CHAPTER 7

IoT and Microservices

There are 2 billion PCs in use today across the globe. There are over 10 billion mobile phones. By 2020, it is predicted that there will be over 250 billion devices connected to the Internet. Some of these devices will be new products, but most will be existing things we use every day that will be enhanced with sensors, such as thermostats, cars, eyeglasses, wrist watches, clothing, street lamps, cars, buildings…you name it, it will likely become connected.

Each of these devices will be gathering data through sensors and sending data to the cloud. The amount of data that will be collected will be measured in petabytes, exabytes, and zettabytes. In other words, IoT is not just about devices but also about data, a lot of data. The reason that we want to collect all this data is to extract knowledge, to provide real-time visualization and data feeds, and to perform historical and predictive analytics that will drive business decisions at velocity and provide real-time notification and status.

IoT Capabilities

To fully realize an IoT solution, several capabilities will be required. These capabilities include the following:

Device Management: The device, upon initialization, will want to establish a relationship with the cloud environment, usually through its unique identifier, such as a serial number, so that the business is notified that the device is active. The business will also want the ability to send commands to the device for the purposes of providing software updates or updating local data caches.

Telemetry Ingestion: Devices may be sending multiple messages a second, and there may be hundreds to thousands of devices or more, which would result in 10's of thousands to possibly millions of messages a day. The cloud platform provides high-volume message ingestion using a single logical endpoint.

Transformation and Storage: Once the messages arrive, the cloud provides a mechanism to select, transform, and route messages to various storage mediums for the purpose of archival and staging for downstream processing.

Status and Notifications: The cloud solution will want to provide the ability to visualize the status of the message pool in real time through tabular or graphical UI components. In addition, some messages may contain information of an alert status so the IoT solution must provide a mechanism for real-time notifications.

Analytics and Data Visualization: The value of collecting so much data in a continuous fashion is to build up an historical record for the purpose of performing analytics to glean business insight. Traditional data warehouse techniques or more modern map-reduce and predictive analytics mechanisms can be employed.

Azure IoT Services

Microsoft provides you two approaches to realizing your IoT solutions:

- Custom Development – build from scratch using a combination of IoT Hub, Stream Analytics and Event Hub along with other Azure resources, custom configuration and code to deliver a complete product
- Scripted Scenarios – leverage pre-scripted starter configurations for business scenarios such as remote monitoring and predictive maintenance and combine with custom configuration and code to create a finished product

Custom Development

The custom development approach will leverage Azure IoT Hub, Azure Stream Analytics and Azure Event Hub for device management, telemetry ingestion, transformation and routing. The Home Biomedical Reference Implementation is an example of this custom development approach. Its use of Event Hub for telemetry ingestion from the home biomedical devices and Stream Analytics for message transformation, alarm state identification and routing is detailed later in this chapter. First let's take a look at the newest service available from Microsoft for IoT called IoT Hub.

IoT Hub

In October 2015, Microsoft announced the general availability of IoT Hub. IoT Hub is a fully managed service that enables:

- Reliable device-to-cloud and cloud-to-device hyper-scale messaging
- Secure communications using per-device security credentials and access control
- Device libraries for popular languages and platforms

IoT Hub provides device registration, command and control and symmetric key management for secure authentication on a per-device basis. To provision IoT Hub, from the Azure Portal click New (+), Internet of Things, Azure IoT Hub. The IoT Hub creation blade appears. The default configuration uses the S1 pricing and scale tier and defines 1 unit of scale. Scaling is done by entering a number of units where each unit supports up to 500 devices. You can have up to 200 units for a maximum of 100K devices per IoT Hub and the ability to ingest 50K messages per day. The S2 pricing tier provides up to 1.5 million messages per day (see Figure 7-1).

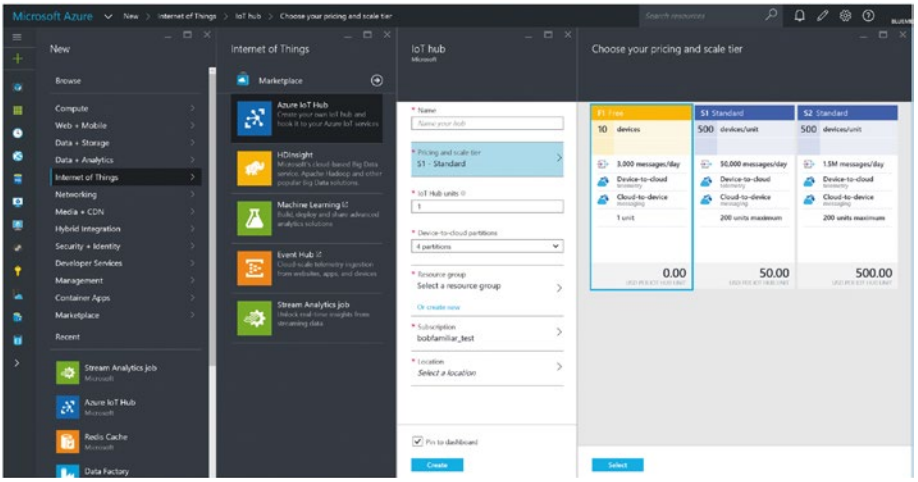

Figure 7-1. *IoT Hub Creation Blade*

Once the IoT Hub is provisioned, you can register devices with the hub so that they can authenticate and send and receive messages. The device provisioning process will be unique to your business and may involve integration with existing systems to align serial numbers, customer information, etc. For demonstration purposes, a sample device registration console application is provided that leverages the ConfigM and DeviceM microservices to register the existing 300 Home Biomedical devices with IoT Hub.

■ **Note** To use this console app, you will need to provision an IoT Hub and update the sample with the connection string information. The sample solution can be found in IoTHub\IoTHubDeviceRegistration.

The IoT Hub connections string information can be found by clicking Settings, Shared Access Policies and selecting the policy of interest (see Figure 7-2). The sample application uses the 'iothubowner' policy.

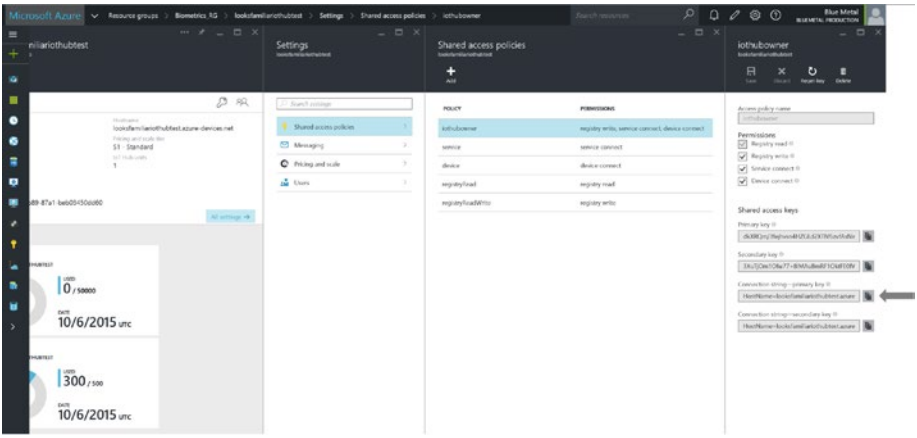

Figure 7-2. *IoT Hub Connection String Blade*

To connect to IoT Hub and register a device, you need to reference the Micrososft Azure Devices NuGet package. In Visual Studio, select the Tools menu, NuGet Package Manager, Package Manager Console and type in this command:

```
> Install-Package Microsoft.Azure.Devices -Pre
```

In code, create an IoT Hub RegistryManager object passing in the connection string from the App.Config file and call the AddDeviceAsync() method passing in a unique id for the device.

```
// initialize the IoT Hub registration manager
RegistryManager registryManager;
registryManager = RegistryManager.CreateFromConnectionString(
        ConfigurationManager.AppSettings["IoTHubConnStr"]);

// register a device
Device device;
device = await registryManager.AddDeviceAsync(new Device("MyDeviceId"));
```

Once the devices have been registered, you can see the number of devices in the IoT Hub registry on the IoT Hub management blade (see Figure 7-3).

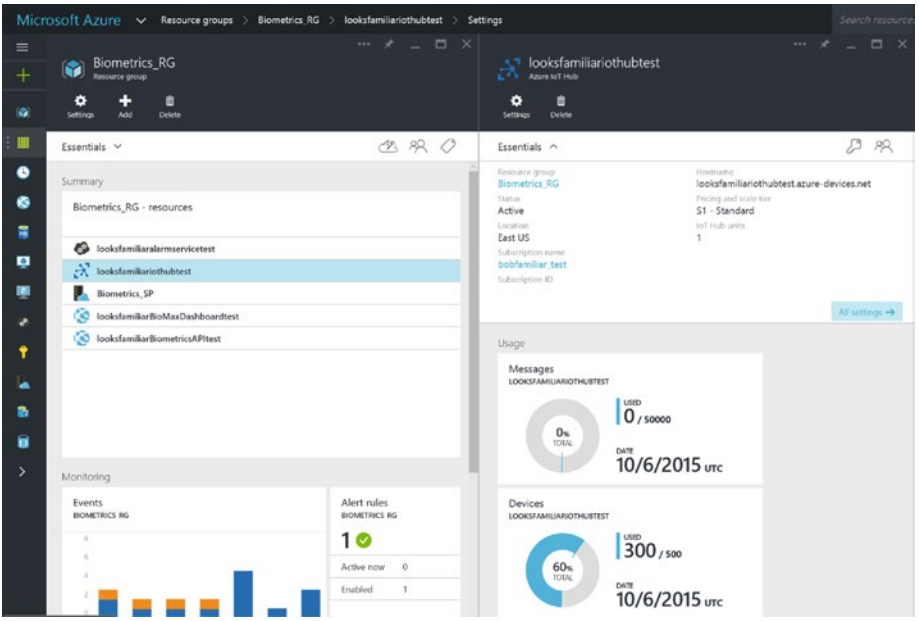

Figure 7-3. *IoT Hub Blade showing 300 registered devices*

Once devices are registered, they can make secure connections to IoT Hub and send and receive messages. IoT device SDKs are available and supported for a variety of languages and platforms including C for Linux distributions, Windows, and RTOS and managed languages such as C#, Java, and JavaScript.

If your solution cannot use the device SDKs, IoT Hub exposes a public protocol that enables devices to use the HTTP 1.1 and AMQP 1.0 protocols. Using the Azure IoT Protocol Gateway component, you can also extend IoT Hub to provide support for MQTT v3.1.1. You can run the Azure IoT Protocol Gateway in the cloud or on premises, and extend it to support custom protocols.

■ **Note** The Azure IoT Protocol Gateway can be found on GitHub: `https://github.com/Azure/azure-iot-protocol-gateway`.

In order to connect to IoT Hub and send messages, you need to reference the Micrososft Azure Devices Client NuGet package. In Visual Studio, select the Tools menu, NuGet Package Manager, Package Manager Console and type in this command:

```
> Install-Package Microsoft.Azure.Devices.Client -Pre
```

The client SDK will use the IoT Hub Uri along with the symmetric key assigned the device to make the secure connection. The Uri can be found on the IoT Hub Blade and has the format [iot-hub-name].azure-devices.net.

```
// get the device from the registry
device = await _registryManager.GetDeviceAsync("MyDeviceId");

// create a connection to the IoT Hub using the Uri and the symmetric key
DeviceClient client= DeviceClient.Create(
    ConfigurationManager.AppSettings["IoTHubUri"],
    new DeviceAuthenticationWithRegistrySymmetricKey(
            "MyDeviceId",
            device.Authentication.SymmetricKey.PrimaryKey));
```

Sending a message to IoT Hub is now straight forward. You collect the sensor readings of interest and call the SendEventAsync() method of the DeviceClient class:

```
Client.SendEventAsync(new Message(Encoding.ASCII.GetBytes(json))).Wait();
```

■ **Note** There is a version of the BioMax Simulator that demonstrates connecting and sending messages to IoT Hub located in IoTHub\BioMaxSimulator-IoTHub.

Scripted Scenario

IoT Suite is a solution-focused offering from Microsoft that provides a point and click approach to provisioning a starter kit for various IoT scenarios. Microsoft provides two scripted scenarios at the time of this writing:

- Remote Monitoring Solution – Provides device management, alerting and notification, telemetry ingestion, data visualization and device geolocation.
- Predictive Maintenance – Using Azure IoT capabilities along with Azure Machine Leaning, provides failure prediction, failure detection, failure type classification, and recommendation of mitigation or maintenance actions after failure.

IoT Suite

To provision an IoT Suite solution, you will need an Azure subscription and then visit https://www.azureiotsuite.com/. From this page you can provision a new solution. As you can see in Figure 7-4, I have already provisioned a Remote Monitoring solution. If I click on the tile, I can get links to the GitHub repository from which the solution was provisioned and guidance on how to customize. I can also de-provision the solution right from this page.

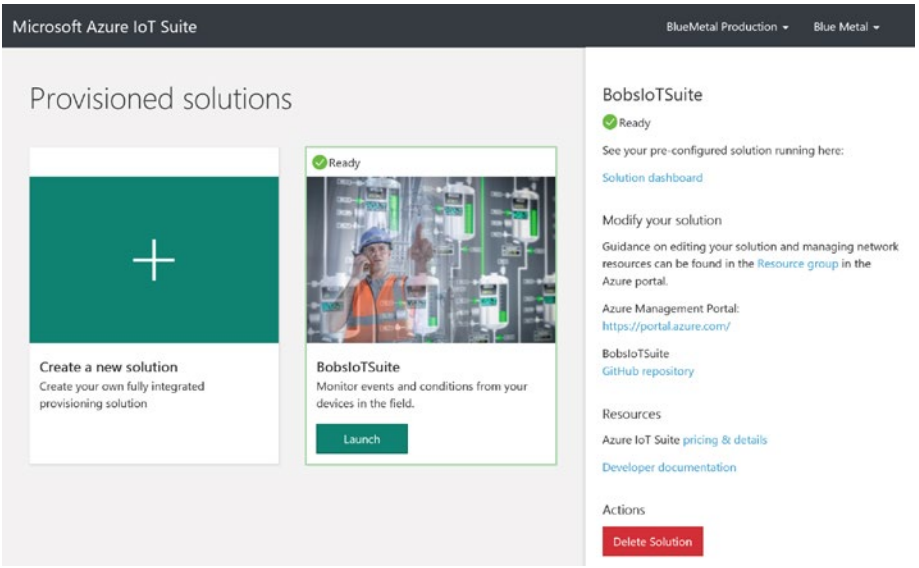

Figure 7-4. *Azure IoT Suite Landing Page*

If I click the 'Launch' button, I am brought to the Dashboard. From here I can see a list of simulated provisioned devices, data streaming from those devices, a map depicting where they are physically located and a menu on the right that provides access to forms for updating the ingestion rules for alerts. In addition there is an add device button (+) in the lower left hand corner to provision additional devices (see Figure 7-5).

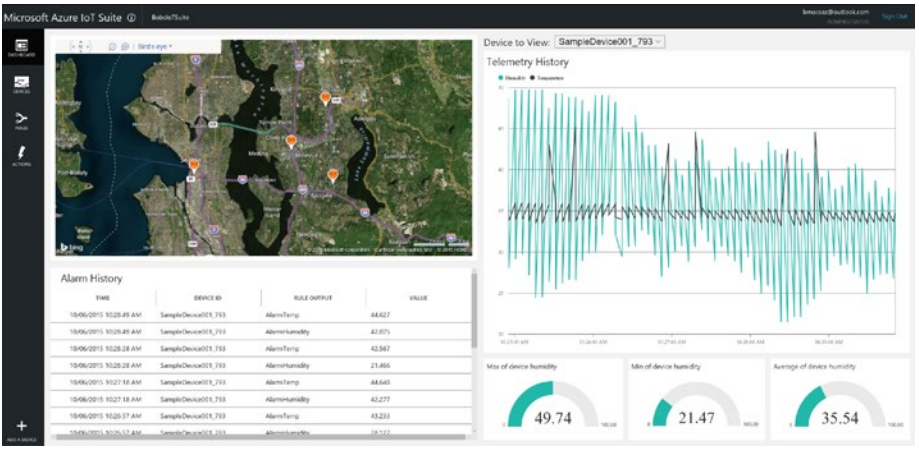

Figure 7-5. *Azure IoT Suite Dashboard*

In the Azure Portal (see Figure 7-6), you will find a new resource group has been created and all of the Azure resources associated with this solution are listed there including an IoT Hub, a DocumentDb database, an Event Hub and three Stream Anlatyics Jobs which you can edit at will. Also, as noted before, you have complete access to the source code and PowerShell scripts for the generated solution on GitHub so that you can configure, customize and extend as needed.

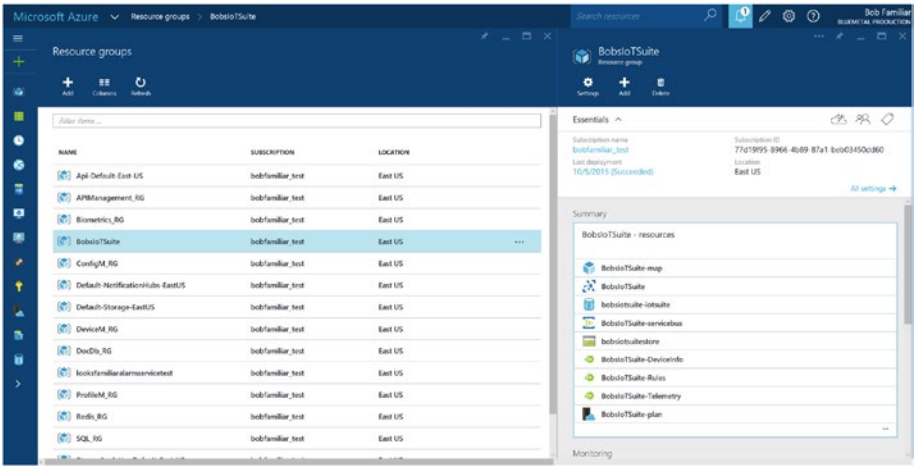

Figure 7-6. *Resource Group listing provisoned services*

The Home Biomedical Reference Implementation, in its current form, demonstrates a custom development approach using Event Hub and Stream Analytics. In the next section of the book, we delve into the details of the reference implementation's IoT capabilities.

The Reference Implementation IoT Capabilities

The Home Biomedical Reference Implementation provides an example of how one can incorporate IoT capabilities into a larger solution. The Reference Implementation uses Microsoft's IoT stack, consisting of Event Hub and Stream Analytics for telemetry ingestion, data transformation, and routing to SQL Database. Real-Time notifications are provided using Event Hub, a custom Event Hub Consumer Cloud Service called Biometrics Alarm Worker, and Notification Hub. Real-time data visualization is provided through a custom API combined with SignalR, which uses Web Sockets to push updates to a web front end (see Figure 7-7).

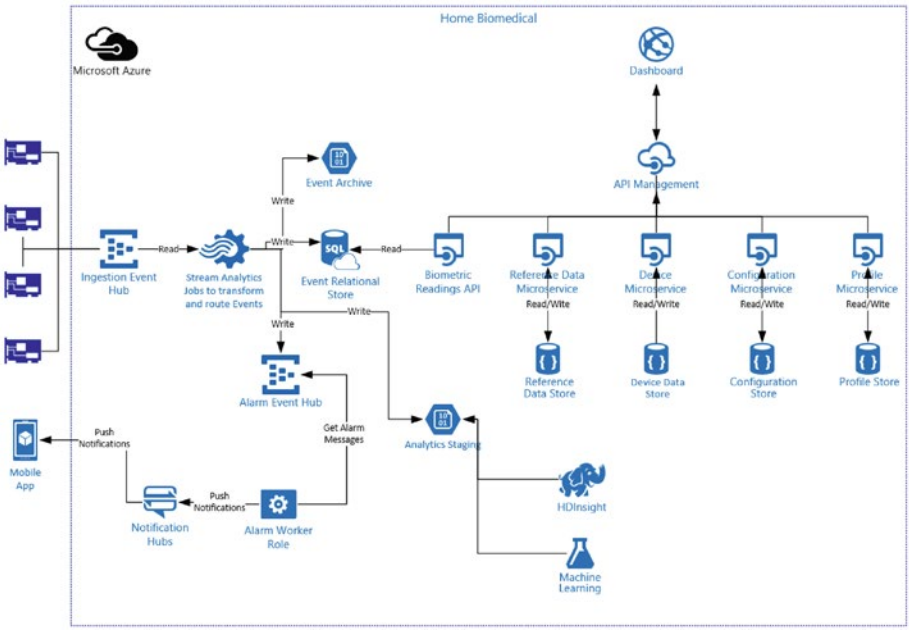

Figure 7-7. *Home Biomedical Microservice Architecture*

Device Management

The DeviceM provides a device registry for provisioning and associating devices with patients and/or participants in pharmaceutical trials. The administrative API provides create, update, and delete operations as well as a get all, which returns all registrations in the store. The public API defines `get by id`, which is the serial number of the device, `get by participant id`, which is the person the device is assigned to, and `get by model`, which returns all registrations for a device of a particular model (see Figure 7-8).

Public API

VERB	ROUTE	NOTES
GET	device/registrations/id/{id}	Get a device registration by device id (serial number)
GET	device/registrations/participant/{id}	Get a device by participant id (assignment to end user)
GET	device/registrations/model/{model}	Get all registrations by model

Admin API

VERB	ROUTE	NOTES
GET	device/registrations	Get all device registrations
POST	device/registrations	Create (provision a new device)
PUT	device/registrations	Update a device registration
DELETE	device/registrations/id/{id}	Delete a device registration

Figure 7-8. *DeviceM API*

The DeviceM model is called Registration. A device registration contains the device serial number (id), product line, model, and version and firmware revision. In addition, the id of the patient or participant is stored at the time the device is provisioned (see Figure 7-9).

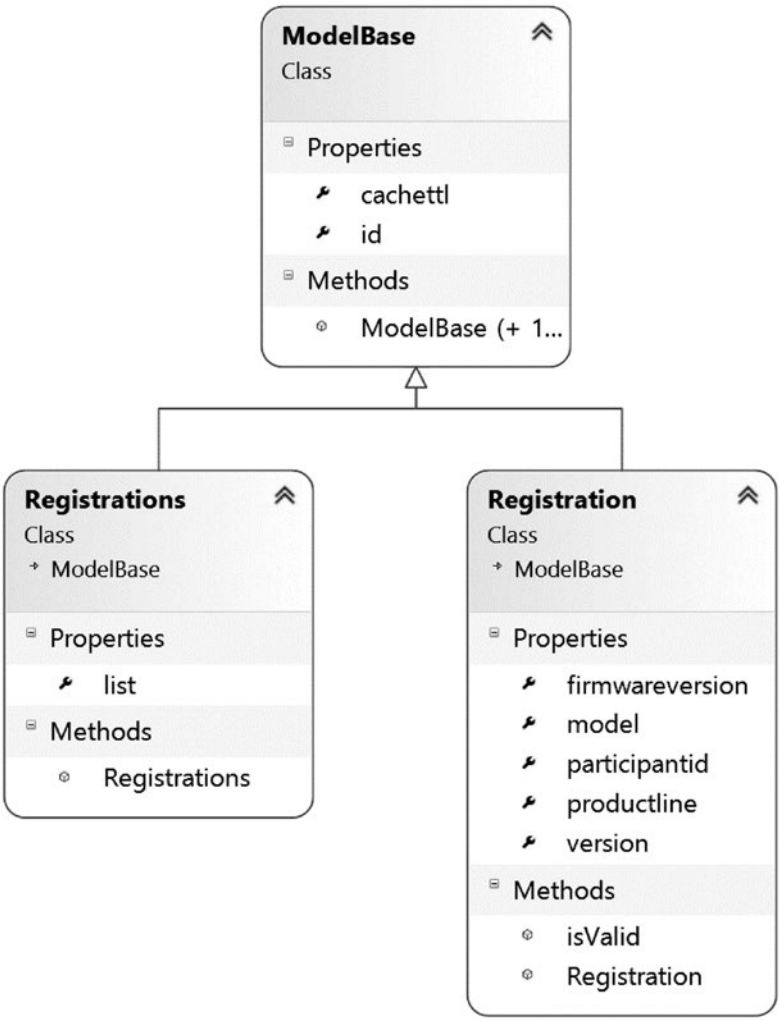

Figure 7-9. *DeviceM Model Registration*

■ **Note** The solutions related to the DeviceM microservice can be found in Microservices\Device.

Telemetry Ingestion

Event Hubs is a highly scalable publish-subscribe event ingestor that can intake millions of events per second so that you can process and analyze the massive amounts of data produced by connected devices and applications. Event Hub is configured with some number of partitions, each partition being able to ingest up to 1MB of data per second. By default, Event Hub is configured with 4 partitions. You can only specify the number of partitions at create time. The value can be set to as low as 2 or as high as 32.

Event Hub partitions are able to ingest up to 1MB of data or 1,000 events per second, whichever state is arrived at first. In high-volume telemetry ingestion scenarios, 1,000 messages usually come first because most messages are small. An Event Hub is created with 4 partitions by default. That value can be set to as low as 2 and as high as 32 but only at Event Hub creation. You can't change the number of partitions after the fact. Event Hub is available in basic and standard modes. Both modes provide the same throughput capabilities. Standard mode supports more consumer groups, brokered connections, and additional storage.

A partition is an ordered sequence of events that is held in a repository (see Figure 7-10). As newer events arrive, they are added to the end of this sequence. Events are kept in the repository for a length of time that is configurable. The default is 1 day but it can be set up to 7 days; 1 to 3 days is customary. Once a message's time-to-live has expired, it is removed from the Event Hub repository.

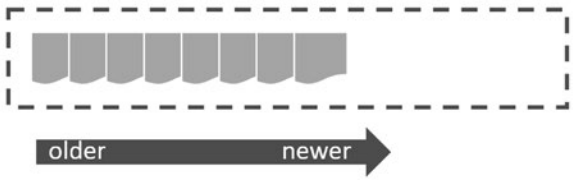

Figure 7-10. *Event Hub Partition Model*

The BioMax-Home Device Simulator

In order to test IoT services, it is necessary to develop an event simulator. Event simulators allow the team responsible for the cloud services to move forward with their development when the devices themselves are not available or do not yet exist. The simulators generate sample telemetry and exercise device provisioning, firmware downloads, and other command and control operations.

Developing device simulators with Event Hub is very straightforward. You use the Service Bus client SDK and add the connection information supplied in the Azure portal to define configuration settings for the endpoint and the name of the Event Hub. You create an object that represents the message you want to send, like sensor readings for a device, fill the object with simulated sensor-reading data, serialize the message to JSON, and send it to the endpoint using the client SDK.

The BioMaxSimulator solution uses the ConfigM Public SDK to look up the locations of the ProfileM Public API and the DeviceM Admin API. The DeviceM Admin SDK is initialized with the endpoint for that service and is used to retrieve the entire device registry. It does this so it can simulate readings coming from the 300 participants in the pharma trial.

```
// instantiate the SDK clients
_config = new ConfigM();
_registry = new DeviceM();
_profiles = new ProfileM();

// get the URL to ConfigM service from the config file
_config.ApiUrl = ConfigurationManager.AppSettings["ConfigM"];

// lookup the manifests for the
// DeviceM and ProfileM microservices
var deviceManifest = _config.GetByName("DeviceM");
var profileManifest = _config.GetByName("ProfileM");

// retrieve their API locations
_registry.ApiUrl = deviceManifest.lineitems[LineitemsKey.AdminAPI];
_profiles.ApiUrl = profileManifest.lineitems[LineitemsKey.PublicAPI];

// get the device registry from the device microservice
_devices = _registry.GetAll();
```

The configuration settings for Service Bus and Event Hub are read from configuration and the Event Hub client is initialized:

```
var bus = ConfigurationManager.AppSettings["servicebus"];
var hubname = ConfigurationManager.AppSettings["eventhub"];
var hub = EventHubClient.CreateFromConnectionString( bus, hubname);
```

The `DeviceMessage` class is used to construct the JSON messages that will be sent to the Event Hub (see Figure 7-11). The class contains the id of the device, the id of the participant that is using the device, the longitude and latitude of where the device is located, a timestamp of when the sensor readings were taken, and a list of sensor readings. The device will take four readings: Glucose, Heart Rate, Temperature, and Blood Oxygen levels as defined by the SensorType enum. This simulator will generate sample readings for these four biometrics.

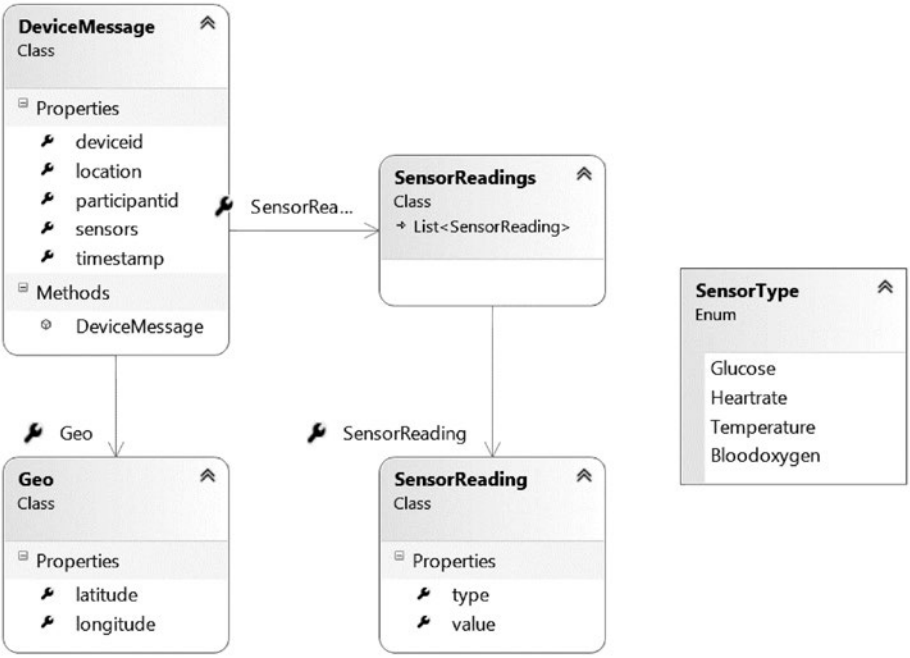

Figure 7-11. *The DeviceMessage Class*

This data model will serialize to JSON as follows:

```
{
  "deviceid": "03015126-aef7-49a3-9a01-1946d98e1383",
  "participantid": "cd57ce66-2065-4bdc-b4d3-ecfb0a5a704f",
  "location": { "longitude": -71.063562, "latitude": 42.290349 },
  "sensors": [
    { "type": 0, "value": 182.0 },
    { "type": 1, "value": 97.0 },
    { "type": 2, "value": 103.0 },
    { "type": 3, "value": 84.0 }
  ],
 "timestamp":"2015-07-13T16:42:16.6125201-04:00"
}
```

The device simulator program enters a loop and generates simulated readings several times a second. The messages are serialized and sent to Event Hub.

```
while (true)
{
    try
    {
        var deviceReading = new DeviceMessage();

        // randomly select a device from the registry
        var device = _devices.list[random.Next(0, 299)];

        // lookup the participant from the profile microservice
        var participant = _profiles.GetById(device.participantid);

        deviceReading.deviceid = device.id;
        deviceReading.participantid = participant.id;

        deviceReading.location.latitude = participant.location.latitude;
        deviceReading.location.longitude = participant.location.longitude;

        // generate simulated sensor reaings
        var glucose = new SensorReading
        {
            type = SensorType.Glucose,
            value = random.Next(70, 210)
        };

        var heartrate = new SensorReading
        {
            type = SensorType.Heartrate,
            value = random.Next(60, 180)
        };

        var temperature = new SensorReading
        {
            type = SensorType.Temperature,
            value = random.Next(98, 105) + (.1 * random.Next(0, 9))
        };

        var bloodoxygen = new SensorReading
        {
            type = SensorType.Bloodoxygen,
            value = random.Next(80, 100)
        };

        deviceReading.sensors.Add(glucose);
        deviceReading.sensors.Add(heartrate);
```

```
            deviceReading.sensors.Add(temperature);
            deviceReading.sensors.Add(bloodoxygen);

            deviceReading.timestamp = DateTime.Now;

            // serialize the message to JSON
            var json = ModelManager.ModelToJson<DeviceMessage>(deviceReading);
            // send the message to EventHub
            eventHubClient.Send(new EventData(Encoding.UTF8.GetBytes(json)));
        }
        catch (Exception exception)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine("{0} > Exception: {1}", DateTime.Now,
            exception.Message);
            Console.ResetColor();
        }

        Thread.Sleep(100);
    }
```

This code is meant to simulate the code executing on a device. In the real world, many of these devices are running a non-Windows OS such as Linux or Linux variants and the code would most likely be written in C. Microsoft provides a C library for Event Hub using the AMQP protocol and has expanded the number of client libraries with the recent release of IoT Hub. Note that Windows 10 IoT is now available and Microsoft licenses that OS for free on physical devices that are 9 inches or less in diameter.

■ **Note** To review the simulator source code refer to the following solution: `Microservices\Biometrics\Simulator\BioMaxSimulator`.

Telemetry Transformation and Storage

Stream Analytics provides low-latency, highly available, elastic event processing over streaming data. Stream Analytics marries extremely well with Event Hub, allowing you to connect to and consume events in the repository based on the properties and values in the JSON message as well as temporal properties such as arrival time. Once messages are selected, they can be directed to one or more storage locations such as Blob Storage, Table Storage, DocumentDb and SQL Database, or sent to another Event Hub for further processing.

To get started with Stream Analytics, you create and configure one or more Stream Analytics jobs (see Figure 7-12). You can do this in either the Classic Portal or the Preview Portal. When creating a job, you specify a unique name, the region the job runs in, and a monitoring storage location.

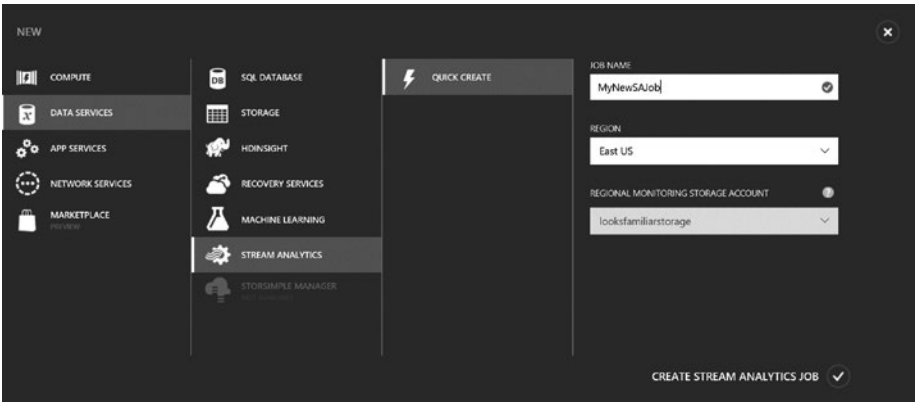

Figure 7-12. *Create Stream Analytics Job*

From the Azure Portal, you can then configure the input, output, and query settings for the Stream Analytics job (see Figure 7-13). Sources of data input can come from Event Hubs or Blob Storage. When defining an input, you provide an alias that will be used in the query ('input' for example). You can also configure the format of the incoming messages, specifying JSON, CSV, or Avro. Avro is a compact and efficient binary file format that leverages JSON for describing Hadoop MapReduce data sets.

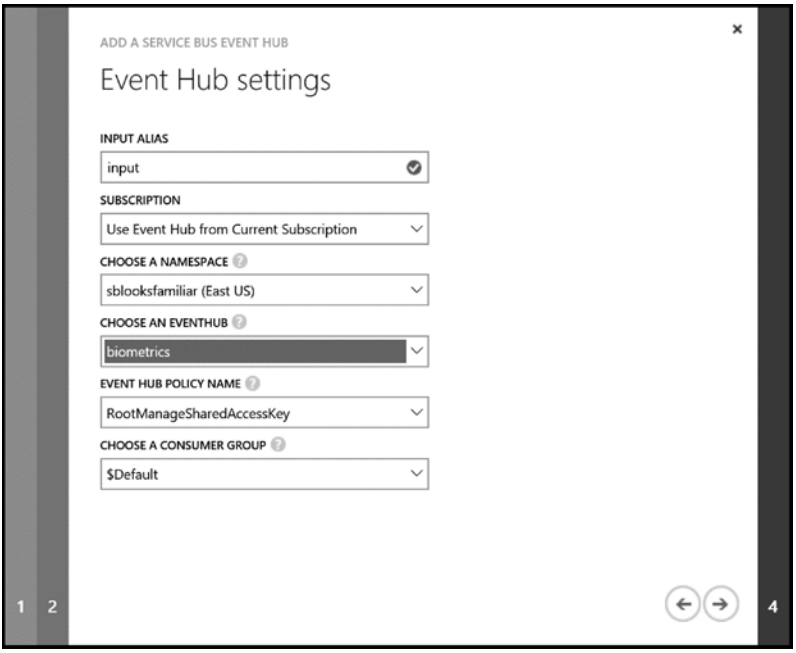

Figure 7-13. *Stream Analytics Job Input Settings for Event Hub*

When you define an output, you provide an alias and then select an output target. The current set of Stream Analytics Outputs includes SQL Database, DocumentDb, Table Storage, Blob Storage, PowerBI, Event Hub, Service Bus queues, and Service Bus topics (see Figure 7-14).

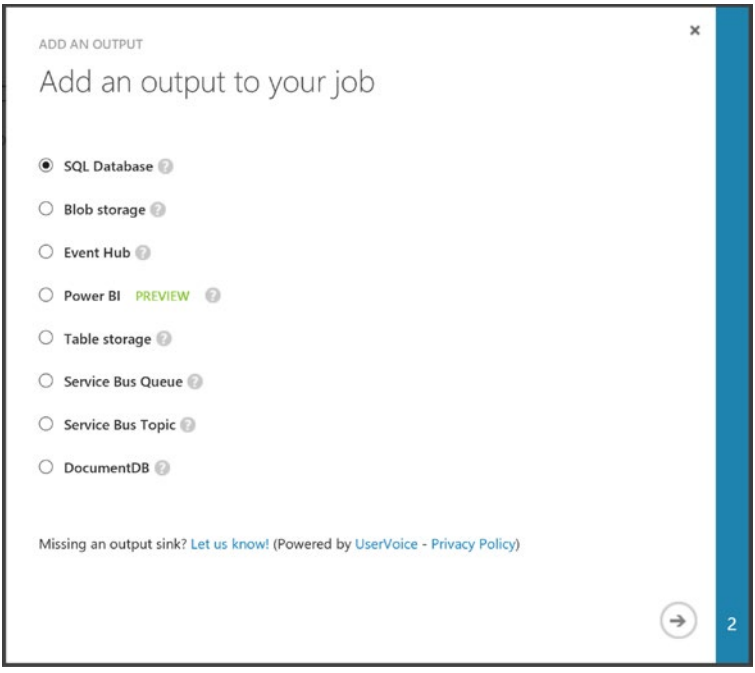

Figure 7-14. *Stream Analytics Job Output Settings*

When configuring SQL Database output, you will be asked to provide the database table name and the login credentials for the database. Note that the table definition in SQL Database must match the columns being selected in the query. In addition, the table must be defined with a clustered index.

Here is the DDL for the SQL Database table that is used by the Reference Implementation:

```
CREATE TABLE[dbo].[biometrics] (
       [deviceid] [char](256) NOT NULL,
       [participantid] [char](256) NOT NULL,
       [longitude] float NOT NULL,
       [latitude] float NOT NULL,
       [reading] datetime NOT NULL,
       [type] bigint NOT NULL,
       [value] float NOT NULL)

CREATE CLUSTERED INDEX[biometrics] ON[dbo].[biometrics] ( [deviceid] ASC )
```

Stream Analytics Queries

Stream Analytics queries are SQL syntax statements that are able to select events based on criteria that includes values in the event, time, and the particular partition where they reside. The Reference Implementation defines six queries:

biometrics-blob: Grab all incoming device messages and send to blob storage using a CSV file format.

biometrics-store: Grab all incoming device messages and send to SQL Database for downstream application integration.

glucose-alarms: Grab only messages that have a glucose reading that is out of bounds and send to the alarms Event Hub endpoint.

heartrate-alarms: Grab only messages that have a heart rate reading that is out of bounds and send to the alarms Event Hub endpoint.

temperature-alarms: Grab only messages that have a temperature reading that is out of bounds and send to the alarms Event Hub endpoint.

bloodoxygen-alarms: Grab only messages that have a blood oxygen reading that is out of bounds and send to the `alarms` Event Hub endpoint.

Each query has a similar structure. Let's looks at one of the alarm queries and dissect its function.

```
1   WITH Device as (SELECT * from input)
2   SELECT
3      Device.deviceid,
4      Device.participantid,
5      Device.location.longitude,
6      Device.location.latitude,
7      Device.timestamp,
8      DeviceSensors.ArrayValue.type,
9      DeviceSensors.ArrayValue.value
10  INTO
11     output
12  FROM
13    Device
14  CROSS APPLY GetElements(Device.sensors) AS DeviceSensors
15  WHERE
16     ((DeviceSensors.ArrayValue.type = 1) AND
        (DeviceSensors.ArrayValue.value > 180))
```

Line 1: Get the next batch of messages from input and create the alias `Device` to refer to an individual message.

Lines 2 through 9: Select the data of interest. Note the use of the "." (dot) dereference to select into the JSON structure.

Lines 10 and 11: Identify the output by alias.

Lines 12 and 13: Specify where the data is coming from, in this case `Device`.

Line 14: The CROSS APPLY function allows you to flatten out an array. The end result is that there will be a unique output message for each element in the array.

Lines 15 and 16: The `where` clause specifies that you are only interested in messages that contain a glucose (type = 1) value that is out of range (value > 180).

Stream Analytics has a feature that allows you to test your queries before putting them into action. This is a very useful feature and should not be overlooked when developing with Stream Analytics. First, let's see how you can test the biometrics-store Stream Analytics query (see Figure 7-15).

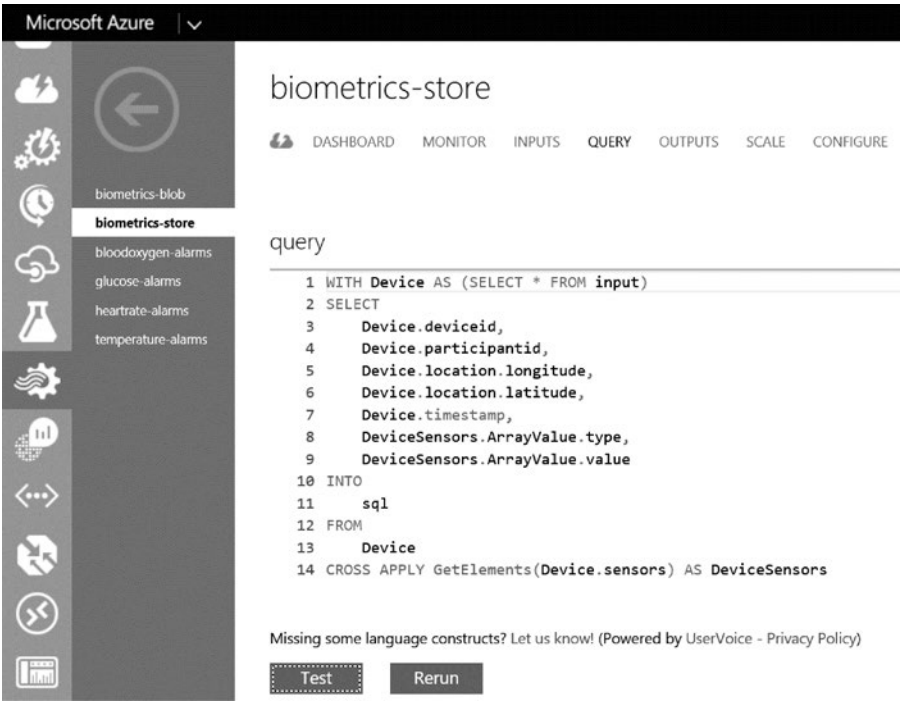

Figure 7-15. *Stream Analytics Query Definition*

When you click the Test button, a dialog pops up and you can browse to a JSON file that may contain one or more sample JSON messages. When you click Ok, the query is run against the input file and the results are displayed on the page. You can also download the results to a spreadsheet for further analysis. As you can see from the output in Figure 7-16, the query processed a single incoming device message and created four output rows.

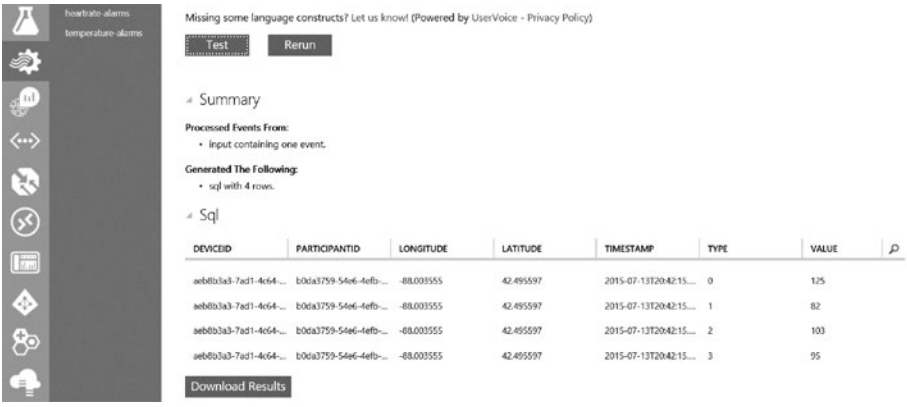

Figure 7-16. *Stream Analytics Test Output*

Now let's see what happens when you run a message through the blood oxygen alarm query of a blood oxygen value that is out of range (see Figure 7-17).

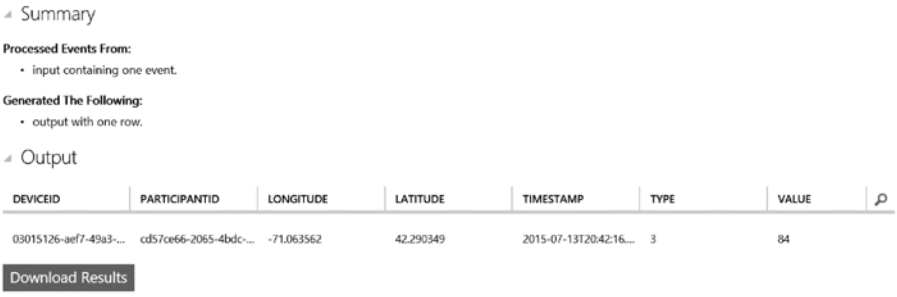

Figure 7-17. *Stream Analytics Output for Alarm Query*

Note that when messages contain out-of-bound values, the new alarm message event with the out-of-bound value is sent to the `alarms` Event Hub for processing. By routing alarm messages to a new Event Hub, you can create a real-time notification process.

Real-Time Notifications

A service that reads from an Event Hub is called a consumer. Stream Analytics, for example, is an Event Hub consumer. It is also possible to create custom Event Hub consumers. As you have seen, Stream Analytics can output to Event Hub, giving you the ability to create a cascading set of Event Hub repositories and Event Hub consumers, which may be useful if you need to run custom business logic on a subset of the incoming messages. Dealing with alarm states is one such scenario.

In the case of alarms, you want to do be able to redirect messages to Notification Hub to provide push notification to mobile devices and log the alarms to SQL Database for reporting purposes. Notification Hub is another service available in Azure Service Bus. Its purpose is to provide push notifications to registered applications. A push notification is a dynamic message that arrives on a device in the form of a badge, toast, or tile message. The applications that can receive push notifications can be running on Windows, Apple, Google, Amazon, or Baidu devices.

A Notification Hub defines a namespace within which one or more push notification hubs can be defined. After you create a notification hub, you can add the necessary certificate and client secret settings for each of the platforms that you want to target (see Figure 7-18).

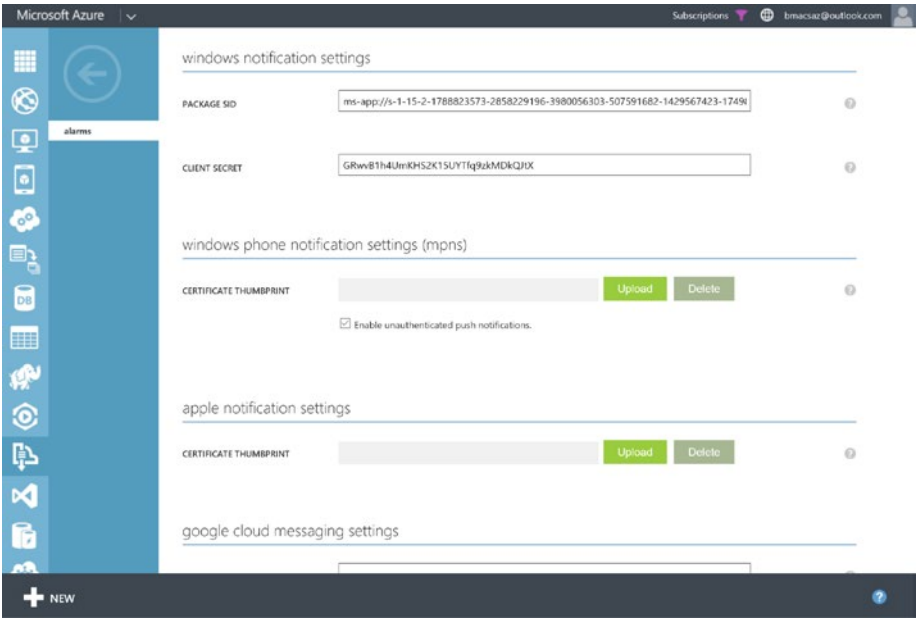

Figure 7-18. *Notification Hub Configuration*

The Biometrics Alarm Notification cloud service connects the dots between the alarm's Event Hub and the alarm's Notification Hub. It will log the alarm to SQL Database using the Biometrics API and send push notifications to a Windows Store application using a push notification hub called alarms. The alarm's Notification Hub is defined within the `alarms-ns` namespace (see Figure 7-19).

Figure 7-19. *Reference Implementation Notification Hub*

Biometrics Alarm Worker

Upon startup, the Biometrics Alarm Worker instantiates an Event Hub Client, the same client that the BioMax Simulator leverages, to connect to the alarms Event Hub. An `EventProcessorHost` is created. This class provides an event-driven model for receiving events from an Event Hub endpoint.

```
// the name of the event hub to receive events from
const string eventHubName = "alarms";

// get the service bus connection string from configuration
var serviceBusConnectionString = RoleEnvironment.
GetConfigurationSettingValue(
        "Azure.ServiceBus.ConnectionString");

// get the storage connection string from configuration
var storageConnectionString = RoleEnvironment.GetConfigurationSettingValue(
        "Azure.Storage.ConnectionString");

// define the transport type as AMQP - advanced message queue protocol
var builder = new ServiceBusConnectionStringBuilder(serviceBusConnection
String);
builder.TransportType = TransportType.Amqp;

// create the event hub client
var eventHubReceiveClient = EventHubClient.CreateFromConnectionString(
        builder.ToString(), eventHubName);

// get the default consumer group
var eventHubConsumerGroup = eventHubReceiveClient.GetDefaultConsumerGroup();
```

```
// create the EventProcessorHost
var eventProcessorHost = new EventProcessorHost( "AlarmsWorker",
    eventHubName,
    eventHubConsumerGroup.GroupName,
    builder.ToString(),
    storageConnectionString);

// register the MessageProcessor class so it recieves the incoming events
eventProcessorHost.RegisterEventProcessorAsync<MessageProcessor>();
```

The `EventProcessorHost` will route incoming events to a class that implements the `IEventProcessor` interface. Your solution defines a class called `MessageProcessor` that implements the `IEventProcessor` interface. This class encapsulates the work that is necessary to prepare a push notification message and send it to the Notification Hub.

The `OpenAsync()` method uses two of your microservice SDKs, ConfigM and ProfileM. ConfigM is used to retrieve the manifests for ProfileM and Biometrics microservices. ProfileM is used to look up the details for the study participant who raised the alarm event and the Biometrics API is used to log the alarm messages to SQL Database. This method also creates the connection to the Notification Hub.

```
_config = new ConfigM
{
    ApiUrl = "<path to the config public api service>"
};

Manifest profileManifest = _config.GetByName("ProfileM");

_profile = new ProfileM
{
    ApiUrl = profileManifest.lineitems["PublicAPI"]
};

var biometricsManifest = _config.GetByName("BiometricsAPI");
_biometricsApi = biometricsManifest.lineitems["PublicAPI"] + "/alarm";

// connect to notification hub
var hub = NotificationHubClient.CreateClientFromConnectionString(
       RoleEnvironment.GetConfigurationSettingValue(
           "Azure.NotificationHub.ConnectionString"),
       RoleEnvironment.GetConfigurationSettingValue(
          "NotificationHubName"));
```

The `ProcessEventsAsync()` method contains the code that will take each incoming alarm event and log it to SQL Database and create a push notification toast message to send to the alarms Notification Hub.

```
// get the alarm message from event hub
var stream = eventData.GetBodyStream();
var bytes = new byte[stream.Length];
stream.Read(bytes, 0, (int) stream.Length);
var json = bytes.Aggregate(string.Empty, (current, t) => current + ((char)
t).ToString());
var alarm = ModelManager.JsonToModel<BiometricReading>(json);

// lookup the user that raised the alarm
var user = _profile.GetById(alarm.participantid);

// log the alarm to biometrics database using the API
Rest.Post(new Uri(_biometricsApi), json);

//format the toast message
var biometric = string.Empty;
switch (alarm.type)
{
    case BiometricType.Glucose:
        biometric = "Glucose";
        break;
    case BiometricType.Heartrate:
        biometric = "Heartrate";
        break;
    case BiometricType.Temperature:
        biometric = "Tempurature";
        break;
    case BiometricType.Bloodoxygen:
        biometric = "Blood Oxygen";
        break;
    default:
        biometric = "Not Set";
        break;
}

// format the toast message
var toast = "<toast><visual><binding template = 'ToastText04'> " +
  $"<text id = '1'>{"Home Biomedical Alert"}</text>" +
  $"<text id = '2'>{"The " + biometric + " reading for " +
  user.firstname + " " + user.lastname + " is out of
  range."}</text>" +
  $"<text id = '3' >{"Contact: " + user.social.phone}</text>" +
  "</binding ></visual></toast>";

// forward the toast to Notification Hub for push
hub.SendWindowsNativeNotificationAsync(toast).Wait();
```

In order to test the Reference Implementation real-time notification mechanism, you will need a mobile application that is associated with the Windows, Apple, or Google stores and is configured to receive notifications. The association is required so that you can retrieve the Package SID and Client Secret necessary to register the application with Notification Hub.

If you have a Windows Store account, you can create an application by reserving a name and then retrieving the Package SID and Client secret. To retrieve these values, reserve an application name, and then under the Services menu on the left, click Push Notifications. On the page, look for the Live Services site link and click through (see Figure 7-20).

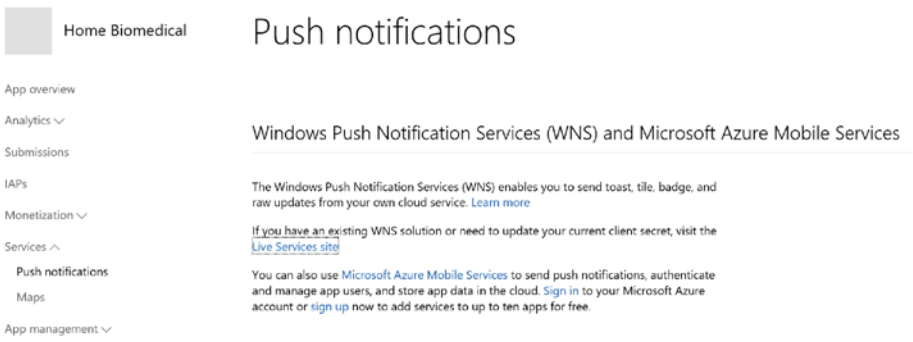

Figure 7-20. *Windows Store Push Notification Instructions*

You will arrive on the page that provides the Package SID and Client Secret. Retrieve these values and enter them on the Notification Hub Configuration page (see Figure 7-21).

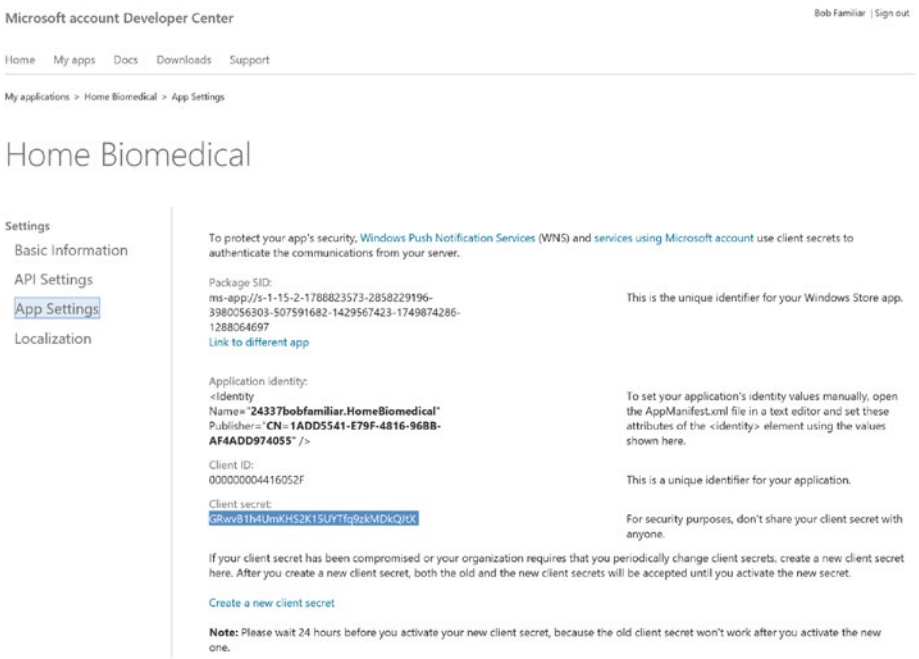

Figure 7-21. *Package SID and Client Secret*

The next step is to associate your Windows Store app with this reserved name in the Store. In Visual Studio, select Project ➤ Store ➤ Associate App with Store. You will be promoted to log into your store account, and you will receive a list of your reserved names. Select the one that you just created and move through the wizard (see Figure 7-22).

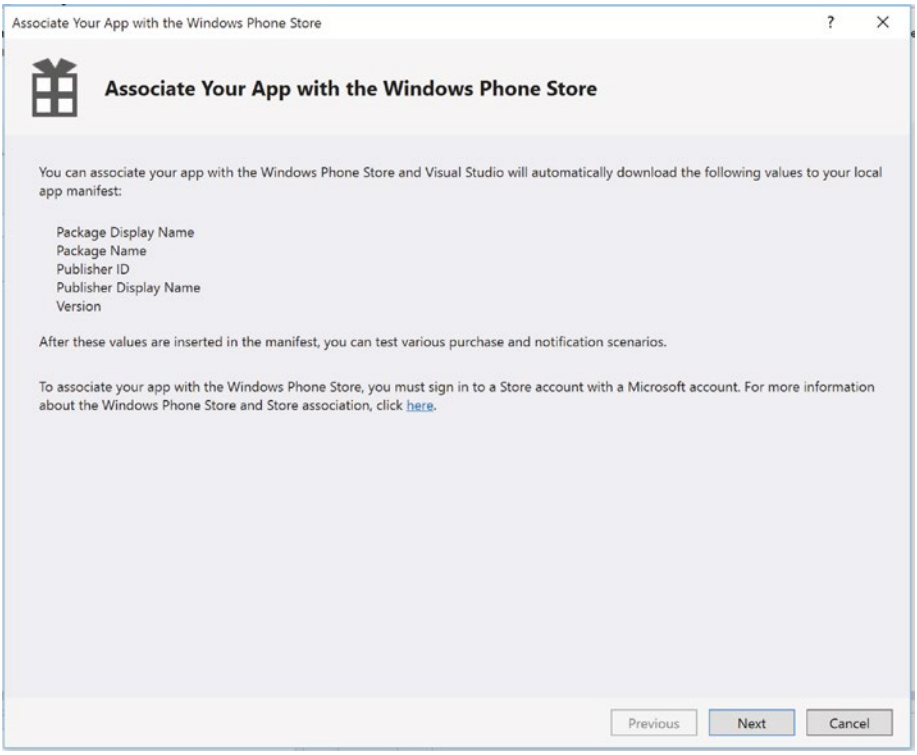

Figure 7-22. *Windows Store Association Wizard*

Open the Package Manifest, and on the Application Tab, set the Toast Capable option to 'Yes' (see Figure 7-23).

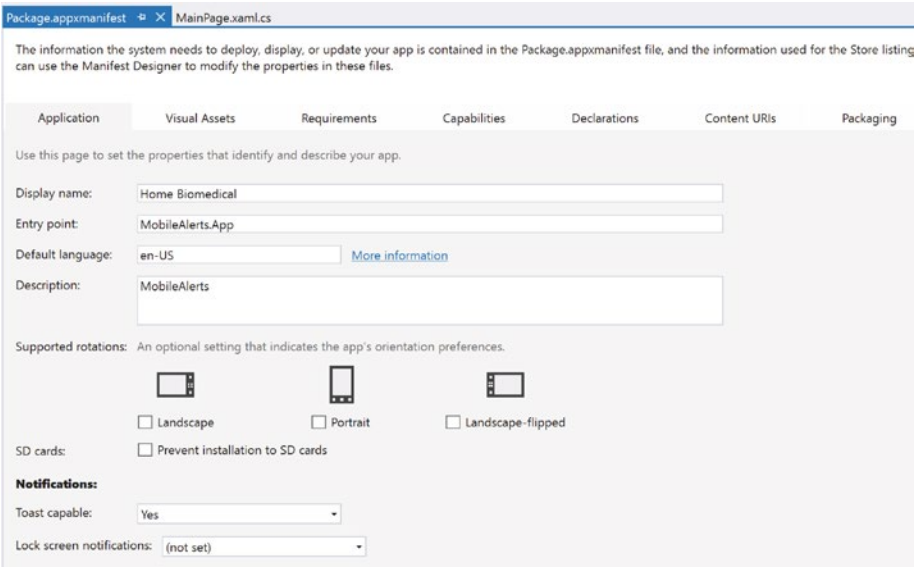

Figure 7-23. *Application Package Manifest*

Using NuGet Package Manager, add the Windows Azure Messaging package to your solution (see Figure 7-24).

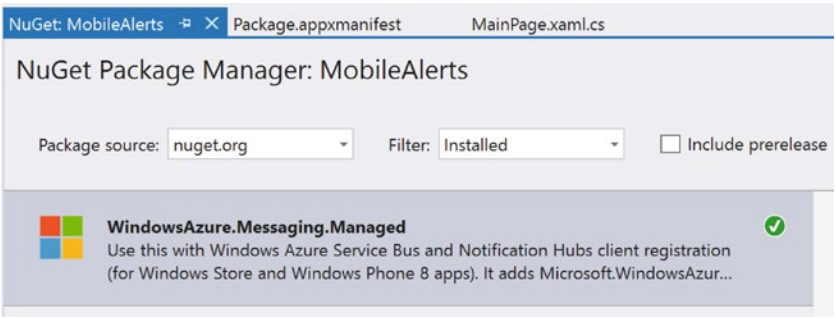

Figure 7-24. *Windows Azure Messaging Package for Notification Hub Clients*

At application startup, create the hub client and the channel on which the push notifications will arrive. This creates a registration between the client application and the alarms push notification endpoint.

```
hub = new NotificationHub("alarms", "<notificaiton hub connection tring");

var channel = await PushNotificationChannelManager.
        CreatePushNotificationChannelForApplicationAsync();

await hub.RegisterNativeAsync(channel.Uri);
```

Testing Push Notifications

To test your mobile application, start the BioMax Simulator and then start your mobile application. You can optionally run the Biometrics Alarm Worker solution locally if you want to set breakpoints in that project. As alarms are picked up by the Stream Analytics jobs, they will be routed to the alarms Event Hub. There they will be picked up the Biometrics Alarm Worker who formats push notifications and sends them to the alarms Notification Hub. The Notification Hub will then push the notifications to any app that has an open channel on that hub. Figure 7-25 shows both the dashboard showing all biometric data being tracked in real time and the mobile app showing an alert toast.

Figure 7-25. *Real-Time Dashboard and Mobile Alerts*

Real-Time Data Visualization

The biometrics-store Stream Analytics job routes device readings to SQL Database. Since the data is a bit cryptic, it makes sense to wrap the data with an API that provides context and, if necessary, business logic so that the data is provided in a meaningful way to the application.

There are many libraries, controls, and products that can be used to create data visualizations in responsive web applications. The Reference Implementation includes a sample application that uses AngularJS, Bootstrap, and D3 to create a wallboard-style dashboard that displays the device locations on maps of New York, Boston, and Chicago. It aggregates sensor reading data on gauges and provides examples of data aggregation (see Figure 7-19).

Biometrics API

The Biometrics API provides a contextual API for accessing the device readings stored in SQL Database. When used in conjunction with ASP.NET SignalR, the API can be used to provide real-time updates to client applications. SignalR allows bi-directional communication between server and client. Servers can push content to connected clients the instant it becomes available. SignalR supports Web Sockets, and falls back to other compatible techniques for older browsers.

■ **Note** For more information on SignalR, including documentation and sample code, visit the official SignalR web site at `www.asp.net/signalr`.

Each row of data in the database contains a device id, participant id, the longitude and latitude coordinates for the location of the device, a time stamp, a sensor id, and a value. Since the data is flowing in real time, the API will return a specified number of rows of the most recent data. There are three endpoints:

```
// return the last N-number of readings by device id
biometrics/device/{deviceid}{/count/{count}

// return the last N-number of readings by participant id
biometrics/participant/{participantid}/count/{count}

// return the last N-number of readings by city and sensor
// type where sensor type is glucose, heartrate, temperature
// or bloodoxygen
biometrics/city/{city}/type/{type}/count/{count}
```

The Home Biomedical Reference implementation has pre-defined a set of 300 participants who are located in Boston, New York, and Chicago. These city names can be used as arguments to the Biometrics API along with the name of the sensor type and a count of records. For example, a possible invocation of the Biometrics API would be

```
http://biometricsapi.azurewebsites.net/biometrics/city/boston/type/glucose/
count/10
```

The data returned would be formatted as depicted in Figure 7-26.

```
▼<BiometricReading>
    <deviceid>f7e1f45d-9e4b-4d2d-ab3f-035928bbdc37</deviceid>
    <latitude>42.29652</latitude>
    <longitude>-71.138602</longitude>
    <participantid>59df0dae-8ff9-4652-9d02-f236f3b5bdcd</participantid>
    <reading>2015-07-15T20:57:07.68</reading>
    <type>Glucose</type>
    <value>184</value>
  </BiometricReading>
▼<BiometricReading>
    <deviceid>4b65017a-74fc-4bc2-8feb-65d9feb73c2c</deviceid>
    <latitude>42.258357</latitude>
    <longitude>-71.255878</longitude>
    <participantid>02a3d29f-c009-4176-8cce-03337abeb02d</participantid>
    <reading>2015-07-15T20:57:07.077</reading>
    <type>Glucose</type>
    <value>161</value>
  </BiometricReading>
▼<BiometricReading>
    <deviceid>3769ade1-06ce-41a0-808f-a1fe7c963f46</deviceid>
    <latitude>42.269741</latitude>
    <longitude>-71.09219</longitude>
```


Figure 7-26. *Biometrics API JSON*

■ **Note** The Biometrics-related solutions can be found in Microservices\Biometrics.

Summary

IoT is not new. Devices connected on a network delivering real-time telemetry have been around for a long time. Think about the connectivity and telemetry acquisition that NASA put in place for the first trip to the moon in 1969. Mission control was monitoring every aspect of the hardware, the capsule, and landing module, as well as the biometrics of the astronauts through their suits.

What has changed in the past couple of years is the commoditization and proliferation of sensors and devices and the commoditization of the services necessary to connect to these devices and ingest the sensor data at volume. Azure is at the forefront of this movement, providing an IoT microservices stack that allows you to bring these types of solutions to market in days and weeks rather than months and years. Azure Event Hub, Stream Analytics, and Notification Hub provide the necessary foundational microservices that, when combined with your custom Microservices, deliver a highly scalable, fault tolerant, reliable Software as a Service IoT solution.