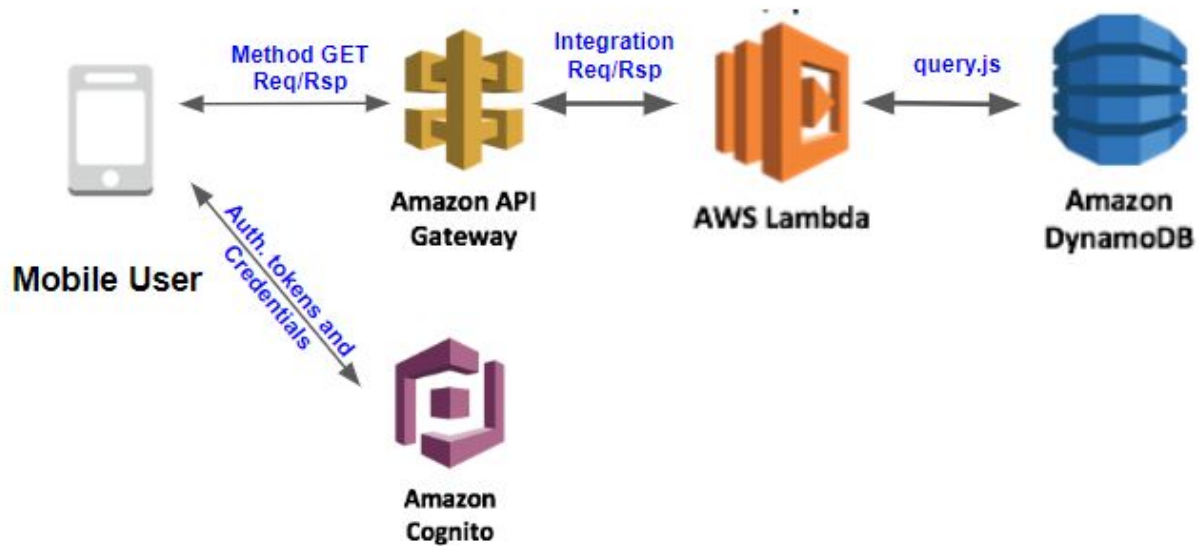
Once the datastores are setup, operations occur in two ways; *update* (from the parking meters) and *query* (from the users through the Mobile Apps).

Update operation:

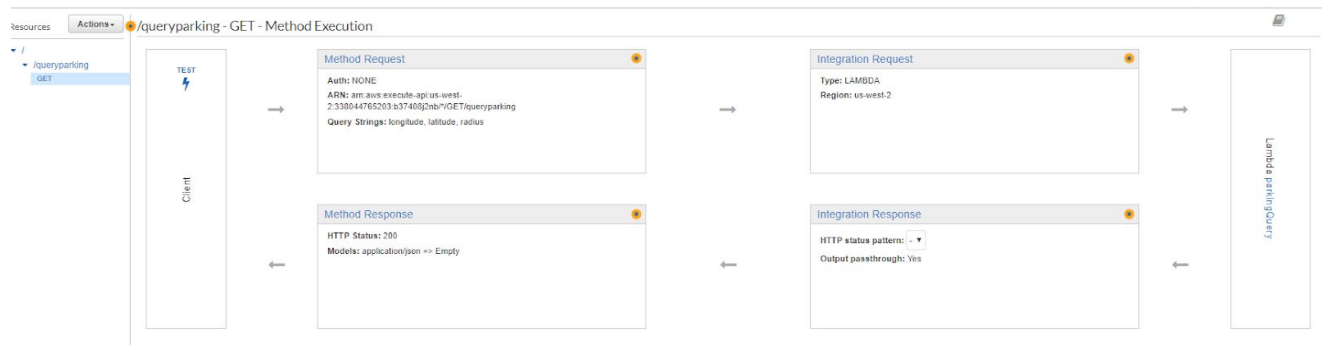


Every parking meter will publish an update to the '**parkingMeters/update**' mqtt topic, with the following JSON format;

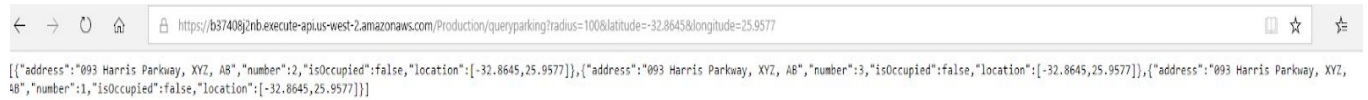
Query operation:



The Mobile Apps will sign-in with AWS Cognito to get the AWS Credentials necessary to access the AWS services. Once the access is granted, a HTTPS Request is issued to the API Gateway, including the Query parameters (current Location - latitude and longitude - with the Radius to consider in the search). The API Gateway will pass the parameters to the ***parkingQuery*** lambda function which after execution, will return the requested information. See appendices for the code of the ***query.js***. Following is the configuration of the API Gateway (note: in order to simplify the testing, authorization is not enabled in this demo).



Once API Gateway is configured and deployed, an access point (http:// address) is generated for public access. This can be tested from the web browser or used as the access point to the REST call from the application;



API Gateway documentation:

This service will provide information for all vacant parking spaces in a specified radius from a specified location (latitude, longitude).

Method: HTTPS GET

Path: Production/queryparking

Parameters (query string):

- Latitude:
- Longitude
- Radius (in meters)

Example:

<https://b37408j2nb.execute-api.us-west-2.amazonaws.com/Production/queryparking?radius=100&latitude=-32.8645&longitude=25.9577>

Possible Enhancements to the Solution

An interesting enhancement could be consider if the parking sensors and logic are integrated with the parking meter payment system, as in this case it would be possible to detect when the parking slot is being used and checking if payment is valid. Otherwise, a notification could be issued to the parking enforcement authorities to notify for this situation.

Appendix A - Code implemented for *import.js*

```
1  const ddbGeo = require('dynamodb-geo');
2  const AWS = require('aws-sdk');
3
4  // Set up AWS
5  AWS.config.update({
6    region: "us-west-2"
7  });
8
9  // Point to the DB for the example.
10 const ddb = new AWS.DynamoDB({ endpoint: new AWS.Endpoint('https://dynamodb.us-west-2.amazonaws.com') });
11
12 // Configuration for a new instance of a GeoDataManager. Each GeoDataManager instance represents a table
13 const config = new ddbGeo.GeoDataManagerConfiguration(ddb, 'parkingMeters');
14 //config.hashKeyLength = 6;
15
16 // Instantiate the table manager
17 const geoManager = new ddbGeo.GeoDataManager(config);
18
19 // Use GeoTableUtil to help construct a CreateTableInput.
20 const createTableInput = ddbGeo.GeoTableUtil.getCreateTableRequest(config);
21
22 // Tweak the schema as desired
23 createTableInput.ProvisionedThroughput.ReadCapacityUnits = 2;
24
25 console.log('Creating table with schema:');
26 console.dir(createTableInput, { depth: null });
27
28 // Create the table
29 ddb.createTable(createTableInput).promise()
30   // Wait for it to become ready
31   .then(function () { return ddb.waitFor('tableExists', { TableName: config.tableName }).promise() })
32   // Load sample data in batches
33   .then(function () {
34     console.log('Loading parking meter information');
35     const data = require('./parking_meters.json');
36     const putPointInputs = data.map(function (meter) {
37       return {
38         //RangeKeyValue: { S: uuid.v4() }, // Use this to ensure uniqueness of the hash/range pairs.
39         RangeKeyValue: { S: meter.deviceID },
40         GeoPoint: {
41           latitude: meter.meter.location[0],
42           longitude: meter.meter.location[1]
43         },
44         PutItemInput: {
45           Item: {
46             number: { N: meter.meter.number.toString() },
47             address: { S: meter.meter.address },
48             location: { NS: meter.meter.location },
49             isOccupied: { BOOL: meter.isOccupied }
50           }
51         }
52       }
53     });
54   });
```

```

54
55     const BATCH_SIZE = 25;
56     const WAIT_BETWEEN_BATCHES_MS = 1000;
57     var currentBatch = 1;
58
59     function resumeWriting() {
60         if (putPointInputs.length === 0) {
61             return Promise.resolve();
62         }
63         const thisBatch = [];
64         for (var i = 0, itemToAdd = null; i < BATCH_SIZE && (itemToAdd = putPointInputs.shift()); i++) {
65             thisBatch.push(itemToAdd);
66         }
67         console.log('Writing batch ' + (currentBatch++) + '/' + Math.ceil(data.length / BATCH_SIZE));
68         return geoManager.batchWritePoints(thisBatch).promise()
69             .then(function () {
70                 return new Promise(function (resolve) {
71                     setInterval(resolve, WAIT_BETWEEN_BATCHES_MS);
72                 });
73             })
74             .then(function () {
75                 return resumeWriting()
76             });
77     }
78
79     return resumeWriting().catch(function (error) {
80         console.warn(error);
81     });
82
83 })
84 .catch(console.warn)
85
86 .then(function () {
87     process.exit(0);
88 });

```

Appendix B - Code implemented for *update.js*

```
1  const ddbGeo = require('dynamodb-geo');
2  const AWS = require('aws-sdk');
3
4  // Set up AWS
5  AWS.config.update({
6    region: "us-west-2"
7  });
8
9  // Point to the DB for the example.
10 const ddb = new AWS.DynamoDB({ endpoint: new AWS.Endpoint('https://dynamodb.us-west-2.amazonaws.com') });
11
12 // Configuration for a new instance of a GeoDataManager. Each GeoDataManager instance represents a table
13 const config = new ddbGeo.GeoDataManagerConfiguration(ddb, 'parkingMeters');
14
15 // Instantiate the table manager
16 const geoManager = new ddbGeo.GeoDataManager(config);
17
18 exports.handler = function(event, context, callback) {
19
20   geoManager.updatePoint({
21     RangeKeyValue: { S: event.deviceID },
22     GeoPoint: { // An object specifying latitude and longitude as plain numbers.
23       latitude: event.latitude,
24       longitude: event.longitude
25     },
26     UpdateItemInput: { // TableName and Key are filled in for you
27       UpdateExpression: 'SET isOccupied = :newName',
28       ExpressionAttributeValues: {
29         ':newName': { BOOL: event.isOccupied }
30       }
31     }
32   }).promise()
33   .then(function() { |
34     console.log('Done!')
35     callback(null, 'Done!');
36   });
37 }
```

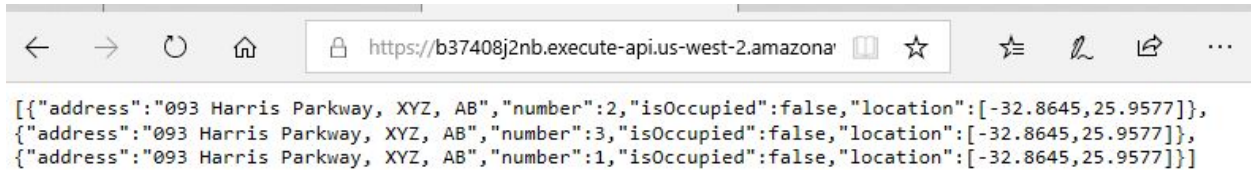
Appendix C - Code implemented for *query.js*

```
1  const ddbGeo = require('dynamodb-geo');
2  const AWS = require('aws-sdk');
3
4  // Set up AWS
5  AWS.config.update({
6    region: "us-west-2"
7  });
8
9  // Point to the DB for the example.
10 const ddb = new AWS.DynamoDB({ endpoint: new AWS.Endpoint('https://dynamodb.us-west-2.amazonaws.com') });
11
12 // Configuration for a new instance of a GeoDataManager. Each GeoDataManager instance represents a table
13 const config = new ddbGeo.GeoDataManagerConfiguration(ddb, 'parkingMeters');
14
15 // Instantiate the table manager
16 const geoManager = new ddbGeo.GeoDataManager(config);
17
18 exports.handler = function(event, context, callback) {
19
20   const parameters = {
21     RadiusInMeter: parseFloat(event.params.querystring.radius),
22     CenterPoint: {
23       latitude: parseFloat(event.params.querystring.latitude),
24       longitude: parseFloat(event.params.querystring.longitude)
25     }
26   };
27
28   geoManager.queryRadius(parameters)
29   .then( points => {
30     var result = Array();
31     points.forEach(function (record) {
32
33       var unmarshalled = AWS.DynamoDB.Converter.unmarshall(record);
34       if (unmarshalled.isOccupied == false) {
35         var location = unmarshalled.location.values;
36         var meter = '{"address": "' + unmarshalled.address + '", "number": ' + unmarshalled.number
37         + ', "isOccupied": ' + unmarshalled.isOccupied + ', "location": [' + location + ']}';
38         var res = JSON.parse(meter);
39         result.push(res);
40       }
41     })
42   })
43
44   console.log(JSON.stringify(result));
45   callback(null, result);
46 });
47 }
```

Appendix D - Update & Query operations testing

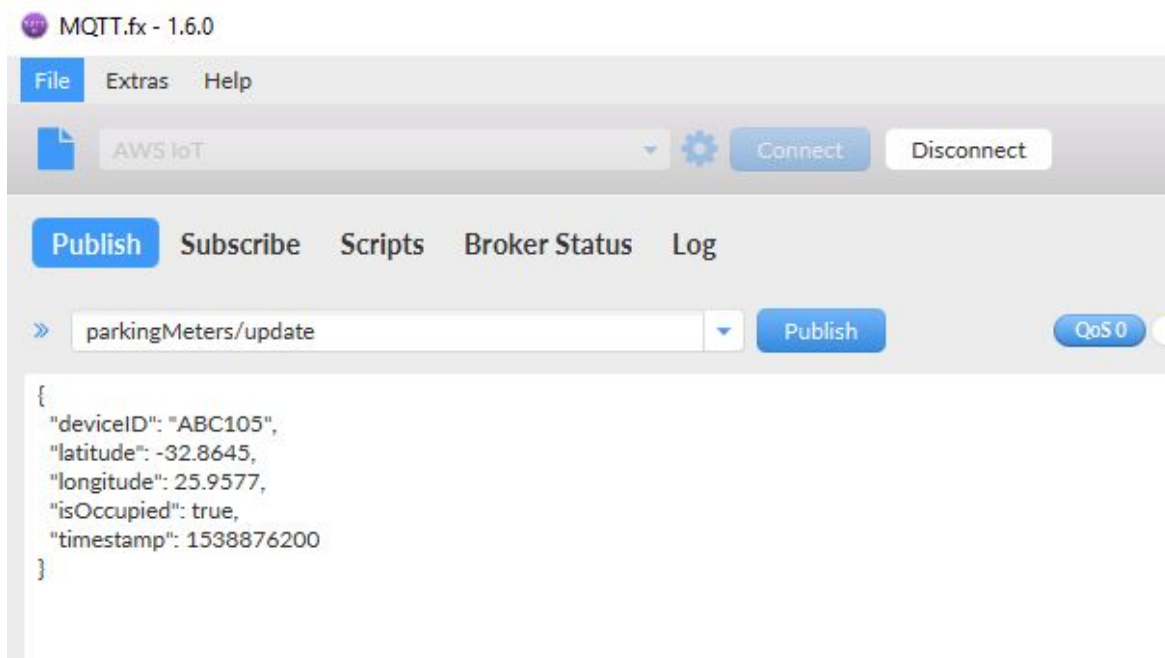
The implemented solution was tested querying the vacant parking spaces for a particular location/radius before and after an update operation in one of the parking meters in this area.

1 - Query operation for location (-32.8645,25.9577) and radius of 100 meters: this brings 3 vacant parking spaces;

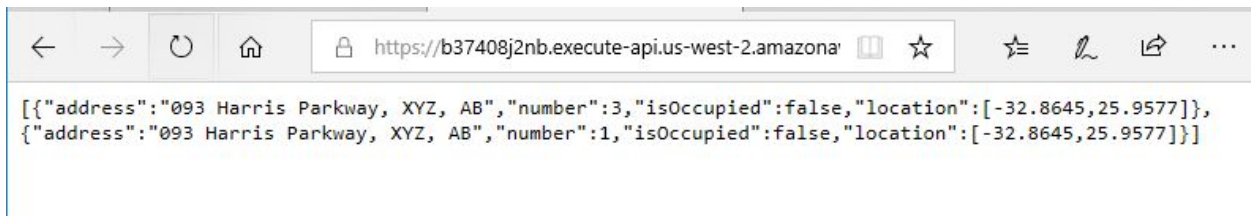


```
[{"address": "093 Harris Parkway, XYZ, AB", "number": 2, "isOccupied": false, "location": [-32.8645, 25.9577]}, {"address": "093 Harris Parkway, XYZ, AB", "number": 3, "isOccupied": false, "location": [-32.8645, 25.9577]}, {"address": "093 Harris Parkway, XYZ, AB", "number": 1, "isOccupied": false, "location": [-32.8645, 25.9577]}]
```

2 - Update operation simulated with MQTT.fx utility, declaring deviceID = ABC105 (corresponding to parking meter number 2 of location (-32.8645,25.9577) , address "093 Harris Parkway, XYZ, AB") changed its state from vacant to occupied;




3 - Refreshing the browser, now we get only two vacant parking spaces (number 2 is now occupied);



```
[{"address": "093 Harris Parkway, XYZ, AB", "number": 3, "isOccupied": false, "location": [-32.8645, 25.9577]}, {"address": "093 Harris Parkway, XYZ, AB", "number": 1, "isOccupied": false, "location": [-32.8645, 25.9577]}]
```


Appendix E - JSON file format used with *import.js*

There are 31 parking meters located in 10 addresses.

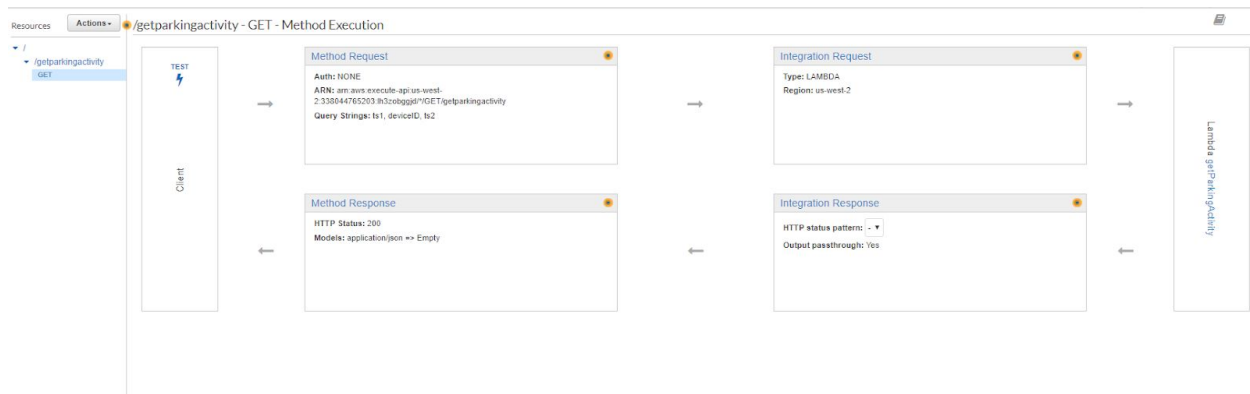
 parking_meters.json - Notepad

File Edit Format View Help

```
[
  {
    "deviceID": "ABC101",
    "isOccupied": false,
    "meter": {
      "number": 1,
      "location": ["-12.8557", "-165.1071"],
      "address": "190 Seth Ways, XYZ, A"
    }
  },
  {
    "deviceID": "ABC102",
    "isOccupied": false,
    "meter": {
      "number": 1,
      "location": ["-20.3352", "-177.8691"],
      "address": "44712 Rau Loaf, XYZ, AB"
    }
  },
  {
    "deviceID": "ABC103",
    "isOccupied": false,
    "meter": {
      "number": 2,
      "location": ["-20.3352", "-177.8691"],
      "address": "44712 Rau Loaf, XYZ, AB"
    }
  },
  {
    "deviceID": "ABC104",
    "isOccupied": false,
    "meter": {
      "number": 1,
      "location": ["-32.8645", "25.9577"],
      "address": "093 Harris Parkway, XYZ, AB"
    }
  },
  {
    "deviceID": "ABC105",
    "isOccupied": false,
    "meter": {
      "number": 2,
      "location": ["-32.8645", "25.9577"],
      "address": "093 Harris Parkway, XYZ, AB"
    }
  },
  ],
```

<

Appendix F - API Gateway for Meter Activity information



```
index.js
1 var AWS = require('aws-sdk');
2 var dynamo = new AWS.DynamoDB.DocumentClient();
3 var table = "parkingActivity";
4
5 exports.handler = function(event, context, callback) {
6
7   var params = {
8     TableName: "parkingActivity",
9     KeyConditionExpression: "#dev = :dev AND #ts BETWEEN :t1 AND :t2",
10    ExpressionAttributeNames: {
11      '#dev': 'deviceId',
12      '#ts': 'timestamp'
13    },
14    ExpressionAttributeValues: {
15      ':dev': event.params.querystring.deviceID,
16      ':t1': parseInt(event.params.querystring.ts1, 10),
17      ':t2': parseInt(event.params.querystring.ts2, 10)
18    }
19  };
20
21  dynamo.query(params, function(err, data) {
22    if (err) {
23      console.error("Unable to query. Error:", JSON.stringify(err, null, 2));
24    } else {
25      console.log("Query succeeded: " + JSON.stringify(data));
26      callback(null, data);
27    }
28  });
29 }
```

API Gateway documentation:

This service will provide activity information for a specified device, in a specified timeframe.

Method: HTTPS GET

Path: Production/getparkingactivity

Parameters (query string):

- deviceID:
- ts1: (Timestamp1)
- ts2: (Timestamp2)

Example:

<https://lh3zobggjd.execute-api.us-west-2.amazonaws.com/Production/getparkingactivity?deviceID=ABC105&ts1=1538875900&ts2=1538975930>

Test result;

```
{
  "Items": [
    {
      "isOccupied": false,
      "deviceID": "ABC105",
      "timestamp": 1538875900
    },
    {
      "isOccupied": true,
      "deviceID": "ABC105",
      "timestamp": 1538876200
    },
    {
      "isOccupied": false,
      "deviceID": "ABC105",
      "timestamp": 1538975930
    }
  ],
  "Count": 3,
  "ScannedCount": 3
}
```

Datastore for activity;

parkingActivity [Close](#)

Overview **Items** Metrics Alarms Capacity

[Create item](#) [Actions](#)

Scan: [Table] parkingActivity: deviceID, timestamp

<input type="checkbox"/>	deviceID	timestamp	isOccupied
<input type="checkbox"/>	ABC105	1538875399	true
<input type="checkbox"/>	ABC105	1538875900	false
<input type="checkbox"/>	ABC105	1538876200	true
<input checked="" type="checkbox"/>	ABC105	1538975930	false