NXP IoT – Weather Station Developer Guide

Contents

1. Introduction

NXP's Rapid IoT prototyping kit is a comprehensive, secure and optimized IoT end node solution with a user-friendly development environment that enables anyone to quickly take their idea to a proof-of-concept. Its architecture is built upon two controllers:

- Kinetis K64F for the main application, powered by an ARM® Cortex®-M4 core
- Kinetis KW41Z for wireless connectivity, powered by an ARM® Cortex®-M0+ core

It includes:

- 10-axis motion sensing, thanks to a combo accelerometer / magneto-meter
- Gyroscope
- Pressure sensor for altitude measurement
- Environmental sensing via temperature/humidity, ambient light and air quality sensors
- Display capabilities with low-power color screen
- Authentication, identification
- User interfaces with LEDs, buzzer and touch plus push buttons
- Additional memory for data storage
- Rechargeable battery

The factory application includes USB and Bluetooth/Thread bootloaders to program your own firmware without external tool, and several IoT

application use-cases leveraging components on board.

In order to allow any user to evaluate this product, NXP has created the “NXP IoT – Weather station” applications. This is a set of applications consisting of a mobile application (available in Android and iOS), a cloud application and a demo FW for the Rapid IoT device.

The usage of the mobile and cloud applications is described in detail in the User Guide document. The main purpose of this document is to guide a developer through the mobile and cloud applications in order to being able to retrieve data from the sensors, sent it to the cloud application, process and plot it in graphs.

Before using the Rapid IoT devices with the provided applications, it is necessary to flash them with the corresponding firmware. The instructions to flash are:

- **Connect** one end of the provided **USB cable to the computer** and the other end to the micro USB type-B **connector of the SLN-RPK-NODE**
- Keep **SW3 button pressed** while briefly pushing **SW5/Reset** button,
- Wait 1-2s for RGB LED to blink Green then **release SW3** button

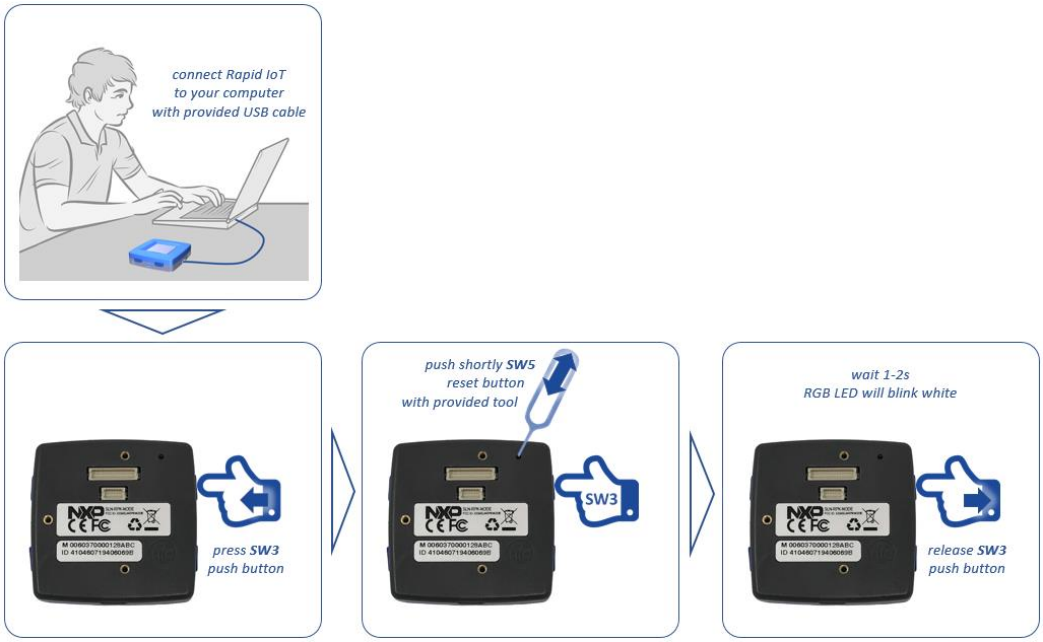

Figure 1. Instructions for USB Mass Storage Device Programming

RGB LED will blink green and your computer will detect a new Mass Storage drive and automatically install the appropriate drivers.

- From your computer file explorer, drag-n-drop or copy-paste into the Mass Storage drive the Weather station demo binary file.

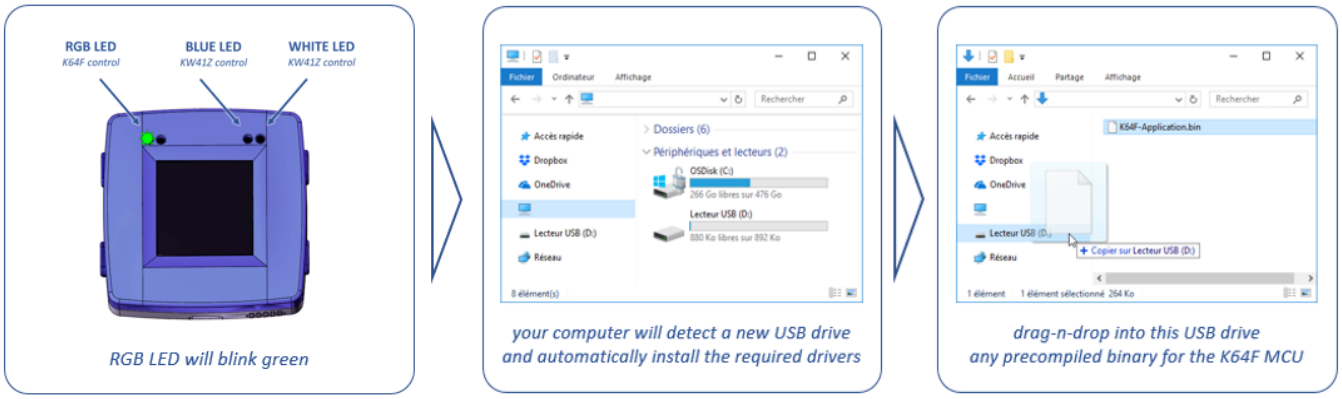

Figure 2. Instructions for pushing a new application through USB

Bootloader will automatically identify the MCU target to reprogram, thanks to the binary file signature.
RGB LED will **blink purple** during download and **blink blue** during serial flash programming.
RGB LED will **blink green** during K64F internal flash (re)programming with the new application (read from Serial Flash memory) and automatically reset, when ready.

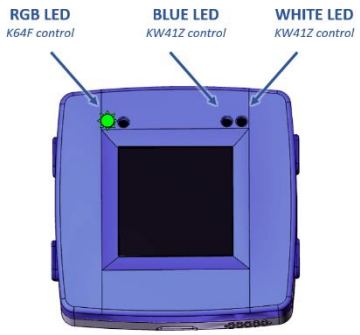

Figure 3. USB Programming LED

RGB LED	BLUE LED	WHITE LED
PURPLE BLINK	OFF	OFF
BLUE BLINK	OFF	OFF
GREEN BLINK	OFF	OFF

Table 1. K64F USB Programming LED Sequence

More information about the Rapid IoT kit can be found in NXP's website:
https://www.nxp.com/support/developer-resources/rapid-prototyping/nxp-rapid-iot-prototyping-kit:IOT-PROTOTYPING

2. Mobile applications

The Weather Station platform relies on a mobile application that can be used to connect to the RapidIoT kit, extract the data and post it to the Cloud. This application is developed for mobile phones

supporting both Android and iOS operative systems. In the present section we will explain how these applications were implemented and describe the relevant parts of the code so other developers can use it as a guide to help them build their systems based in the RapidIoT.

The source code of both versions of the mobile application can be found in the following links:

- Android app: https://bitbucket.org/mobileknowledge/weather_station_demo-androidapp/src/master/
- iOS app: https://bitbucket.org/mobileknowledge/weather_station_demo-iosapp/src/master/

Development setup

2.1.1. Android application

For the Android application development, Android studio version 3.2.1 has been used as IDE. This IDE is widely used for Android development and has extensive documentation available in the official Android website. Link for download: https://developer.android.com/studio/ (latest version 3.2.1 at the time this developer guide was written).

The application is programmed in Java programming language.

2.1.2. iOS application

For the iOS application development, XCode version 10.1 has been used as IDE. In order to download this IDE, please go to App Store in MAC OS and type “XCode”. The iOS application has been created using Swift 4 as programming language.

Applications architecture

2.2.1. Android and iOS application

The mobile applications are built in a layered structure, following the “Model-Business-Presentation” design pattern. A simple diagram of the relationship between layers can be found in Figure 4.

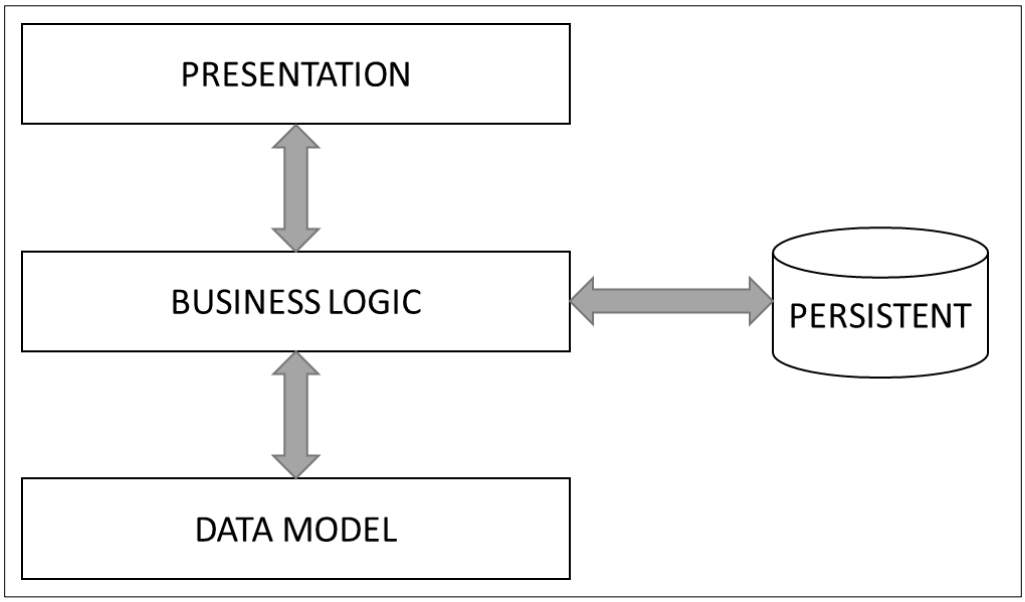

Figure 4. Mobile applications architecture

In order to have a consistent structure for the Android and iOS source codes, the packages structure in both applications should be the following:

- Presentation: This package contains the files related to the UI, i.e., Activities (Android) and View Controllers (iOS).
- Business Logic: This package contains the files related to the application logic: BLE connectivity, Internet connectivity, JSON generation, Data validation, etc.
- Data model: This package contains the files related to the data model, i.e., Device class or Measurement class.
- Persistent: This package will contain the files related to the persistent data storage, i.e., shared preferences (Android), UserDefaults (iOS).

Implementation of the main functional blocks

For the sake of a clear structure and a practical use of the present document, we will analyze the applications code from a functional point of view. This way, we will address the operations that should be interesting for a developer willing to implement an application to obtain data from the RapidIoT device and use it in a backend system.

2.3.1. Connecting to the RapidIoT

2.3.1.1. RapidIoT BLE configuration

The RapidIoT BLE interface is configured to provide the user with a set of BLE services and characteristics to ease access to the information retrieved from the sensors integrated in the device. Each sensor in the device has its own BLE characteristic where the microcontroller posts the information, so the user can filter the information he will be notified about.

The RapidIoT follow the UUID characteristics convention for the main Bluetooth sensor types that are included in the Bluetooth specifications. In the following table, you can find the UUID used for the service and characteristics used.

Name	UUID	Properties
Service	0ab5b670-c2ce-c4ab-e711-6ccbaa65c888	
Pressure characteristic	00002A6D-0000-1000-8000-00805F9B34FB	NOTIFY, READ
Temperature characteristic	00002A1C-0000-1000-8000-00805F9B34FB	NOTIFY, READ
Humidity characteristic	00002A6F-0000-1000-8000-00805F9B34FB	NOTIFY, READ
Light characteristic	0ab5b672-c2ce-c4ab-e711-6ccbaa65c888	NOTIFY, READ
MAC Address characteristic	0e1bb826-612e-aca9-0e49-88f712ec1cb7	READ

The pressure, temperature and humidity data use the UUIDs defined in the Bluetooth specifications while the light sensor information is posted to a non-standard UUID. The sensors characteristics have Notify and Read properties so the user can read one data asynchronously or 'subscribe' to the characteristic and receive updates with the sensor information. The RapidIoT kit also includes a characteristic to obtain the device MAC address.

2.3.2. Collecting data from the sensors

2.3.2.1. Implementation in Android

The Android application implements a Singleton class called WSDBleManager to manage the connection to the RapidIoT device. This class includes methods to scan for devices in range, trigger the connection/disconnection to a certain device, or the definition of the BleManager callbacks for Android. The WSDBleManager makes use of a third party library that is included in the dependencies section from the build.gradle file of the app. This way, the WSDBleManager extends the BleManager and BleManagerCallbacks from this library:

```
public class WSDBleManager extends BleManager<BleManagerCallbacks> {
```

Figure 5. 'WSDBleManager' class header in Android app

The 'BleManagerCallbacks' includes methods that are called depending on the state that the connection goes through. We can highlight the following methods:

```
void onDeviceConnected(final BluetoothDevice device);

void onDeviceDisconnected(final BluetoothDevice device);

void onDeviceReady(final BluetoothDevice device);

void onBondingRequired(final BluetoothDevice device);

void onBonded(final BluetoothDevice device);
```

Figure 6. ‘BLEManagerCallbacks’ in Android app

The ‘BleManagerCallbacks’ can be overwritten at any moment by using the method ‘setGattCallbacks’ from the ‘BleManager’ class.

Scanning for devices

The method used to start scanning for devices is called ‘scanAvailableDevices’. This method is present in the ‘WSDBleManager’ and requires as input an object that implements the ‘WSDBleScanListerner’ interface. This interface defines two methods: the ‘onDeviceScanned’ and the ‘onDeviceScanTimeout’. The first will be called when a new device is discovered and the second when the scanning has reached its timeout. They are used in the main activity to show the newly discovered devices to the user.

```
public void scanAvailableDevices(WSDBleScanListener wsdBleScanListener) {..}
```

Figure 7. ‘scanAvailableDevices’ method in Android

In the ‘scanAvailableDevices’ method we configure the settings that we are going to use for the scanning and include a filter so we will only look for devices with our Service UUID:

```
final ScanSettings scanSettings = new ScanSettings.Builder()
        .setScanMode(ScanSettings.SCAN_MODE_BALANCED)
        // Refresh the devices list continuously
        .setReportDelay(0)
        // Hardware filtering has some issues on selected devices
        .setUseHardwareBatchingIfSupported(false)
        .build();

ScanFilter scanFilterDES = new ScanFilter.Builder()
        .setServiceUuid(ParcelUuid.fromString(WSDemoUUID.toString()))
        .build();
```

Figure 8. Set scan settings in Android app

Once the settings and filter are configured, it starts scanning:

```
bluetoothLeScanner.startScan(filterList, scanSettings, scanCallback);
```

Figure 9. Start scan in Android app

The ‘WSDBleManager’ holds a list with the devices that have been scanned in the ArrayList ‘scanListDevices’. Whenever a device is scanned, we check that it has not been discovered previously and then process it as a new one, calling the ‘onDeviceScanned’ method from the ‘WSDBleScanListerner’.

Connecting to a device

When the user selects one of the devices from the list, we trigger the connection to that device by calling the ‘initConnection’ from the ‘WSDBleManager’:

```
showConnectingDialog();
getBLEManager().initConnection (deviceClicked);
getBLEManager().stopScan();
```

Figure 10. Start a connection to a device in the Android app

The ‘initConnection’ method needs as input an object of the ‘DeviceModel’ type. This object holds information that will be used by the underlying library to identify the device we want to connect to.

```
public void initConnection(DeviceModel bleDevice) {
    List<ParcelUuid> serviceUuidList = bleDevice.getServiceUuidList();
    mDevice = bleDevice;
    for(int i=0; i< serviceUuidList.size(); i++){
        if(serviceUuidList.get(i).getUuid().equals(WSDemoUUID)){
            mDevice.setDeviceType("WSDDemoUUID");
        }
    }
    connect(mDevice.getDevice());
}
```

Figure 11. ‘initConnection’ in Android app

In the ‘initConnection’ method, before starting the connection, we check that the UUID from the RapidIoT service is among the ones available in the specified device.

When the connection to the device is successfully established, the method ‘onDeviceReady’ is called. It is important to remark that the first time a phone is connected to a RapidIoT kit, it might ask for a PIN to verify the connection. The process is called bonding. In this case, the ‘onBondingRequired’ method is called. You can use this method to display a message to the user saying that he needs to introduce the PIN to finish the connection. Once the user correctly introduces the PIN, the ‘onBonded’ method is called.

Below we can see the ‘onBondingRequired’, ‘onBonded’ and ‘onDeviceReady’ methods from the application:

```
@Override
public void onBondingRequired(BluetoothDevice device) {
    super.onBondingRequired(device);

    bondingRequired = true;
    Toast.makeText(getApplicationContext(), "Please follow pairing instructions in
                        Notification Bar", Toast.LENGTH_LONG).show();
}
```

Figure 12. ‘onBondingRequired’ method in Android app

```
@Override
public void onBonded(BluetoothDevice device) {
    super.onBonded(device);

    //If the connecting dialog is shown we close it
    if(connectingDialog.isShowing()) {
        connectingDialog.dismiss();
    }
    getBLEManager().generateSessionID();    // We generate a sessionID

    if(bondingRequired) {

       if(MeasurementManager.getInstance().isCloudPostingEnabled(ctx))
           MeasurementManager.getInstance().startPostDataService(ctx);
           goToDataDisplayActivity();    // we move to the DataDisplay screen
    }
}
```

Figure 13. ‘onBonded’ method in Android app

```
@Override
public void onDeviceReady(BluetoothDevice device) {
    super.onDeviceReady(device);

    getBLEManager().generateSessionID();    // We generate a sessionID
    if(connectingDialog.isShowing()) {
        connectingDialog.dismiss();
    }
    if(!bondingRequired) {
        if(MeasurementManager.getInstance().isCloudPostingEnabled(ctx))
            MeasurementManager.getInstance().startPostDataService(ctx);
            goToDataDisplayActivity();
    }
    else{
        bondingRequired = false;
    }
}
```

Figure 14. ‘onDeviceReady’ method in Android app

If the bonding procedure is required, we set the Boolean ‘bondingRequired’ to true. In that case, we will redirect the user to the next screen (data display screen) when the device is bonded. In case the RapidIoT device is already known by the app and no bonding is needed, we will redirect when the ‘onDeviceReady’ method is called.

After the connection is successfully achieved, the ‘BleManager’ executes the ‘initGatt’ method from the ‘BleManagerGattCallback’. This method should be overwritten with all requests needed to initialize the profile. In our case, we will use it to request the ‘subscription’ to all notifications from the sensors characteristics:

```
private final BleManagerGattCallback mGattCallback = new BleManagerGattCallback()
{

  @Override
  protected Deque<Request> initGatt(final BluetoothGatt gatt) {
    final LinkedList<Request> requests = new LinkedList<>();
     // We enqueue the notification request for every sensor characteristic
    requests.push(Request.newEnableNotificationsRequest(
                                                mPressureCharacteristic));
    requests.push(Request.newEnableNotificationsRequest(
                                                mHumidityCharacteristic));
    requests.push(Request.newEnableNotificationsRequest(
                                                mTemperatureCharacteristic));
    requests.push(Request.newEnableNotificationsRequest(
                                                mLightCharacteristic));
    return requests;
  }
[..]
}
```

Figure 15. ‘initGatt’ method in Android app

Processing data from the sensors

Once we have requested notifications from a sensor characteristic, we will receive a message every time the RapidIoT delivers a new measurement from that sensor. This message is received through the ‘onCharacteristicNotified’ method from the ‘BleManagerGattCallback’. As an example, we will look at the case of a measurement coming from the Pressure sensor:

```
@Override
public void onCharacteristicNotified(final BluetoothGatt gatt,
                        final BluetoothGattCharacteristic characteristic){

  if(characteristic.getUuid().equals(PressureUUID)){
      byte[] rawpressure =  characteristic.getValue();
      MeasurementModel measurement = new MeasurementModel(
                           MeasurementUtils.getPressureMeasurement(rawpressure));

      if(fitsRefreshRate(MeasurementManager.TYPE_PRESSURE)){
            getMeasurementManager().addPressure(measurement);
      }
    }
    [..]
}
```

Figure 16. Read characteristic value and process it in Android app

As can be seen from the snippet above, we check the characteristic of the notification received to identify the sensor it comes from. Once we know the type of measurement we need to convert the data to float and create with it a ‘MeasurementModel’ type object:

```
      byte[] rawpressure =  characteristic.getValue();
      MeasurementModel measurement = new MeasurementModel(
                           MeasurementUtils.getPressureMeasurement(rawpressure));
```

Figure 17. Process raw measure in Android app

To perform the conversion we rely on the 'MeasurementUtil' static class which basically operates with the payload bytes to extract the value depending on the notation followed for every sensor.

In case the timings fit the refresh rate selected by the user (i.e., the time difference with the last measurement processed is above the refresh rate selected) the measurement is stored using the 'addPressure' method from 'MeasurementManager':

```
      if(fitsRefreshRate(MeasurementManager.TYPE_PRESSURE)){
            getMeasurementManager().addPressure(measurement);
      }
```

Figure 18. Add Pressure measurement in Android app

The 'MeasurementManager' is a singleton class used to store and handle the data extracted from the RapidIoT kit. On it we define four ArrayLists to store the latest measurements of every type:

```
// We declare the four ArrayList that store the latest measurements for every type
private ArrayList<MeasurementModel> arrayTemperature = new ArrayList<>();
private ArrayList<MeasurementModel> arrayHumidity = new ArrayList<>();
private ArrayList<MeasurementModel> arrayPressure = new ArrayList<>();
private ArrayList<MeasurementModel> arrayLight = new ArrayList<>();
```

Figure 19. Measurements arrays in Android app

The most important methods that the 'MeasurementManager' class implements are the following:

```
public void addTemperature(MeasurementModel measurement){..}

public void addHumidity(MeasurementModel measurement){..}

public void addPressure(MeasurementModel measurement){..}

public void addLight(MeasurementModel measurement){..}

public void startPostDataService(Context mContext){..}

public void stopPostDataService(Context mContext){..}

public ArrayList<MeasurementModel> getSelectedMeasurementArray(){..}
```

Figure 20. 'MeasurementManager' class in Android app

The methods 'addTemperature', 'addHumidity', 'addPressure' and 'addLight' add a measurement to the corresponding ArrayList. The 'getSelectedMeasurementArray' returns the ArrayList with the last four measurements so they can be printed in the table displayed to the user. The 'startPostDataService' and 'stopPostDataService' start and stop the background service that is in charge of posting the data to the server.

2.3.2.2. Implementation in iOS

Similar to the Android application, there is a singleton object for the BLE communication. This object is called 'sharedBLEInstance' and is created as follows in the 'BLEManager' class:

```
static let sharedBLEInstance = BLEManager()
```

Figure 21. Getting the singleton instance for the BLEManager in iOS

The ‘BLEManager’ class implements several delegates which are used in the different phases of the Bluetooth communication. You can find a list of these delegate methods in Figure 22.

```
protocol BLEDelegate {
   func bleDidUpdateState(state: CBManagerState)
   func bleDidConnectToPeripheral(peripheral: CBPeripheral?)
   func bleDidFailConnectToPeripheral()
   func bleDidDisconnectFromPeripheral()
   func bleDidReceiveData(char: String, data: Data?)
   func bleDidDiscoverDevice(peripheral: CBPeripheral?, advertisementLocalName: String?)
}
```

Figure 22. BLEManager delegate methods

The Rapid IoT device in the iOS application is another Singleton object, so it can be accessed anywhere through the application. This object is updated each time a new Rapid IoT device is connected/disconnected. It contains 4 arrays of four elements, one for each measurement. The ‘Device’ class definition can be found in Figure 23.

```
class Device : NSObject{
   private var name: String?
   private var bleMACAddress: String?
   private var temperatureData : Array<Measurement> = Array<Measurement>(repeating:
Measurement(value: nil,timestamp: nil), count: 4)
   private var pressureData : Array<Measurement> = Array<Measurement>(repeating:
Measurement(value: nil,timestamp: nil), count: 4)
   private var humidityData : Array<Measurement> = Array<Measurement>(repeating:
Measurement(value: nil,timestamp: nil), count: 4)
   private var lightData : Array<Measurement> = Array<Measurement>(repeating: Measurement(value:
nil,timestamp: nil), count: 4)

   //The device object follows the singleton pattern, we will access this instance throughout the whole
application.
   static let deviceInstance = Device()

   […]
```

Figure 23. ‘Device’ class in iOS app

Scanning for devices

Right after initializing the application, the BLEManager starts scanning for Rapid IoT devices. This is done in the ‘WelcomeViewController’ class inside the ‘viewWillAppear’ method as shown in Figure 24.

```
if(BLEManager.sharedBLEInstance.getBLEState() == CBManagerState.poweredOn){
        if(!BLEManager.sharedBLEInstance.isBLEScanning())
        {
           BLEManager.sharedBLEInstance.startScan()
        }
    }
```

Figure 24. Start scanning for new Bluetooth devices

Before starting the scanning, we check that the BLE adapter state is powered on and there is no active scanning already ongoing. In order to scan only for Rapid IoT devices, we need to perform a configuration in the ‘startScan’ method. This is shown in Figure 25. We include only the service UUID we are interested in, in this case the Rapid IoT service UUID.

```
func startScan() {
      if !isBLEAvailable() {
         print("[ERROR] Couldn´t start scanning, BLE not available")
         return
      }

      isScanning = true

      let services:[CBUUID] = [CBUUID(string: Constants.WEATHER_SERVICE_UUID)]
      self.centralManager!.scanForPeripherals(withServices: services, options:
[CBCentralManagerScanOptionAllowDuplicatesKey: true])
   }
```

Figure 25. ‘startScan’ method in iOS app

When a new device is scanned, the ‘bleDidDiscoverDevice’ callback in the ‘WelcomeViewController’ class is called. This can be seen in Figure 26. Every time a new device is discovered, it is necessary to check that it is not null and that the device is not already contained in the ‘bleDevicesList’.

The Rapid IoT devices stored in the ‘bleDevicesList’ have a field called ‘lastUpdate’. This field is updated each time a device is discovered via BLE and every time the ‘devicesTable’ is reloaded we check if the device has not been found in the last 2 seconds. In case the device has not been found in the last 2 seconds, we remove it from the ‘devicesTable,’ as we consider it is not available to connect.

```
func bleDidDiscoverDevice(peripheral: CBPeripheral?, advertisementLocalName: String?){
    if let mPeripheral = peripheral?.name{
      print("Discovered device: " + mPeripheral)
    }
    else {
      print("Couldnt get name of discovered device")
    }

    if(peripheral?.name != nil && advertisementLocalName != nil &&
!(bleDevicesList.map{$0.0}).contains(peripheral!)){
      bleDevicesList.append((peripheral!, advertisementLocalName!, Date()))
    }
    else if let i = bleDevicesList.index(where: ({$0.peripheral === peripheral})){
      bleDevicesList[i].lastUpdate = Date()
    }

    //Update devices table to show the user the new discovered device (if any)
    devicesTable.reloadData()
  }
```

Figure 26. Discover new Bluetooth devices in iOS app

Connecting to a device

When the user taps on any device in the ‘devicesTable’ list the connection procedure starts. This is shown in Figure 27.

As can be observed in the comments inside the code snippet, there are three options when the user taps on any device in the table:

1- The device tapped in the table is a new device and we are connected to another device. In this case we display a pop up warning the user if he wants to disconnect from the old device and connect to the new one.

2- The device tapped is the same we are currently connected to. In this case we just move to the Data display screen and nothing in the Bluetooth connection changes.

3- There is no device connected in the application and the user taps in a device. In this case we just start the connection with the new device.

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    self.devicesTable.deselectRow(at: indexPath, animated: true)

    //When the user taps a cell there are three different options:
    // 1 - The device tapped is not the same device it is connected to the app, in this case show a dialog
to warn the user if he wants to switch devices
    // 2 - The device tapped is the same device connected to the app, take the user to the Data display
view
    // 3 - There are no devices connected to the app, in this case stops the bluetooth scanning and
begins the connection procedure with the selected device
    if(BLEManager.sharedBLEInstance.isConnected() && (bleDevicesList[indexPath.row].advName !=
BLEManager.sharedBLEInstance.getCurrentPeripheralName())){
      showDisconnectDialog(indexPath: indexPath)
    }
    else if (BLEManager.sharedBLEInstance.isConnected() && (bleDevicesList[indexPath.row].advName
== BLEManager.sharedBLEInstance.getCurrentPeripheralName())){
      self.performSegue(withIdentifier: "ShowView", sender: self)
    }
    else{
      BLEManager.sharedBLEInstance.stopScan()
      BLEManager.sharedBLEInstance.connectToPeripheral(peripheral:
bleDevicesList[indexPath.row].peripheral)
      selectedIndexPath = indexPath
    }
  }
```

Figure 27. Connect to a device in iOS app

As previously explained, the ‘BLEManager’ class is in charge of managing the Bluetooth communications. When a new device is connected, the ‘didConnect’ callback is called. Right after we get a valid connection we stop scanning for new devices and proceed to discover the services and characteristics of the new connected device.

The characteristics we are looking for appear in section 2.3.1, we need to subscribe to those characteristics in order to get notifications each time they are written with a new value. This can be found in Figure 28.

```
//CBPeripheral delegate
  //When the services have been discovered we call the chracteristics of that service.
  func peripheral(_ peripheral: CBPeripheral, didDiscoverServices error: Error?) {
    if error != nil {
      print("[ERROR] Error discovering services. \(error!.localizedDescription)")
      return
    }

    for service in peripheral.services! {
      peripheral.discoverCharacteristics(nil, for: service)
    }
  }

  //When the device services have been read we need to read all the chracteristics from the MAC
address and sensors data
  func peripheral(_ peripheral: CBPeripheral, didDiscoverCharacteristicsFor service: CBService, error:
Error?) {
    if error != nil {
      print("[ERROR] Error discovering characteristics. \(error!.localizedDescription)")
      return
    }

    for characteristic in service.characteristics! {
      self.characteristics[characteristic.uuid.uuidString] = characteristic
      if(characteristic.uuid.uuidString == Constants.MAC_ADDRESS_CHAR_READ_UUID){
        self.readMACAddressCharacteristic()
        self.enableNotificationsTemp(enable: true)
        self.enableNotificationsHumidity(enable: true)
        self.enableNotificationsPressure(enable: true)
        self.enableNotificationsLight(enable: true)
      }
    }
  }
```

Figure 28. Discover services and characteristics in iOS app

Processing data from the sensors

Once we are subscribed to the Bluetooth notifications to receive the measurements in their corresponding characteristics, let's see how to process and store them in the application.

Every time the 'BLEManager' receives a new value for any subscribed characteristic, the 'didUpdateValueFor' callback method is called. We check what measurement the characteristic that has been written corresponds to and call a delegate method to notify the controller. This is shown in Figure 29.

```
//Each time a new value is received from any of the characteristics this function gets called and we
notify the delegate with the new data read.
   func peripheral(_ peripheral: CBPeripheral, didUpdateValueFor characteristic: CBCharacteristic, error:
Error?) {
      if error != nil {
         print("[ERROR] Error updating value. \(error!.localizedDescription)")
         return
      }

      /* Data change */
      if characteristic.uuid.uuidString == Constants.TEMP_CHAR_READ_UUID {
         self.delegate?.bleDidReceiveData(char: Constants.TEMP_CHAR_READ_UUID, data:
characteristic.value! as Data)
      }
      else if characteristic.uuid.uuidString == Constants.HUMIDITY_CHAR_READ_UUID {
         self.delegate?.bleDidReceiveData(char: Constants.HUMIDITY_CHAR_READ_UUID, data:
characteristic.value! as Data)
      }
      else if characteristic.uuid.uuidString == Constants.PRESSURE_CHAR_READ_UUID {
         self.delegate?.bleDidReceiveData(char: Constants.PRESSURE_CHAR_READ_UUID, data:
characteristic.value! as Data)
      }
      else if characteristic.uuid.uuidString == Constants.LIGHT_CHAR_READ_UUID {
         self.delegate?.bleDidReceiveData(char: Constants.LIGHT_CHAR_READ_UUID, data:
characteristic.value! as Data)
      }
      else if characteristic.uuid.uuidString == Constants.MAC_ADDRESS_CHAR_READ_UUID {
         self.delegate?.bleDidReceiveData(char: Constants.MAC_ADDRESS_CHAR_READ_UUID, data:
characteristic.value! as Data)
      }
   }
```

Figure 29. ‘bleDidUpdateValueFor’ callback in iOS app

Now let’s look at the ‘DataDisplayViewController’ and go to the implementation of the ‘bleDidReceiveData’ delegate. We will only check the temperature measurement. The rest of the measurements have a similar process.

At first we need to check that the characteristic is the one we want to store (temperature, humidity…). Then there is a variable for each characteristic called ‘isReadyToTakeTemperature’, ‘isReadyToTakeHumidity’… These variables are set to true using a timer refreshed with the configured Refresh Rate in the Settings menu. When it is time to store a measurement, this variable is set to true and the measurement is processed from the HEX string received in the BLE characteristic and stored as float in the ‘Device’ object together with the timestamp. See Figure 30.

```
func bleDidReceiveData(char: String, data: Data?) {
      if(char == Constants.TEMP_CHAR_READ_UUID){
         if(isReadyToTakeTemperature)
         {
            isReadyToTakeTemperature = false

            //Parse the value from the array and return temperature in Float value
            let hexEncoded = data!.hexEncodedString()
            let element1 = hexEncoded.substring(with: 4..<6)
            let element2 = hexEncoded.substring(with: 2..<4)
            let sum = element1 + element2

            let result = Float32(strtoul(sum,nil,16))/100

            //Get current timestamp and form measure object
            let now = UInt64(NSDate().timeIntervalSince1970)

            //Store the measure in the device object
            let newMeasurement = Measurement(value: result, timestamp: now)
            Device.deviceInstance.addTemperatureValue(measure: newMeasurement)

            TableUpdater.tableUpdaterInstance.updateTable(sensorType:
Constants.SensorType.temperature)
         }
      }
[….]
```

Figure 30. Processing the measurement data received from Bluetooth in iOS app

2.3.3. Posting data to the cloud

2.3.3.1. Implementation in Android

Another functionality of the Weather Station app allows the user to post the data gathered from the RapidIoT kit to the cloud application. The communication between the Weather Station app and the Cloud is implemented using an MQTT broker from Amazon Web Services. Here we describe how the communication with the broker is set up for the Android application. For the backend communication between the Cloud application and the broker please refer to section 4 of the present document.

The Android application uses the ‘Paho’ library from the Eclipse group to establish the communication with the broker and publish the messages. This library is included in the *build.gradle* file from the app:

```
dependencies {
[..]
implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.1.1'
implementation 'org.eclipse.paho:org.eclipse.paho.android.service:1.1.1'
[..]
}
```

MQManager class

In order to handle the communication with the broker, the application makes use of the ‘MQManager’ class. This class holds specific information needed to connect to the AmazonMQ broker, like the server URI, the username and password and the topics used to publish data or clear the database:

```
// Weather Station Demo specific credentials and parameters
private String clientId = "WSDClient";
private final String serverURI = "ssl://52.34.158.120:8883" ;
private final String publishDataTopic = "weather-station-android";
private final String clearDataTopic = "weather-station-clear-database";
private static final String userName = "****";
private static final String password = "****";
```

Figure 31. Broker parameters in Android app

The most relevant methods implemented in the ‘MQManager’ are the following:

```
public void connect(){..}

public boolean isConnected (){..}

public boolean publishLastMeasurement() {..}

public void clearDataFromCloud() {..}
```

Figure 32. Broker communication methods in Android app

The ‘connect’ method sets up all parameters and executes the connection to the MQ broker:

```
public void connect(){
    // We configure here the connection to the MQTT broker
    MqttConnectOptions connectOptions = new MqttConnectOptions();
    connectOptions.setMqttVersion(MqttConnectOptions.MQTT_VERSION_3_1_1);
    connectOptions.setAutomaticReconnect(false);
    connectOptions.setCleanSession(false);
    connectOptions.setUserName(userName);
    connectOptions.setPassword(password.toCharArray());
    connectOptions.setWill(publishDataTopic,"/will".getBytes(), 0,false);
    java.util.Properties sslClientProperties = new Properties();
    sslClientProperties.setProperty("com.ibm.ssl.protocol", "SSL");
    connectOptions.setSSLProperties(sslClientProperties);

    try{
        // We create the MQTT client, set the callback and connect
        client = new MqttClient(serverURI, clientId, new MemoryPersistence());
        client.setCallback(mqttCallback);
        client.connect(connectOptions);

    }catch(MqttException exception){
        Log.d(APPTAG, "MQManager - Exception!");
        exception.printStackTrace();
    }
}
```

Figure 33. Connection to the broker in Android app

Some of these parameters are MQTT specific, like the Quality of Service (QoS), the protocol or the server address and credentials. Before connecting, we also define the ‘MQTTCallback’, which specifies the routines that will be called depending on the communication events or notifications (e.g., connection lost, connection complete or message arrived).

The ‘isConnected’ method returns a Boolean to indicate if there is an active connection with the broker.

The method ‘publishLastMeasurement’ is used to publish the latest set of measurements to the cloud.

```
public boolean publishLastMeasurement() {
    MqttMessage msg = new MqttMessage();

    // We create an instance of the MeasurementManager and the WSDBleManager
    MeasurementManager measurementManager = MeasurementManager.getInstance();
    WSDBleManager bleManager = WSDBleManager.getInstance(mContext);

    // We check that we have a complete set of measurements to post
    if(measurementManager.isMeasurementSetComplete()) {
        // We retrieve the MAC address
        String macAddress = bleManager.getDeviceMAC().replace(":", "");

        // Create the PostDataMessage object to generate the JSON string
        PostDataMessageModel myBrokerMessage = new PostDataMessageModel(
                              bleManager.getDeviceName(),
                              macAddress,
                              bleManager.getSessionID(),
                              measurementManager.getLatestSetOfMeasurements());

        msg.setPayload(myBrokerMessage.generateJSON().getBytes());

        try {
            // We publish the message to the android queue from the broker
            client.publish(publishDataTopic, msg.getPayload(), 2, false);

        } catch (MqttException e) {
            e.printStackTrace();
            Log.d(APPTAG, "Publish Exception! " + e.getMessage());
            return false;
        }
    }
    else{
        // If we don't have a complete set we don't publish
        return false;
    }
    return true;
}
```

Figure 34. Publish data to broker in Android app

As can be seen from the snippet above, the first step is to check if we have a complete set of measurements (at least one measurement from every sensor). In that case, we generate an object of the class ‘PostDataMessageModel’ that helps us generating the JSON message that will later be sent to the broker using the ‘publish’ method from the ‘MQTTClient’ class.

The last method we are going to review from the ‘MQManager’ class is the ‘clearDataFromCloud’.

```
public void clearDataFromCloud() {
    MqttMessage msg = new MqttMessage();
    WSDBleManager bleManager = WSDBleManager.getInstance(mContext);

    // We create the cleardatamodel
    ClearDataMessageModel clearDataMessage = new ClearDataMessageModel(
                        bleManager.getSessionID());

    msg.setPayload(clearDataMessage.generateJSON().getBytes());

    try {
        client.publish(clearDataTopic, msg.getPayload(), 2, false);

    } catch (MqttException e) {
        e.printStackTrace();
        Log.d(APPTAG, "Publish Exception! " + e.getMessage());
    }
}
```

Figure 35. ‘clearDataFromCloud’ method in Android app

Similar to the ‘publishLastMeasurement’, this method uses the ‘ClearDataMessageModel’ to build the JSON message that is sent to the AmazonMQ.

PostDataService and ClearDataService

The application is designed so the tasks involved in the communication with the server are running in a background service to avoid blocking the UI. The application consists of two different services: the ‘PostDataService’ and the ‘ClearDataService’.

The ‘PostDataService’ is responsible for sending the latest set of measurement to the cloud. When the service is created, we connect to the broker and start a background thread to post the latest measurement periodically. The periodicity of this operation depends on how the user sets it up in the settings menu and the availability of new data.

```
public class PostDataService extends Service {

   @Override
   public int onStartCommand(Intent intent, int flags, int startId) {
       myMQManager = MQManager.getInstance(getApplicationContext());
       myMQManager.connect();

       // We create the thread and start it
       thread = new Thread(new ServiceThread(startId));
       thread.start();
       return START_STICKY;
   }

   final class ServiceThread implements Runnable{
     int serviceId;  // Reference of the ServiceID that created the thread

     ServiceThread(int serviceId){
         this.serviceId = serviceId;
     }

     @Override
     public void run() {
        synchronized (this){
            while(active) { // this stops the thread when the service is stopped

                //We check that the MQManager is connected
                if(myMQManager!=null && myMQManager.isConnected()){
                        myMQManager.publishLastMeasurement();
                }
                try{
                    wait(postingPeriod);
                }catch(InterruptedException e){
                    e.printStackTrace();
                }
            }
        }
     }
[..]
}
```

Figure 36. ‘PostDataService’ class in Android app

On the other hand, the ‘ClearDataService’ is used to send a message to the Cloud app, to delete all the data from the current session. It follows a similar structure to the ‘PostDataService’, but in this case the task calls the ‘clearDataFromCloud’ message from the ‘MQManager’.

```
public class ClearDataService extends Service {

    private MQManager myMQManager;

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        myMQManager = MQManager.getInstance(getApplicationContext());
        myMQManager.connect();

        // We create the thread and start it
        Thread thread = new Thread(new ServiceThread(startId));
        thread.start();

        return START_STICKY;
    }

    final class ServiceThread implements Runnable{
        int serviceId;  // Reference of the ServiceID that created this thread

        ServiceThread(int serviceId){
            this.serviceId = serviceId;
        }

        @Override
        public void run() {
            synchronized (this){
                //We check that the MQManager is connected to send the message
                if(myMQManager!=null && myMQManager.isConnected()){
                    myMQManager.clearDataFromCloud();
                }
            }
            stopSelf();
        }
    }
[..]
}
```

Figure 37. ‘ClearDataService’ class in Android app

2.3.3.2. Implementation in iOS

Network class

The process of uploading data to the cloud is managed in the iOS application through the ‘Network’ class. When the ‘DataDisplayViewController’ class is loaded in the screen, a new ‘Network’ object is created and initialized. The process of setting up this object is shown in Figure 38.

In the ‘Network’ object, the CocoaMQTT library as MQTT client.

```
//Create a client object in the Amazon MQ instance. This client will publish the data to a particular
queue that will be redirected to the cloud application in order to store this data in the database
  func setupBroker(){
    let clientID = "CocoaMQTT-" + String(ProcessInfo().processIdentifier)
    mqtt = CocoaMQTT(clientID: clientID, host: messageBrokerURL, port: 8883)
    mqtt!.dispatchQueue = DispatchQueue.global(qos: .userInitiated)
    mqtt!.username = "xxxxxx"
    mqtt!.password = "xxxxxx"
    mqtt!.willMessage = CocoaMQTTWill(topic: "/will", message: "dieout")
    mqtt!.keepAlive = 60
    mqtt!.cleanSession = true
    mqtt!.delegate = self
    mqtt!.enableSSL = true
  }
```

Figure 38. Configuration of the broker in the iOS app

Publish messages into the broker app

Now let's see how the measurements JSON is formed. Once we have initialized the 'Network' object in the 'DataDisplayViewController', there is a timer to publish the data in the broker application according to the Data Refresh rate configured in the Settings menu. Before sending the data to the broker application, we need to check that the data posting configuration is enabled and the data is valid. See Figure 39.

```
//Send measurements to the cloud application according to the data refresh rate defined in the settings
view
    Timer.scheduledTimer(withTimeInterval: TimeInterval(Persistent.readDataRefreshRate() ?? 1),
repeats: true){ timer in
      DispatchQueue.main.async {
        if(BLEManager.sharedBLEInstance.isConnected()){
          if Persistent.readCloudPostSettings() ?? false{
            if(DataValidation.isValidData(device: Device.deviceInstance)){
              let getJSON = JSONGeneration.encodeMeasurementJSON(deviceObject:
Device.deviceInstance)
              self.networkSample?.sendDataToServer(data: getJSON, isDatabaseClean: false)
            }
          }
          else{
            print("Cloud post data is not active")
          }
        }
      }
    }
```

Figure 39. Form measurement JSON

The Clear database JSON is formed when the user clicks on ‘Clear database’ button in the ‘DataDisplayViewController’, the formation of the JSON is shown in Figure 40.

```
  func clearDatabaseTapped(clearDatabase: Bool) {
    if(clearDatabase){
      let clearDatabaseString = JSONGeneration.clearCloudDataJSON(sessionID:
Persistent.readSessionID()!)
      self.networkSample?.sendDataToServer(data: clearDatabaseString!, isDatabaseClean: true)
    }
  }
```

Figure 40. Form Clear database JSON

The JSONs are generated using the ‘JSONEncoder’ method provided in the ‘Foundation’ library. The generation of the measurement and clear database JSONs is shown in Figure 41.

```
  //Create a JSON object with the measurements, in order to form the JSON object we use the
DeviceJSON structure defined above. Use JSONEncoder functionality included in Swift.
  static func encodeMeasurementJSON(deviceObject: Device) -> String{
    let createStructure = DeviceJSON(name: deviceObject.getName(), bleMACAddress:
deviceObject.getBleAddress(), sessionID: Persistent.readSessionID()!,
                     temperature: deviceObject.getTemperature(position: 0).value!, humidity:
deviceObject.getHumidity(position: 0).value!,
                     pressure: deviceObject.getPressure(position: 0).value!, light:
deviceObject.getLight(position: 0).value!,
                     timestamp: deviceObject.getTemperature(position: 0).timestamp!)

    let jsonEncoder = JSONEncoder()
    let jsonData = try! jsonEncoder.encode(createStructure)
    return String(data: jsonData, encoding: String.Encoding.utf8) ?? "Unknown JSON"
  }

  //This function creates the JSON object sent to the cloud for the clear database functionality. In this
case we don't use a predefined structure since the data to be included in the JSON object is quite simple.
  static func clearCloudDataJSON(sessionID: String) -> String?{
    let createStructure = ClearCloud(SessionID: sessionID)

    let jsonEncoder = JSONEncoder()
    let jsonData = try! jsonEncoder.encode(createStructure)

    print(String(data: jsonData, encoding: String.Encoding.utf8))
    return String(data: jsonData, encoding: String.Encoding.utf8) ?? "Unknown JSON"

  }
```

Figure 41. JSONGeneration class

Finally, once the JSON is formed, the 'sendDataToServer' method is called. In this method we publish the data either to the 'weather-station-ios' or 'weather-station-clear-database' topic. See Figure 42.

```
//Send a JSON data to the server (in case that the cloud posting option is enabled). All the data will be sent to the server with QOS 2 (Send exactly once)
  func sendDataToServer(data: String, isDatabaseClean: Bool){
    if(isReadyToPublish){
      if(!isDatabaseClean){ //Check if the message sent belongs to a clean database command, in this case redirect to another queue in the broker.
        mqtt?.publish("weather-station-iOS", withString: data, qos: .qos2)
      }
      else{
        mqtt?.publish("weather-station-clear-database", withString: data, qos: .qos2)
      }
    }
  }
```

Figure 42. 'sendDataToServer' method in iOS app

3. Cloud application

Development setup

The selected IDE for the development of the cloud application has been Visual studio community 2017 version 15.9.5. The web application has been programmed using .NET framework, following the ASP.NET MVC programming model. This popular programming pattern provides a clear way to distinguish between business, presentation and data model layers.

Visual studio community 2017 can be downloaded in the following link: https://www.asp.net/

Together with the cloud application, a broker application has been developed to receive the measurements from the mobile app and store them in the database. A more in-depth explanation regarding the functionality of both applications is provided in the following sections of this document.

The application source code of both applications can be found in the following links:

- Cloud app: https://bitbucket.org/mobileknowledge/weather_station_demo-cloudapp/src/master/
- Broker app: https://bitbucket.org/mobileknowledge/weather_station_demo-brokerapp/src/master/

The database engine used to store the measurements retrieved from the sensors is SQL Server. It is hosted in Amazon Web Services. In the development phase, the application used to connect to this SQL database was Microsoft SQL Server Management Studio version 17.9.1.

For first time users of ASP.NET MVC technology, it is recommended to check the tutorials from the Microsoft official website to start getting familiar with some concepts that will be explained later on in this document: https://docs.microsoft.com/es-es/aspnet/mvc/overview/getting-started/introduction/getting-started

Application architecture

The cloud application is built following the 'Model-View-Controller' design pattern. A simple diagram of the relationship between layers can be found in Figure 43.

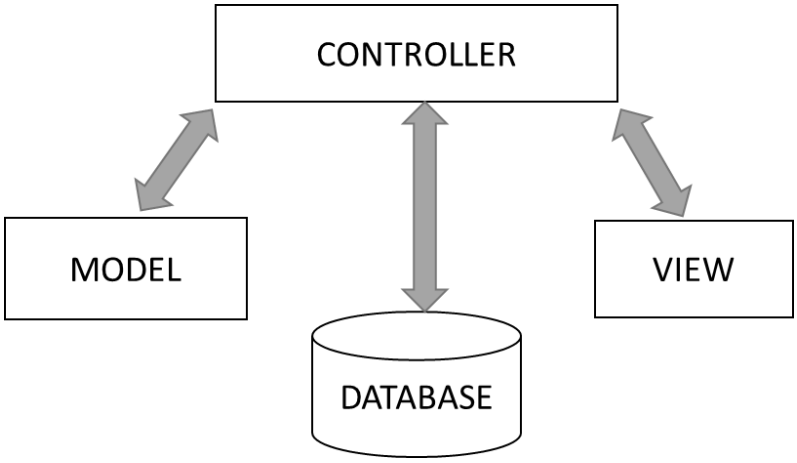

Figure 43. Cloud application architecture

The application structure consists of 3 main blocks with the purpose of isolating and clearly identifying the functions and competences of each entity. The layers involved in the design are:

- **View**: The view block is the interface between the controller and the end user. It is in charge of interacting with the user in order to present information or to process input actions from the user and derive them to the controller layer.
- **Controller**: This package contains the files related to the application logic: process login, request information from a device with a specific MAC address, JSON interpretation, Data validation, etc.
- **Model**: The Model block contains the data model definition, for example the Rapid IoT device class, etc.

Regarding the project organization in Visual Studio, there is one solution called "WeatherStationDemo_CloudApp". This solution is composed of two projects:

- WeatherStationDemo_CloudApp: It has the same name as the solution and is the main project. All the application logic is here (controllers, views).
- WeatherStationDemo_Contracts: This project contains definitions regarding the data Model that will be used through the CloudApp project.

3.2.1. View layer

The View layer consists of two screens: the Login view and the Data display view.

3.2.1.1. Login view

The Cloud application is defined to have a login view (Figure 44), where the user has to introduce the corresponding login parameters (MAC Address of the Rapid IoT device and session ID). The Bluetooth

MAC address of the Rapid IoT device shall be the last 4 digits of the full address (which can be found at the back of the device or in the mobile apps).

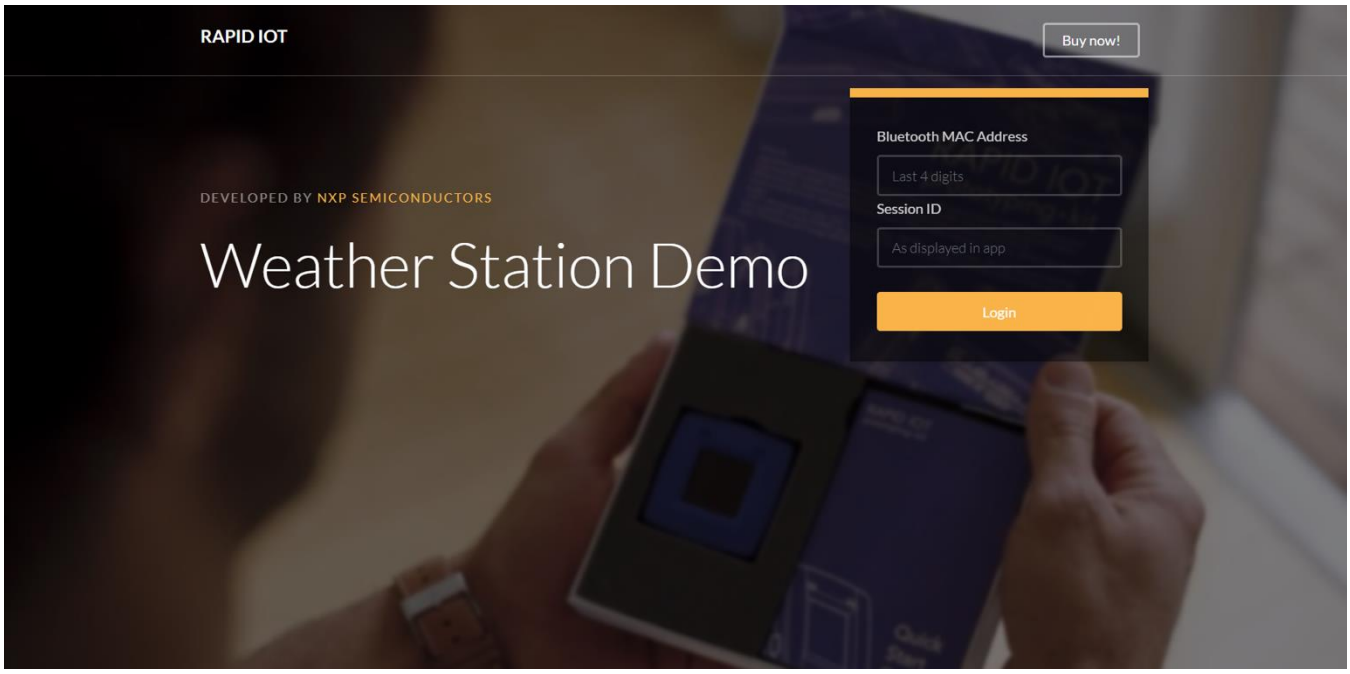

Figure 44. Web UI: Login view

The HTML code for the login form view is shown in Figure 45. This form is submitted by clicking on the 'Login' button, the 'Index' POST method in the Login controller is called when submitting the form. The Login controller methods are explained in section 3.2.2.1.
Inside the HTML, you can observe some portions of code that make use of Razor Markup. This syntax is based on ASP.NET. For more information about it, please check the ASP.NET tutorials provided at the beginning of the section.

```
@using (Html.BeginForm("Index", "Login", FormMethod.Post, new { @class = "form", role =
"form" }))
        {
        @Html.AntiForgeryToken()
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })

        [……]
          <form action="#">
            <div class="row form-group">
               <div class="col-md-12">
                  @Html.LabelFor(m => m.bleMACAddress)
                  @Html.TextBoxFor(m => m.bleMACAddress, new { @class = "form-control",
placeholder = "Last 4 digits" })
                  @Html.ValidationMessageFor(m => m.bleMACAddress, "", new { @class =
"text-danger" })
                </div>
             </div>
           <div class="row form-group">
              <div class="col-md-12">
                  @Html.LabelFor(m => m.SessionID)
                  @Html.TextBoxFor(m => m.SessionID, new { @class = "form-control",
placeholder = "As displayed in app" })
                  @Html.ValidationMessageFor(m => m.SessionID, "", new { @class = "text-
danger" })
               </div>
             </div>
          <br />
          <div class="row form-group">
              <div class="col-md-12">
                 <input type="submit" class="btn btn-primary btn-block" value="Login">
               </div>
          </div>
         [……]
        }
```

Figure 45. Login view HTML code

3.2.1.2. Data display view

Once these login parameters are validated, the Data display view is displayed in the web browser, in this view the data retrieved from the sensors are displayed graphically.

The Data display view (Figures 46-47) has a header containing the session information (MAC address of the device and session ID), together with a Logout button which will close the current session and bring the user again to the Login view.

The ‘Export’ button will trigger the .csv or .txt file download containing all the sensor measurements data in the database associated with the current user logged into the application. See Figure 48.

At the bottom of the screen, it is possible to configure the time scale in seconds or minutes and set the temperature scale in Celsius or Fahrenheit degrees.

Figure 46. Web UI: Data display view (1)

Figure 47. Web UI: Data display view (2)

The graphs are plotted using the google.charts library. For more information about this library please refer to its website:

https://developers.google.com/chart/

In the ‘Main.cshtml’ file there are two JavaScript functions called ‘initChartsOptions’ and ‘initCharts’. The initialization of the graphs can be found in these functions. More information about how to plot graphically can be found in Section 3.4.

In Figure 48 you can see the HTML code to export a measurement either in .csv or .txt format. When submitting the form, the 'ExportData' method in the 'DataDisplayController' is called and will be in charge of generating the file and triggering the download.

As previously mentioned, in case of any doubt with Razor Markup language, please go to the ASP.NET tutorials.

```
@using (Html.BeginForm("ExportData", "DataDisplay", FormMethod.Post, new { role = "form"
}))
  {
    <div class="wsdchartRowE row">
       <label class="wsdMarginRight">
           <font color="white">File format: </font>
       </label>

       @Html.DropDownListFor(m => m.formats,(IEnumerable<SelectListItem>)ViewBag.Formats,
       new { @class = "wsd-form-control" })
    </div>
    <div class="wsdchartRowG row">
       <fieldset>
          <div class="row-mt-5em">
             <input type="submit" class="btn-export-refresh" value="Export">
          </div>
       </fieldset>
    </div>
  }
```

Figure 48. Data display HTML Export file

3.2.2. Controller layer

This section defines the different modules that implement the functional behavior of the controller layer. The controller layer has two main elements: Login and Data Display controller.

3.2.2.1. Login Controller

In Figure 49, the source code of the Login controller is shown. It consists of two 'Index' methods. The first method is the HTTP GET. This method is called when there is an HTTP GET request to the Login controller, since the Login controller is the first controller called in the application when any user starts the cloud application by opening a new browser tab and entering the endpoint URL. This method is the entry point of the application. It returns the 'View' object to be loaded in the screen; in this case, it corresponds to the 'Views/Login/Index.cshtml' file.

The second 'Index' method corresponds to the HTTP POST request; in this case, the method is called when the user submits the login information by clicking on 'Login' in the Login view (see Figure 5). The input parameter is a LoginViewModel object. This class is shown in Figure 49. This is a simple object that contains one parameter for the BLE MAC address and another for the Session ID. Right after the 'Index' method is called in an HTTP POST request, these parameters are checked:

- In case these model parameters are valid, the next step is to check in the database if this login information exists (i.e., if there has been any session created with those values). If there is an existing session with those parameters, this method will redirect the user to the Data Display View, which is explained in the next section.

- In case the login data is not valid, a warning message should appear to the user and the Login view will remain in the screen.

```
public class LoginController : Controller
    {
        //This is the application entry point. The user is redirected to the Index view.
        [AllowAnonymous]
        public ActionResult Index()
        {
            return View();
        }

        //When hitting the Login button this HTTPPOST method is called with the user
information for login.
        [AllowAnonymous]
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Index(LoginViewModel modelLogin)
        {
            //Check if there is no missing fields in the login model (BLE MAC address and
session ID)
            if (!ModelState.IsValid)
            {
                return View(modelLogin);
            }
            else
            {
                //Get database instance
                using (var DbContext = new WeatherStationDatabaseContext())
                {
                    //Check if there is any row that contains data for the session ID and
MAC address provided.
                    var retrieveDeviceInfo = DbContext.devices_table.Where(device =>
device.SessionID == modelLogin.SessionID
                        && device.bleMACAddress.Substring(12) ==
modelLogin.bleMACAddress).FirstOrDefault();

                    //If there is data just redirecto to the DataDisplayController with
the current session ID and MAC Address.
                    //Otherwise show an error in login
                    if (retrieveDeviceInfo != null)
                    {
                        return RedirectToAction("Main", "DataDisplay", new { sessionID =
modelLogin.SessionID,
                            MACAddress = modelLogin.bleMACAddress
                        });
                    }
                    else
                    {
                        TempData["ErrorMessage"] = "Error in login, please check that the
login data is correct.";
                        return View();
                    }
                }
            }
        }
    }
```

Figure 49. Login controller source code

3.2.2.2. Data Display Controller

The data display controller source code will be shown in separate code snippets in order to help digest the code. The first code snippet is shown in Figure 50 and shows the 'Main' method. This is the entry point of the data display controller.

This class has five static LinkedLists created to store each measurement value and the global timestamp,. These lists are filled when reading the database and shared with the Data Display View file to plot the graphs.

The 'Main' method has the SessionID and MAC Address as input parameters. Once this method receives these values, they are stored in cookies so the cloud application can access them at any moment of execution.

Finally, before displaying the View associated to the Main method of the Data Display Controller (Views/DataDisplay/Main.cshtml file), a ViewBag object is created to share information with the view. ViewBag objects are created in the controller and they are used to pass any kind of information to the view (more information about this type of objects can be found in the ASP.NET MVC tutorial).

```
//This is the main controller of the cloud application. It handles the data display in
graphs, allows the user to download
    //measurements, allows to change from Celsius to Fahrenheit degrees and logout the
current session.
    public class DataDisplayController : Controller
    {
        //Create a linked list to store the data for the sensors. Since we are storing
the data using the same timestamp we can
        //use a common list containing the timestamp instead of a separate list for each
measurement.
        static LinkedList<float> temperatureList = new LinkedList<float>();
        static LinkedList<long> timestampList = new LinkedList<long>();
        static LinkedList<float> humidityList = new LinkedList<float>();
        static LinkedList<float> pressureList = new LinkedList<float>();
        static LinkedList<float> lightList = new LinkedList<float>();

        //This is the entry point of the Controller when the user logs into the Cloud
app, the session ID and MAC Address are stored as
        //cookies.
        public ActionResult Main(String sessionID, String MACAddress)
        {
            //Store the SessionID and MAC Address in cookies so they can be accessible
from the Web client.
            Session["sessionID"] = sessionID;
            Session["MACAddress"] = MACAddress;

            //Pass the View a ViewBag variable containing the .csv and .txt formats to
display the dropdownlist.
            ViewBag.Formats = FileFormatDefinition.listFormats.ToList();
            ViewBag.SessionID = sessionID;
            ViewBag.MACAddress = MACAddress;

            return View();
        }
```

Figure 50. Data display controller source code – Main method

The ‘Logout' method is shown in Figure 51. It receives an HTTP GET request from the view after the user clicks on the ‘Logout’ button. It will delete the cookies created for the current session and redirect to the Index method of the Login controller.

```
//The logout method deletes the current user data and returns to the Index page.
        public ActionResult Logout()
        {
            //Deletes the current user information, just store null value to make sure
the previous value has dissapeard.
            Session["sessionID"] = null;
            Session["MACAddress"] = null;

            return RedirectToAction("Index", "Login");
        }
```

Figure 51. Data display controller source code – Logout

One of the most important methods of the Data Display controller is ‘UpdateCharts’. This method is called when refreshing the graphs in the view. As input parameters, it reads the value of the timescale and temperature scale in the HTML view file and processes the data accordingly. See Figure 52.

At first we need to get an instance of the database; in ASP.NET MVC this is done with the following piece of code:

```
      using (var DbContext = new WeatherStationDatabaseContext())
```

After doing this, in DbContext we have a reference to the database and we can operate with it using the pre-defined methods in ASP.NET MVC framework.

Once we have the reference to obtain objects from the database, we count the number of elements in the database. In case there is at least one measurement (count > 0), we proceed to read the last record and store it in the temporary LinkedLists.

In order to know the next element to store in the LinkedLists, we read the timestamp of the last stored measurement. Check if the timescale is minutes or seconds and start iterating over the database elements until the timespan is consistent with the selected timescale.

This procedure is repeated until the temporary LinkedLists are filled with 30 elements (if the database contains less than 30 elements, the loop will stop beforehand).

```
public JsonResult UpdateCharts(string timescale, string tempscale)
        {
            //Get an instance of the database.
            using (var DbContext = new WeatherStationDatabaseContext())
            {
                //Read the sessionID and look in the database how many measurements there
are with that particular Session ID.
                var mSessionID = Session["sessionID"];
                var databaseCount = DbContext.devices_table.Count(x => x.SessionID ==
mSessionID.ToString());

                //The default value for the temperature scale is Celsius, check if this
setting has been changed.
                var myTempScale = tempscale;
                bool changeToFahrenheit = tempscale.Equals("Fahrenheit");

                //Check if there are elements for the current session ID
                if (databaseCount > 0)
                {
                     […]
                    //Read last timestamp stored in the database.
                    var dbRow = DbContext.devices_table.Where(x => x.SessionID ==
mSessionID.ToString()).ToList().LastOrDefault();

                    //We need to keep track of the last timestamp and ID stored in the
table as a reference to fill the Linked lists
                    //with the previous data.
                    var lastTimestamp = dbRow.Measurement.Timestamp;
                    var lastId = dbRow.Id;

temperatureList.AddFirst(convertTempIfNeeded(dbRow.Measurement.Temperature,
changeToFahrenheit));
                    humidityList.AddFirst(dbRow.Measurement.Humidity);
                    pressureList.AddFirst(dbRow.Measurement.Pressure);
                    lightList.AddFirst(dbRow.Measurement.Light);
                    timestampList.AddFirst(lastTimestamp);

                    […]
                    while (index < databaseCount)
                    {
                        //Reads the next database row taking into account the last ID
read from the previous row.
                        var nextDbRow = DbContext.devices_table.Where(x => x.Id < lastId
&& x.SessionID == mSessionID.ToString()).
                            OrderByDescending(x => x.Id).FirstOrDefault();
                        var nextTimestamp = nextDbRow.Measurement.Timestamp;

                        TimeSpan difference =
TimeConverter.UnixTimeStampToDateTime(nextTimestamp)
                            - TimeConverter.UnixTimeStampToDateTime(lastTimestamp);

                        switch (timescale)
                        {
                            case "Minutes":
                                if (difference.TotalMinutes <= -1) //If the time
difference is 1 minute or more we can store it
                                {
```

```
temperatureList.AddFirst(convertTempIfNeeded(nextDbRow.Measurement.Temperature,
changeToFahrenheit));
                    […]
                                    rowsInserted++;
                                }
                                break;
                            case "Seconds":
                                […]
                        }

                        lastId = nextDbRow.Id;

                        if (rowsInserted >= 30) break;

                        index++;
                    }
                }

                var result = new
                {
                    TemperatureValues = temperatureList,
                    TemperatureTimestampValues = timestampList,
                    HumidityValues = humidityList,
                    PressureValues = pressureList,
                    LightValues = lightList,
                    IsFahrenheit = changeToFahrenheit
                };

                return Json(result, JsonRequestBehavior.AllowGet);
            }
```

Figure 52. Data display controller source code – Update graphs

Finally, the last important method in the Data Display controller is ‘ExportData’. The code snippet for this method can be found in Figure 53. This method is called when the user clicks on the ‘Export’ button in the Data display view.

The input parameter is a ‘FileFormatModel object’ and it has two possible values: ‘csv’ or ‘txt’, depending on the option selected in the view.

Then we need to get an instance of the database. Later we get a list with all the values in the database that corresponds to the session ID provided. Finally, we loop through all the values in the database and add a new line for each entry in a ‘StringWriter’ object.

Once we have all the records of the database stored in the ‘StringWriter’ object, it is just a matter of exporting the text to an external file using ‘StreamWriter’ and UTF8 encoding format.

```
public ActionResult ExportData(FileFormatModel fileFormatModel)
{
    //Read file export format
    string format = Request.Form["exportformat"];
    var exportFormat = fileFormatModel.formats;

    //Create string builder objects that will be filled with the measurements data from
the database.
    string strDelimiter = "; ";
    StringBuilder sb = new StringBuilder();
    StringWriter sw = new StringWriter();
    using (var DbContext = new WeatherStationDatabaseContext())
    {
       var mSessionID = Session["sessionID"];
       var rowsAffected = DbContext.devices_table.Where(x => x.SessionID ==
mSessionID.ToString()).OrderByDescending(x => x.Id).ToList();

        […]
        foreach (RapidIoTDevice currentDevice in rowsAffected)
        {
            sw.Write(currentDevice.Measurement.Timestamp + strDelimiter);
            sw.Write(currentDevice.Name + strDelimiter);
            sw.Write(currentDevice.SessionID + strDelimiter);
            sw.Write(currentDevice.bleMACAddress + strDelimiter);
            sw.Write(currentDevice.Measurement.Temperature + strDelimiter);
            sw.Write(currentDevice.Measurement.Humidity + strDelimiter);
            sw.Write(currentDevice.Measurement.Pressure + strDelimiter);
            sw.Write(currentDevice.Measurement.Light);
            sw.Write("\r\n");
         }

          //Set the file extension according to the user input in the dropdownlist.
          string filename = "RapidIoT_" + mSessionID;
          if (exportFormat!=null && exportFormat.Equals(".txt"))
          {
              filename = filename + ".txt";
          }
          else
          {
               filename = filename + ".csv";
           }

           //Set the text format and add the current datetime to the title
           Response.ContentType = "text/plain";
           Response.AddHeader("content-disposition", "attachment;filename=" +
string.Format(filename, string.Format("{0:ddMMyyyy}", DateTime.Today)));
           Response.Clear();

           using (StreamWriter writer = new StreamWriter(Response.OutputStream,
Encoding.UTF8))
           {
              writer.Write(sw.ToString());
               }
               Response.End();

           }
```

Figure 53. Data display controller source code – Export data

3.2.3. Model layer

The model layer purpose is to represent the entities that are used in the controller and database. In this case there are two entities defined: RapidIoTDevice and Measurement.

The RapidIoTDevice class is shown in Figure 54. It contains the data needed to be stored in the database each time a new measurement is retrieved. The Measurement class is shown in Figure 55. It encapsulates all the individual measurement data into a single object.

```
public class RapidIoTDevice
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string bleMACAddress { get; set; }
    public string SessionID { get; set; }
    public Measurement Measurement { get; set; }
}
```

Figure 54. RapidIoTDevice class

```
public class Measurement
{
    public float Temperature { get; set; }
    public float Humidity { get; set; }
    public float Pressure { get; set; }
    public float Light { get; set; }
    public long Timestamp { get; set; }
}
```

Figure 55. Measurement class

These classes can be found in the 'WeatherStationDemo_CloudContracts' project. This project is included in the 'WeatherStationDemo_CloudApp' solution.

3.2.3.1. Database format

The database stores all information coming from the devices sensors deployed. It is formed by a single table, called 'RapidIoTDevices'. This table has the following fields:

Field	Description	Type
Id	Unique identifier of the entry	Automated Int
bleMACAddress	Last 4 bytes of the MAC address of the device that took the measurement.	String
Name	Name of the device as shown in BLE advertising	String

SessionID	Identifier of the session from that device.	String
timestamp	Parameter to identify the moment when the measurement took place.	long
mTemperature	Temperature in Celsius (default).	Float value
mHumidity	Humidity in percentage.	Float value
mPressure	Air pressure measured in hectoPascals	Float value
mLight	Light intensity measured in lux.	Float value

More information about how to connect to the database and add new measurements can be found in section 3.3.

How to add a new measurement to the database

The cloud application database model has been created following the ASP.NET MVC code first approach. In case you are new to this approach, please review the following tutorial link to get a global idea of the procedure (http://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx).

In code first migrations, the database is built according to a data model previously created. In our case we have the RapidIoTDevice and Measurement classes that were presented in section 3.2.3.

The 'WeatherStationDatabaseContext' file in 'DataContext' folder is the file in charge of generating the database and the tables belonging to that database according to a particular data model. You can find this file in Figure 56.

```
public class WeatherStationDatabaseContext : DbContext
{
   public WeatherStationDatabaseContext() : base("WeatherStationDatabaseContext")
   {
   }

   //Add a table called devices_table in the Database. The table rows have the same
objects as the RapidIoTDevice class
    public DbSet<RapidIoTDevice> devices_table { get; set; }
}
```

Figure 56. Database context class

Once we have built this file with the tables we want in our database, we need to go to the Package manager console in visual studio and generate a new database migration. Database migrations will automatically appear under the 'Migrations' folder.

The commands sequence to generate a new database migration is the following:

1. enable-migrations
2. add-migration 'addYourMigrationNameHere
3. update-Database -ConnectionString "Data Source= aanon59uu774jt.cq4m2m0a8jif.us-west-2.rds.amazonaws.com,1433;Initial Catalog= aanon59uu774jt;User Id=WeatherStationDemoUser;password=wsdadmin12345!" -ConnectionProviderName "System.Data.SqlClient"

When the ‘update-database’ command is executed, the database hosted in Amazon Web Services will be updated with the latest changes introduced in the migration.

Now let’s see how to add a new measurement in the database; for instance, we will add the Air Quality measurement in the database.

First we need to go to the ‘WeatherStationDemo_CloudContracts’ project and modify the ‘Measurement’ class by adding the new value. See Figure 57.

```
public class Measurement
{
    public float Temperature { get; set; }
    public float Humidity { get; set; }
    public float Pressure { get; set; }
    public float Light { get; set; }
    public float AirQuality { get; set; }
    public long Timestamp { get; set; }
}
```

Figure 57. Measurement class modified

Now we need to execute the commands sequence as explained before to introduce this new parameter as a column in the database table.

After executing the ‘add-migration’ step, the new generated Migration file will be as shown in Figure 58.

```
public partial class IntroduceAirQuality : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.RapidIoTDevices", "Measurement_AirQuality", c =>
c.Single(nullable: false));
    }

    public override void Down()
    {
        DropColumn("dbo.RapidIoTDevices", "Measurement_AirQuality");
    }
}
```

Figure 58. Migration file generated with Air Quality column

Finally, after executing the ‘update-database’ command, the AWS database table will be updated with the new column.

Now that the database and the data model have been modified, it is necessary to do a couple of modifications in the broker application. These modifications are straightforward. We need to open the ‘DataModels’ folder, go to the ‘Measurement’ class and add the new value as we did in Figure 14.

On the other hand, open the ‘DatabaseOperations’ class in the ‘Utils’ folder, and modify the ‘checkUserLimitAndAddMeasurementToDatabase’ method to include this new parameter. This modification is shown in Figure 59.

```
if(rapidIoTDevice.SessionID != null)
{
     SqlCommand insertCommand = new SqlCommand("INSERT INTO dbo.RapidIoTDevices (Name,
SessionID, Measurement_Temperature," +
"Measurement_Humidity,Measurement_Pressure,Measurement_Light,Measurement_AirQuality,Measu
rement_Timestamp,bleMACAddress) " + "Values
(@Name,@SessionID,@Meas_Temp,@Meas_Humid,@Meas_Pres,@Meas_Light,@Meas_AirQuality,@Meas_Ti
me,@bleMACAddress)"
     , con);
     insertCommand.Parameters.AddWithValue("@Name", rapidIoTDevice.Name);
     insertCommand.Parameters.AddWithValue("@SessionID", rapidIoTDevice.SessionID);
     insertCommand.Parameters.AddWithValue("@Meas_Temp",
rapidIoTDevice.Measurement.Temperature);
     insertCommand.Parameters.AddWithValue("@Meas_Humid",
rapidIoTDevice.Measurement.Humidity);
     insertCommand.Parameters.AddWithValue("@Meas_Pres",
rapidIoTDevice.Measurement.Pressure);
     insertCommand.Parameters.AddWithValue("@Meas_Light",
rapidIoTDevice.Measurement.Light);
     insertCommand.Parameters.AddWithValue("@Meas_AirQuality",
rapidIoTDevice.Measurement.AirQuality);
     insertCommand.Parameters.AddWithValue("@Meas_Time",
rapidIoTDevice.Measurement.Timestamp);
     insertCommand.Parameters.AddWithValue("@bleMACAddress",
rapidIoTDevice.bleMACAddress);
     insertCommand.ExecuteNonQuery();
 }
```

Figure 59. Adding Air Quality measurement to the DatabaseOperations class

Plotting measurements in graphs

In this section we will go through the process of plotting in graphs the measurements read from the database. As shown in the code snippet in Figure 52, the result of calling the ‘UpdateCharts’ method in the ‘DataDisplayController’ is a JSON string containing the last 30 measurements of a particular session.

These measurements are sent to the ‘UpdateCharts’ which can be found in the ‘Main.cshtml’ file. In Figure 60, you can see how the controller method is called. It is done using AJAX framework in order to make the UI refresh asynchronous to the rest of view. The JSON sent from the controller is received as input parameter, called ‘result’ in the success function.

Once we have the JSON with all the information from the measurements, they need to be parsed and plotted in the graphs. Let’s see an example of how it is done with the humidity measurement in Figure 61.

```
function updateCharts()
{
    var timeScale = document.getElementById("timescale");
    var tempScale = document.getElementById("temperatureformat");
    var data = {
       timescale: timeScale.value,
       tempscale: tempScale.value
    };

   $.ajax({
    type: "POST",
    url: "/DataDisplay/UpdateCharts",
    contentType: "application/json; charset=utf-8",
    updateTargetId: "UpdateCharts",
    async: true,
    dataType: "json",
    data: JSON.stringify(data),
    success: function (result) {
       […]
```

Figure 60. UpdateCharts function in JavaScript

```
var humidityValuesArray = [];
                    result.HumidityValues.forEach(function (element) {
                        humidityValuesArray.push(element);
                    });

                    var dataHumidityChartGC = new google.visualization.DataTable();
                    dataHumidityChartGC.addColumn('datetime', 'Time');
                    dataHumidityChartGC.addColumn('number', 'Humidity');

                    for (i = 0; i < timestampsArray.length; i++) {
                        dataHumidityChartGC.addRow([new Date(timestampsArray[i] * 1000),
humidityValuesArray[i]]);
                    }

                    var humidityChartGC = new
google.visualization.LineChart(document.getElementById('humidityChartGC'));
                    humidityChartGC.draw(dataHumidityChartGC, chartOptionsHumGC);
```

Figure 61. Draw humidity graph

4. Broker application

The broker application is a simple console application created to avoid overloading the Cloud application to get the values from the Active MQTT message broker and store them in the database.

In the Weather station applications, a messaging server implementing MQTT protocol has been created in an AmazonMQ (which runs an ActiveMQTT Server) instance.

On one side, the mobile applications encapsulate the messages sent to the cloud application in JSON objects. These JSON messages are sent to different topics.

On the other side, the broker application is subscribed to these topics. Every time a topic is written, the broker application parses the JSON object received and stores the information in the database.

In Figure 62 you can see the main loop of the broker application. There is one thread created for each topic (Android, iOS, clean database), also there is a variable to keep the status of the subscription. At the application startup, all the threads are initialized but in case any exception arises, they will automatically initiate again, so the server is always active.

```
static void Main(string[] args)
{
    registerThreads = true;

    //Create an infinite loop to check continuously if the broker is subscribed to the
server.
     while (true)
     {
         try
         {
            if (registerThreads)
            {
                registerThreads = false;

                listener = new Listener();

                //Create new threads to register to the topics for a better distribution
of computing workload.
                 new Thread(() =>
                 {
                     listener.registerIOSTopic();
                  }).Start();

                  new Thread(() =>
                  {
                      listener.registerAndroidTopic();
                   }).Start();

                   new Thread(() =>
                   {
                       listener.registerClearDBTopic();
                    }).Start();
            }
         }
         catch(Exception e)
         {
            Console.WriteLine("Exception raised: " + e);
            registerThreads = true;
          }
      }
  }
```

Figure 62. Broker app – Main loop

The endpoint information can be found in the source code in the ‘BrokerAccountInfo’ class. These parameters are shown in Figure 63.

```
public static class BrokerAccountInfo
{
    public const string AWS_WSD_ENDPOINT = "ssl://b-a696cbdc-47a5-45be-a9ee-da0dd8f1d3b5-
1.mq.us-west-2.amazonaws.com:61617";
    public const int AWS_WSD_PORT = 8883;
    public const string ACTIVEMQ_CLIENTID = "WeatherStation-CloudApp";
    public const string ACTIVEMQ_USERNAME = "xxxxx";
    public const string ACTIVEMQ_PASSWORD = "xxxxx";

    public static String connectionString = "user id=xxxxxxx;" +
                                            "password=xxxxxxx;" +
                                            "server=aanon59uu774jt.cq4m2m0a8jif.us-west-
2.rds.amazonaws.com;" +
                                            "Trusted_Connection=no;" +
                                            "database=aanon59uu774jt; " +
                                            "connection timeout=30";
}
```

Figure 63. Broker app – Connection parameters

Finally, it is necessary to create the connection with the server. For that we create the 'Listener' class. It is an inner class which will be used later on to subscribe to all the topics. The constructor of the 'Listener' class is shown in Figure 64.

```
public class Listener
{
      //Topics created in ActiveMQ instance
      public const string IOS_DESTINATION_TOPIC = "weather-station-iOS";
      public const string ANDROID_DESTINATION_TOPIC = "weather-station-android";
      public const string CLEARDB_DESTINATION_TOPIC = "weather-station-clear-database";

      IConnectionFactory connectionFactory;
      IConnection _connection;

      public Listener()
      {
          try
          {
              //Create a connection to the Amazon Web Services endpoint with the broker
application
              connectionFactory =
                  new ConnectionFactory(BrokerAccountInfo.AWS_WSD_ENDPOINT);
              _connection =
connectionFactory.CreateConnection(BrokerAccountInfo.ACTIVEMQ_USERNAME,
BrokerAccountInfo.ACTIVEMQ_PASSWORD);
              _connection.Start();
          }
          catch (Exception ex)
          {
              Console.WriteLine(ex);
              Console.WriteLine("Could not start listener. Press <ENTER> to exit.");
              Console.Read();
          }
      }

[…]
```

Figure 64. Broker app – Listener class constructor

JSON structure

There are two different messages that the mobile applications send to the broker app:

- Add measurement: This message encapsulates all the sensors measurements together with the timestamp, device Name, device MAC Address and session ID. You can find an example of an add measurement JSON below:

```
{
 "bleMACAddress":"0060370000127496",
 "sessionID":"75C31C",
"measurement":{"humidity":20.190000534057617,"temperature":29.360000610351562
,"light":11.159999847412109,"timestamp":1544544113,"pressure":1017},
 "name":"RPK-7496"
 }
```

- Clear database: This message contains as parameter the session ID. When the broker application receives this kind of message, it deletes all the measurements from the database which contains that particular session ID. You can find an example of a Clear database JSON below:

```
{
 "SessionID":"0DFE17"
}
```

How to process a measurement and store it in the database

As previously explained, the broker app is subscribed to different topics in the AmazonMQ server:

- weather-station-iOS: all the JSON messages sent from any iOS application are received in this topic.
- weather-station-android: all the JSON messages sent from any Android application are received in this topic.
- weather-station-clear-database: all Clear database messages are received here, no matter if they are coming from Android or iOS.

Let's look at an example to get a measurement from the AmazonMQ server, process it and store in the database.

First it is necessary to subscribe for the iOS topic in the broker application. Once subscribed, the method will be continually looping while waiting for new incoming messages in the topic. This can be seen in Figure 65.

```
public void registerIOSTopic()
{
      //Create a new session to subscribe to the iOS topic.
      ISession _session = _connection.CreateSession();
      IDestination dest = _session.GetTopic(IOS_DESTINATION_TOPIC);

      //Create a new consumer and start listening for incoming messages to the topic.
      using (IMessageConsumer consumer = _session.CreateConsumer(dest))
      {
          Console.WriteLine("iOS Listener started.");
          Console.WriteLine("iOS Listener created.rn");
          IMessage message;

          //We will continuously check for new messages in this loop
          while (true)
          {
              message = consumer.Receive();

              //If there is a new message in the topic just read and process it.
              if (message != null)
              {
                  var bytesMessage = message as IBytesMessage;
                  if (bytesMessage != null)
                  {
                       byte[] content = bytesMessage.Content;
                       string result = System.Text.Encoding.UTF8.GetString(content);
                       if (!string.IsNullOrEmpty(result))
                       {
                         var receivedObject = ParseJSON.parseMeasurementJSON(result);
                             if (receivedObject != null){
                                DatabaseOperations.
                                checkUserLimitAndAddMeasurementToDatabase(receivedObject)
                             }
                        }
                   }
               }
           }
      }
  }
```

Figure 65. Subscribe and receive messages from the iOS topic

When a new message arrives, it is received as an IBytesMessage object. It is necessary to convert it to a string. Once the message has been converted to the desired format, we parse the JSON string and store it in a RapidIoTDevice object. The method to parse the JSON object is shown in Figure 66.

Finally, if the RapidIoTDevice object is not null, we can ensure there is a new message to store in the database and proceed to store it.

```
public static RapidIoTDevice parseMeasurementJSON(string json)
{
   //We need to check that the will message is not introduced here as would raise an
exception since it does not include any measure
   if (!string.Equals("/will", json)) {
       RapidIoTDevice wsdObjectReceived =
JsonConvert.DeserializeObject<RapidIoTDevice>(json);
       return wsdObjectReceived;
   }
   else {
      return null;
   }
}
```

Figure 66. Parse add measurement JSON object

There is a restriction in the functional requirements that each user has to be able to store at maximum 1 Mb of data. So before adding the value to the database, we check that this 1 Mb value hasn't been exceeded. In case the value is exceeded, the oldest measurement will be deleted and the new one added. This method can be seen in Figure 67.

The procedure to subscribe and process the measurements coming from the Android application is very similar. We just need to change the iOS topic name for the Android topic name.

```
//Before storing the incoming measurement we need to check if the user has reached its
maximum storage size,
        //to check this we count the number of entries in the database for that Session
ID, in case the max. limit is not
        //exceeded, we proceed to create a new query and store the measurement in the
database.
public static void checkUserLimitAndAddMeasurementToDatabase(RapidIoTDevice
rapidIoTDevice)
{
    using (SqlConnection con = new SqlConnection(BrokerAccountInfo.connectionString))
    {
         con.Open();
         using (SqlCommand command = new SqlCommand("SELECT COUNT(*) FROM
dbo.RapidIoTDevices " +
                    "WHERE SessionID = '" + rapidIoTDevice.bleMACAddress + "'", con))
        {
           //Get the number of entries in Int32 format.
           Int32 count = (Int32)command.ExecuteScalar();

           //In case the Max. number of entries is exceeeded we delete the last
measurement and store the new received measurement
           if (count >= maxNumberOfEntriesPerUser)
           {
               var deleteCommand = new SqlCommand("DELETE TOP 1 FROM dbo.RapidIoTDevices
WHERE bleMACAddress = '" +
                           rapidIoTDevice.bleMACAddress + "'", con);
               deleteCommand.ExecuteNonQuery();
            }

            if(rapidIoTDevice.SessionID != null)
            {
               SqlCommand insertCommand = new SqlCommand("INSERT INTO
dbo.RapidIoTDevices (Name, SessionID, Measurement_Temperature," +

"Measurement_Humidity,Measurement_Pressure,Measurement_Light,Measurement_Timestamp,bleMAC
Address) " + "Values (@Name,
@SessionID,@Meas_Temp,@Meas_Humid,@Meas_Pres,@Meas_Light,@Meas_Time,@bleMACAddress)"
                                          , con);
                insertCommand.Parameters.AddWithValue("@Name", rapidIoTDevice.Name);
                insertCommand.Parameters.AddWithValue("@SessionID",
rapidIoTDevice.SessionID);
                insertCommand.Parameters.AddWithValue("@Meas_Temp",
rapidIoTDevice.Measurement.Temperature);
                insertCommand.Parameters.AddWithValue("@Meas_Humid",
rapidIoTDevice.Measurement.Humidity);
                insertCommand.Parameters.AddWithValue("@Meas_Pres",
rapidIoTDevice.Measurement.Pressure);
                insertCommand.Parameters.AddWithValue("@Meas_Light",
rapidIoTDevice.Measurement.Light);
                insertCommand.Parameters.AddWithValue("@Meas_Time",
rapidIoTDevice.Measurement.Timestamp);
                insertCommand.Parameters.AddWithValue("@bleMACAddress",
rapidIoTDevice.bleMACAddress);
                insertCommand.ExecuteNonQuery();
             }

        }
    con.Close();
```

Figure 67. Add a measurement to the database

Regarding the Clear database command, as in the iOS and Android topics, it is necessary to subscribe to this topic. This process is shown in Figure 68.

Once we are subscribed to the topic, there is an infinite loop checking for new incoming messages in the Topic. When we receive a new message we check that it is not null and contains a valid 'Clear database' message.

```
public void registerClearDBTopic()
{
     //Create a new session to subscribe to the iOS topic.
     ISession _session = _connection.CreateSession();
     IDestination dest = _session.GetTopic(CLEARDB_DESTINATION_TOPIC);

     //Create a new consumer and start listening for incoming messages to the topic.
     using (IMessageConsumer consumer = _session.CreateConsumer(dest))
     {
         Console.WriteLine("Clear DB Listener started.");
         Console.WriteLine("Clear DB Listener created.rn");
         IMessage message;

         //We will continuously check for new messages in this loop
         while (true)
         {
             message = consumer.Receive();

             //If there is a new message in the topic just read and process it.
             if (message != null)
             {
                  var bytesMessage = message as IBytesMessage;
                  if (bytesMessage != null)
                  {
                      byte[] content = bytesMessage.Content;
                      string result = System.Text.Encoding.UTF8.GetString(content);
                      if (!string.IsNullOrEmpty(result))
                      {
                          //Parse the JSON with the clear database message to retrieve
the corresponding Session ID that will be used in the DatabaseOperations class.
                          var receivedObject = ParseJSON.parseDatabaseClearJSON(result);
                          if (receivedObject != null)
DatabaseOperations.deleteMeasurementsWithSessionID(receivedObject.SessionID);
                          }
                       }
                   }
               }
          }
    }
```

Figure 68. Subscribe to Clear database topic

Then we proceed to parse the message into a 'ClearCloud' object which contains the session ID. This can be seen in Figure 69.

```
public static ClearCloud parseDatabaseClearJSON(string json)
{
    ClearCloud databaseObjectReceived = JsonConvert.DeserializeObject<ClearCloud>(json);
    return databaseObjectReceived;
}
```

Figure 69. Parse clear database JSON

Finally, when the object has been retrieved, we introduce the session ID in the 'deleteMeasurementWithSessionID' method which is in charge of executing the corresponding SQL command. This is shown in Figure 70.

```
//The Session ID is included as parameter, we need to create an SQL query to delete from
the devices table all the measurements with the Session ID introduced as parameter.
public static void deleteMeasurementsWithSessionID(String mSessionID)
{
    //Create a connection with the broker messager
    using (SqlConnection con = new SqlConnection(BrokerAccountInfo.connectionString))
    {
        con.Open();
        using (SqlCommand deleteCommand = new SqlCommand("DELETE FROM dbo.RapidIoTDevices
WHERE SessionID = '" + mSessionID + "'", con))
        {
           deleteCommand.ExecuteNonQuery();
        }
       con.Close();
     }
}
```

Figure 70. Delete all measurements from database

Revision history

Table 2. Sample revision history

Revision number	Date	Substantive changes
1.0	01/2019	Initial release

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support