Real-Time Big Data Processing

Lab 1 - Getting Started with Event Hubs and IoT Hubs

Overview

In this lab, you will create an Azure Event Hub and use it to collect event data from a client application. You will then create an Azure IoT Hub and submit messages to it from a simulated Internet-of-Things (IoT) device.

What You'll Need

To complete the labs, you will need the following:

- A web browser
- A Microsoft account
- A Microsoft Azure subscription
- A Windows, Linux, or Mac OS X computer
- The lab files for this course

Note: To set up the required environment for the lab, follow the instructions in the Setup document for this course. Specifically, you must have signed up for an Azure subscription and installed Node.JS on your computer.

Capturing Events in an Azure Event Hub

In this exercise, you will create an Azure event hub to which applications can submit event details.

Note: The Microsoft Azure portal is continually improved in response to customer feedback. The steps in this exercise reflect the user interface of the Microsoft Azure portal at the time of writing, but may not match the latest design of the portal exactly.

Create a Service Bus Namespace

Before you can create event hubs, you must create a service bus namespace.

1. In a web browser, navigate to http://portal.azure.com, and if prompted, sign in using the Microsoft account that is associated with your Azure subscription.
2. In the Microsoft Azure portal, in the Hub Menu, click **New**. Then in the **Internet of Things** menu, click **Event Hubs**.
3. In the **Create namespace** blade, enter the following settings, and then click **Create**:
    - **Name**: *Enter a unique name (and make a note of it!)*
    - **Pricing tier**: Basic

- **Subscription**: *Select your Azure subscription*
- **Resource Group**: *Create a new resource group with a unique name*
- **Location:** *Select any available region*
- **Pin to dashboard**: *Not selected*

4. In the Azure portal, view **Notifications** to verify that deployment has started. Then wait for the service bus namespace to be deployed (this can take a few minutes.)

Create a Storage Account

In this lab, you will enable archiving for your event hub so that you can view the messages that it receives. To use event hub archiving, you require an Azure storage account:

1. In the Microsoft Azure portal, in the Hub Menu, click **New**. Then in the **Storage** menu, click **Storage account**.
2. In the **Create storage account** blade, enter the following settings and click **Create**:
    - **Name**: *Enter a unique name (and make a note of it!)*
    - **Deployment model**: Resource manager
    - **Account kind**: General purpose
    - **Performance**: Standard
    - **Replication**: Locally-redundant storage (LRS)
    - **Storage service encryption**: Disabled
    - **Subscription**: *Select your Azure subscription*
    - **Resource group**: *Use the existing resource group you created in the previous procedure*
    - **Location**: *Select the region where you created your service bus namespace*
3. In the Azure portal, view **Notifications** to verify that deployment has started. Then wait for the storage account to be deployed (this can take a few minutes.)

Add an Event Hub

Now that you have a storage account, you are ready to add an event hub to your service bus namespace.

1. In the Azure portal, browse to the service bus namespace you created previously.
2. In the blade for your service bus namespace, click **Add Event Hub**.
3. In the **Create Event Hub** blade, enter the following settings and click **Create**:
    - **Name**: *Enter a unique name (and make a note of it!)*
    - **Partition Count**: 2
    - **Message Retention**: 1
    - **Archive**: On
    - **Time window (minutes)**: 5
    - **Size window (MB)**: 300
    - **Container:** *Select the storage account you created in the previous procedure, and then add a container with the following settings, and select it after it has been created:*
        - **Name**: event-archive
        - **Access type**: Private
4. In the Azure portal, wait for the notification that the event hub has been created.

Create Shared Access Policy Settings

Access to event hubs is controlled using shared access policies.

1. In the Azure portal, in the blade for your service bus namespace, select the event hub you just created.

2. In the blade for your event hub, click **Shared access policies**.
3. On the **Shared access policies** blade for your event hub, click **Add**. Then add a shared access policy with the following settings:
    - **Policy name**: DeviceAccess
    - **Claim**: Manage
4. Wait for the shared access policy to be created, then select it, and note that primary and secondary connection strings have been created for it. Copy the primary connection string to the clipboard - you will use it to connect to your event hub from a simulated client device in the next exercise.

Run a Node.JS Script to Submit Events

Now that you have created an event hub, you can submit events to it from devices and applications. In this exercise, you will use a Node.JS script to simulate random device readings.

1. Open a Node.JS console and navigate to the **eventclient** folder in the folder where you extracted the lab files.
2. Enter the following command, and press RETURN to accept all the default options. This creates a package.json file for your application:

    ```
    npm init
    ```

3. Enter the following command to install the Azure Event Hubs package:

    ```
    npm install azure-event-hubs
    ```

4. Use a text editor to edit the **eventclient.js** file in the **eventclient** folder.
5. Modify the script to set the **connStr**, and **eventHub** variables to reflect your event hub, and shared access policy connection string, as shown here:

    ```
    var EventHubClient = require('azure-event-hubs').Client;

    var connStr = '<YOUR_CONNECTION_STRING>';
    var eventHub = '<YOUR_EVENT_HUB>'

    var client = EventHubClient.fromConnectionString(connStr, eventHub)
    client.createSender()
        .then(function (tx) {
            setInterval(function(){
              dev = 'dev' + String(Math.floor((Math.random() * 10) + 1));
              val = String(Math.random());
              console.log(dev + ": " + val);
              tx.on('errorReceived', function (err) { console.log(err); });
              tx.send({ device: dev, reading: val});
            }, 1000);
        });
    ```

6. Save the script and close the file.
7. In the Node.JS console window, enter the following command to run the script:

    ```
    node eventclient.js
    ```

8. Observe the script running as it submits simulated events. After a few events have been submitted, press CTRL+C to stop the script running.

View Archived Events

Archived events are stored in your Azure storage account.

1. Wait a few minutes for the events you submitted to be archived.
2. Start Azure Storage Explorer, and if necessary, sign into your azure subscription using your Microsoft account.
3. Expand your storage account, and then expand **Blob Containers**.
4. Double-click the **event-archive** container.
5. Double-click the folder for your service bus namespace, and then the folder for your event hub.
6. Browse the folders in the event hub folder to find the archive files that have been generated for your events. You can open these files to view them in a text editor (the files are in Avro format, so some characters may not display correctly in a text editor or browser).

Capturing Device Messages in an IoT Hub

In this exercise, you will create an IoT hub and submit simulated device readings to it.

Create an IoT Hub

In this procedure, you will use the Azure portal to create an IoT hub.

1. In the Microsoft Azure portal, in the Hub Menu, click **New**. Then in the **Internet of Things** menu, click **IoT Hub**.
2. In the **IoT Hub** blade, enter the following settings, and then click **Create**:
    - **Name**: *Enter a unique name (and make a note of it!)*
    - **Pricing and scale tier**: Free
    - **IoT Hub Units**: 1
    - **Device-to-cloud partitions**: 2 partitions
    - **Subscription**: *Select your Azure subscription*
    - **Resource Group**: *Select the resource group you created previously*
    - **Enable Device management**: *Leave unselected*
    - **Location:** *Select any available region*
    - **Pin to dashboard**: *Not selected*
3. In the Azure portal, view **Notifications** to verify that deployment has started. Then wait for the service bus namespace to be deployed (this can take a few minutes.)

Get the Hostname and Connection String for the Hub

IoT hubs use shared access policies to secure connections from client devices. You must use an appropriate connection string in client-side code when connecting from a device.

1. In the Azure portal, browse to the IoT Hub you created previously.
2. In the blade for your IoT Hub, make a note of the **Hostname** for your IoT hub – you will need this later. Then click **Shared access policies**.
3. In the **Shared access policies** blade, click the **iothubowner** policy, and then make a note of the **connection string-primary key** for your Iot Hub – you will need this later.

> **Note**: For more information about access control for IoT hubs, see Access control in the "Azure IoT Hub developer guide."

Create a Device Identity

Each device that sends data to the IoT hub must be registered with a unique identity.

1. Open a Node.JS console and navigate to the **createdeviceid** folder in the folder where you extracted the lab files.
2. Enter the following command, and press RETURN to accept all the default options. This creates a package.json file for your application:

   ```
   npm init
   ```

3. Enter the following command to install the Azure IoT Hub package:

   ```
   npm install azure-iothub --save
   ```

4. Use a text editor to edit the **createdeviceid.js** file in the **createdeviceid** folder.
5. Modify the script to set the **connStr** variable to reflect the shared access policy connection string for your IoT Hub, as shown here:

   ```
   'use strict';

   var iothub = require('azure-iothub');

   var connStr = '<IOT-HUB-CONNECTION-STRING>';

   var registry = iothub.Registry.fromConnectionString(connStr);

   var device = new iothub.Device(null);
   device.deviceId = 'MyDevice';
   registry.create(device, function(err, deviceInfo, res) {
     if (err) {
       registry.get(device.deviceId, printDeviceInfo);
     }
     if (deviceInfo) {
       printDeviceInfo(err, deviceInfo, res)
     }
   });

   function printDeviceInfo(err, deviceInfo, res) {
     if (deviceInfo) {
       console.log('Device id: ' + deviceInfo.deviceId);
       console.log('Device key: ' +
   deviceInfo.authentication.SymmetricKey.primaryKey);
     }
   }
   ```

6. Save the script and close the file.
7. In the Node.JS console window, enter the following command to run the script:

   ```
   node createdeviceid.js
   ```

8. Verify that the script registers a device with the ID *MyDevice*, and make a note of the device key generated by the script – you will need this later.

Submit Messages to the IoT Hub

1. In the Node.JS console, navigate to the **iotdevice** folder in the folder where you extracted the lab files.
2. Enter the following command, and press RETURN to accept all the default options. This creates a package.json file for your application:

   ```
   npm init
   ```

3. Enter the following command to install the Azure IoT device and AMQP protocol packages:

```
npm install azure-iot-device azure-iot-device-amqp --save
```

4. Use a text editor to edit the **iotdevice.js** file in the **iotdevice** folder.
5. Modify the script to set the **connStr** variables to reflect your IoT hub hostname and the device key that was generated when you registered the device in the previous exercise, as shown here:

```
'use strict';

var clientFromConnectionString = require('azure-iot-device-
amqp').clientFromConnectionString;
var Message = require('azure-iot-device').Message;

var connStr = 'HostName=<HOSTNAME>;DeviceId=MyDevice;SharedAccessKey=<DEVKEY>';

var client = clientFromConnectionString(connStr);

function printResultFor(op) {
  return function printResult(err, res) {
    if (err) console.log(op + ' error: ' + err.toString());
    if (res) console.log(op + ' status: ' + res.constructor.name);
  };
}

var connectCallback = function (err) {
  if (err) {
    console.log('Could not connect: ' + err);
  } else {
    console.log('Client connected');

    // Create a message and send it to the IoT Hub every second
    setInterval(function(){
        var r = Math.random();
        var data = JSON.stringify({ device: 'MyDevice', reading: r });
        var message = new Message(data);
        console.log("Sending message: " + message.getData());
        client.sendEvent(message, printResultFor('send'));
    }, 1000);
  }
};

client.open(connectCallback);
```

6. Save the script and close the file.
7. In the Node.JS console window, enter the following command to run the script:

```
node iotdevice.js
```

8. Observe the script running as it starts to submit device readings. Then leave the script running and start the next procedure.

Read Messages from the IoT Hub

When devices submit messages to your IoT hub, you can read the messages from an event hub endpoint that gets created automatically.

1. Open a new Node.JS console, and navigate to the **iotreader** folder in the folder where you extracted the lab files.

2. Enter the following command, and press RETURN to accept all the default options. This creates a package.json file for your application:

```
npm init
```

3. Enter the following command to install the Azure IoT Hub package:

```
npm install azure-event-hubs --save
```

4. Use a text editor to edit the **iotreader.js** file in the **iotreader** folder.
5. Modify the script to set the **connStr** variable to reflect the shared access policy connection string for your IoT Hub, as shown here:

```
'use strict';

var EventHubClient = require('azure-event-hubs').Client;

var connStr = '<IOT-HUB-CONNECTION-STRING>';

var printError = function (err) {
  console.log(err.message);
};

var printMessage = function (message) {
  console.log('Message received: ');
  console.log(JSON.stringify(message.body));
  console.log('');
};

var client = EventHubClient.fromConnectionString(connStr);
client.open()
    .then(client.getPartitionIds.bind(client))
    .then(function (partitionIds) {
        return partitionIds.map(function (partitionId) {
            return client.createReceiver('$Default', partitionId, {
'startAfterTime' : Date.now()}).then(function(receiver) {
                console.log('Created partition receiver: ' + partitionId)
                receiver.on('errorReceived', printError);
                receiver.on('message', printMessage);
            });
        });
    })
    .catch(printError);
```

6. Save the script and close the file.
7. In the Node.JS console window, enter the following command to run the script:

```
node iotreader.js
```

8. Observe the script running as it reads the device readings that were submitted to the IoT hub by the **iotdevice.js** script.
9. After the scripts have been running for a while, stop them by pressing CTRL+C in each of the console windows. Then close the console windows.

Note: You will use the resources you created in this lab when performing the next lab, so do not delete them. Ensure that all Node.js scripts are stopped to minimize ongoing resource usage costs.