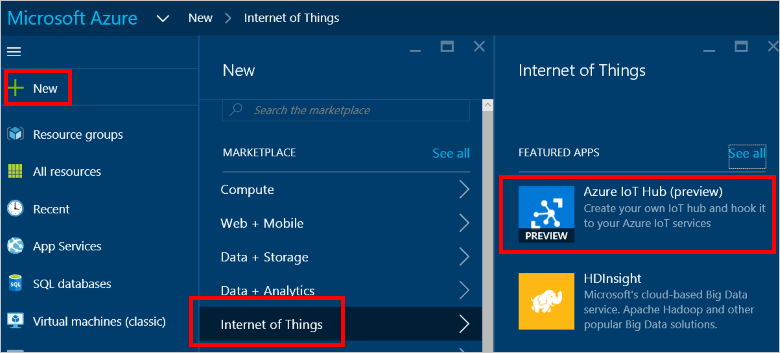
Microsoft Azure IoT Hub   
Hands on Lab

v1.2: 6 Sept, 2016

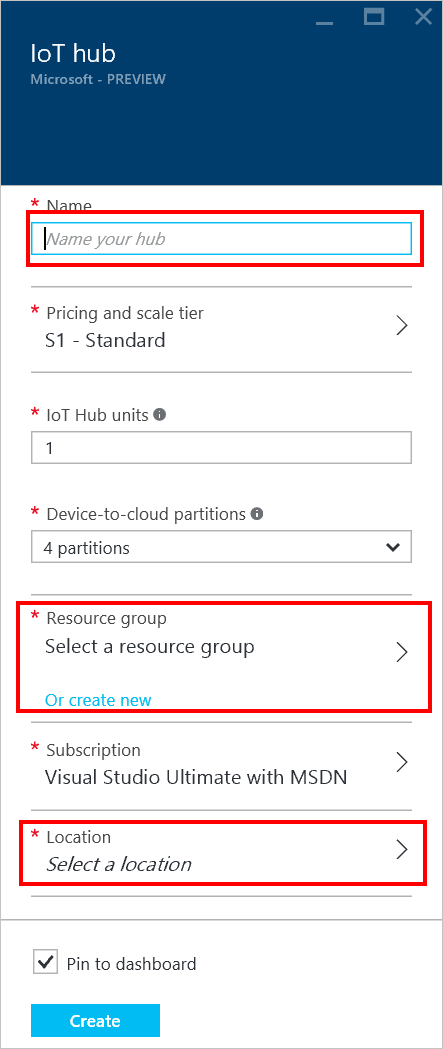
Create an IoT hub

You need to create an IoT Hub for you simulated device to connect to. The following steps show you how to complete this task using the Azure portal.

1. Sign in to the [Azure portal](https://portal.azure.com/).
2. In the Jumpbar, click **New**, then click **Internet of Things**, and then click **Azure IoT Hub**.

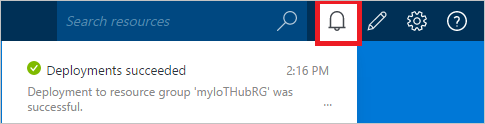


1. In the **IoT hub** blade, choose the configuration for your IoT hub.

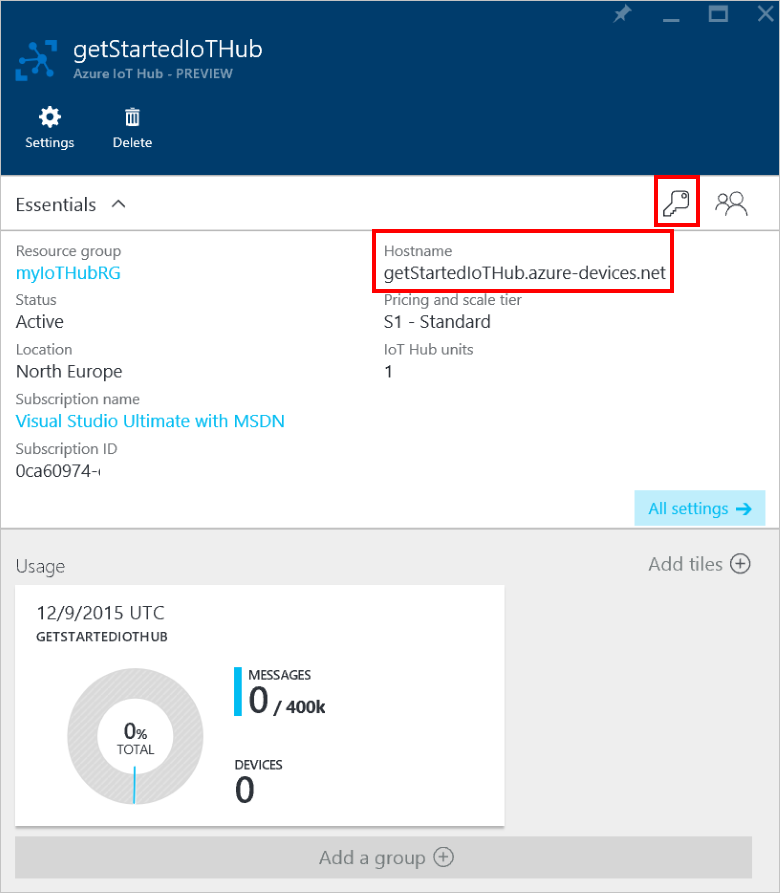


* + In the **Name** box, enter a name for your IoT hub. If the **Name** is valid and available, a green check mark appears in the **Name** box.
  + Select a **Pricing and scale tier**. This tutorial does not require a specific tier.
  + In **Resource group**, create a new resource group, or select an existing one. For more information, see [Using resource groups to manage your Azure resources](https://azure.microsoft.com/en-us/documentation/articles/resource-group-portal/).
  + In **Location**, select the location to host your IoT hub.

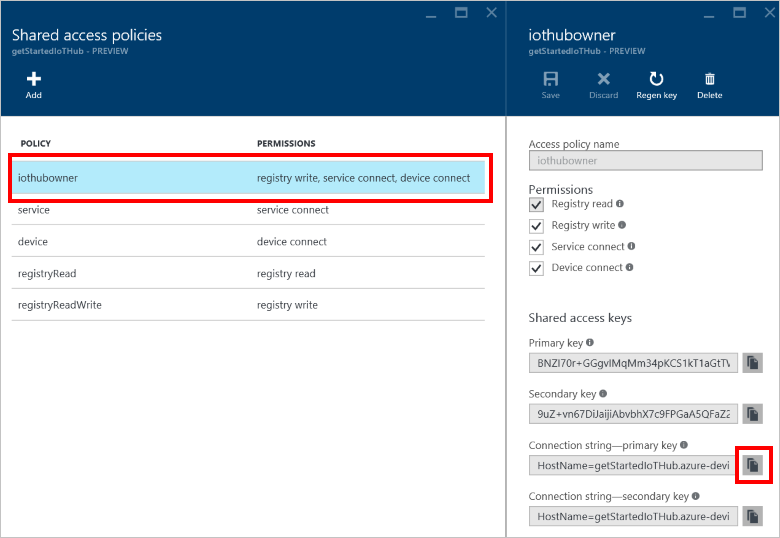
1. When you have chosen your IoT hub configuration options, click **Create**. It can take a few minutes for Azure to create your IoT hub. To check the status, you can monitor the progress on the Startboard or in the Notifications panel.



1. When the IoT hub has been created successfully, open the blade of the new IoT hub, make a note of the **Hostname**, and then click the**Keys** icon.



1. Click the **iothubowner** policy, then copy and make note of the connection string in the **iothubowner** blade.

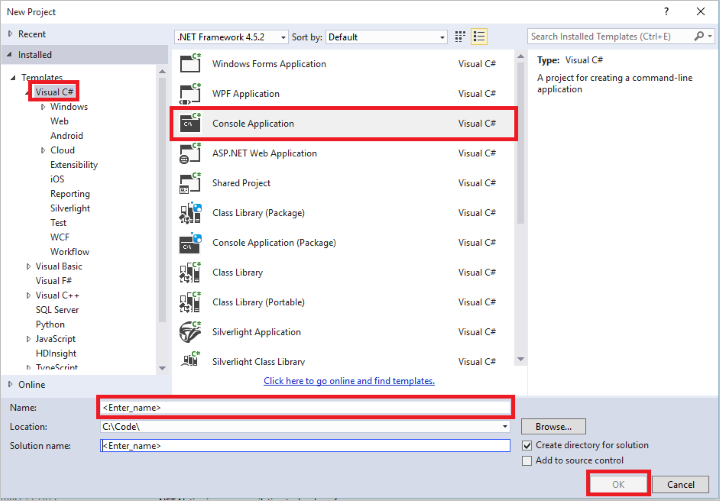


You have now created your IoT hub and you have the hostname and connection string you need to complete the rest of this tutorial.

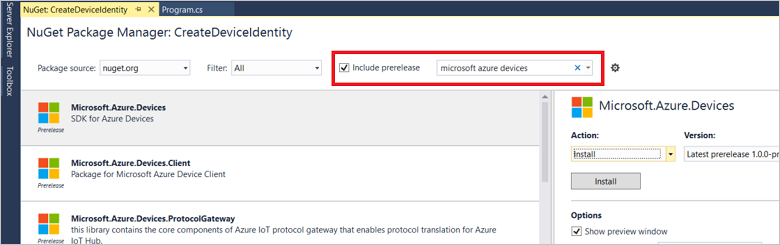
Create a device identity

In this section, you'll create a Windows console app that creates a new device identity in the identity registry in your IoT hub. A device cannot connect to IoT hub unless it has an entry in the device identity registry. Refer to the **Device Identity Registry** section of the [IoT Hub Developer Guide](https://azure.microsoft.com/en-us/documentation/articles/iot-hub-devguide/#identityregistry) for more information. When you run this console application, it generates a unique device ID and key that your device can identify itself with when it sends device-to-cloud messages to IoT Hub.

1. In Visual Studio, add a new Visual C# Windows Classic Desktop project to the current solution using the **Console Application** project template. Name the project **CreateDeviceIdentity**.



1. In Solution Explorer, right-click the **CreateDeviceIdentity** project, and then click **Manage NuGet Packages**.
2. In the **NuGet Package Manager** window, make sure the **Include prerelease** option is checked. Then search for **Microsoft Azure Devices**, click **Install** to install the **Microsoft.Azure.Devices** package, and accept the terms of use.



1. This downloads, installs, and adds a reference to the [Microsoft Azure IoT Service SDK](https://www.nuget.org/packages/Microsoft.Azure.Devices/) NuGet package.
2. Add the following using statements at the top of the **Program.cs** file:

Copy

using Microsoft.Azure.Devices;

using Microsoft.Azure.Devices.Common.Exceptions;

1. Add the following fields to the **Program** class, replacing the placeholder value with the connection string for the IoT hub you created in the previous section:

Copy

static RegistryManager registryManager;

static string connectionString = "{iothub connection string}";

1. Add the following method to the **Program** class:

Copy

private async static Task AddDeviceAsync()

{

string deviceId = "myFirstDevice";

Device device;

try

{

device = await registryManager.AddDeviceAsync(new Device(deviceId));

}

catch (DeviceAlreadyExistsException)

{

device = await registryManager.GetDeviceAsync(deviceId);

}

Console.WriteLine("Generated device key: {0}", device.Authentication.SymmetricKey.PrimaryKey);

}

This method creates a new device identity with ID **myFirstDevice** (if that device ID already exists in the registry, the code simply retrieves the existing device information). The app then displays the primary key for that identity. You will use this key in the simulated device to connect to your IoT hub.

1. Finally, add the following lines to the **Main** method:

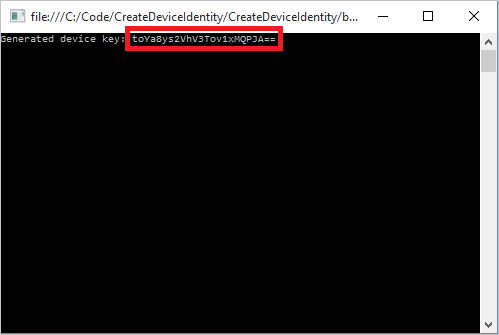
Copy to clipboardCopy

registryManager = RegistryManager.CreateFromConnectionString(connectionString);

AddDeviceAsync().Wait();

Console.ReadLine();

1. Run this application, and make a note of the device key.



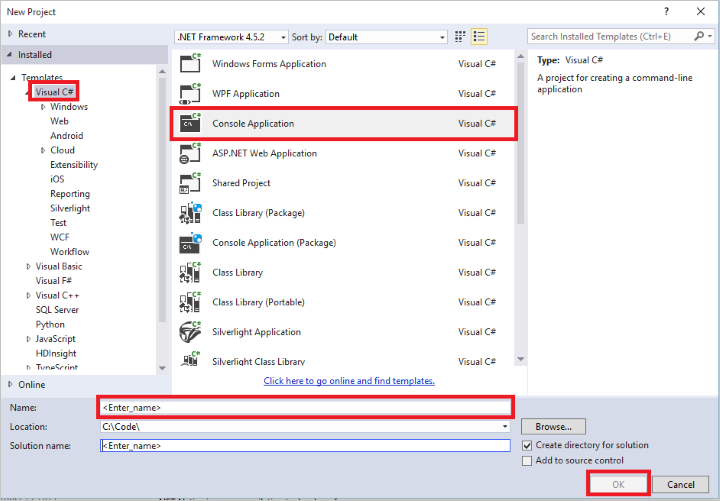
NOTE:

The IoT Hub identity registry only stores device identities to enable secure access to the hub. It stores device IDs and keys to use as security credentials and an enabled/disabled flag that enables you to disable access for an individual device. If you application needs to store other device-specific metadata, it should use an application-specific store. Refer to [IoT Hub Developer Guide](https://azure.microsoft.com/en-us/documentation/articles/iot-hub-devguide/#identityregistry) for more information.

Create a simulated device app

In this section, you'll create a Windows console app that simulates a device that sends device-to-cloud messages to an IoT hub.

1. In Visual Studio, add a new Visual C# Windows Classic Desktop project to the current solution using the **Console Application** project template. Name the project **SimulatedDevice**.



1. In Solution Explorer, right-click the **SimulatedDevice** project, and then click **Manage NuGet Packages**.
2. In the **NuGet Package Manager** window, make sure the **Include prerelease** option is checked. Search for **Microsoft Azure Devices Client**, click **Install**, and accept the terms of use.

This downloads, installs, and adds a reference to the [Azure IoT - Device SDK NuGet package](https://www.nuget.org/packages/Microsoft.Azure.Devices.Client/).

1. Add the following using statement at the top of the **Program.cs** file:

Copy

using Microsoft.Azure.Devices.Client;

using Newtonsoft.Json;

using System.Threading;

1. Add the following fields to the **Program** class, substituting the placeholder values with the IoT hub hostname you retrieved in the *Create an IoT hub* section and the device key retrieved in the *Create a device identity* section:

Copy

static DeviceClient deviceClient;

static string iotHubUri = "{iot hub hostname}";

static string deviceKey = "{device key}";

1. Add the following method to the **Program** class:

Copy

private static async void SendDeviceToCloudMessagesAsync()

{

double avgWindSpeed = 10; // m/s

Random rand = new Random();

while (true)

{

double currentWindSpeed = avgWindSpeed + rand.NextDouble() \* 4 - 2;

var telemetryDataPoint = new

{

deviceId = "myFirstDevice",

windSpeed = currentWindSpeed

};

var messageString = JsonConvert.SerializeObject(telemetryDataPoint);

var message = new Message(Encoding.ASCII.GetBytes(messageString));

await deviceClient.SendEventAsync(message);

Console.WriteLine("{0} > Sending message: {1}", DateTime.Now, messageString);

Thread.Sleep(1000);

}

}

This method sends a new device-to-cloud message every second. The message contains a JSON-serialized object with the deviceId and a randomly generated number to simulate a wind speed sensor.

1. Finally, add the following lines to the **Main** method:

Copy

Console.WriteLine("Simulated device\n");

deviceClient = DeviceClient.Create(iotHubUri, new DeviceAuthenticationWithRegistrySymmetricKey("myFirstDevice", deviceKey));

SendDeviceToCloudMessagesAsync();

Console.ReadLine();

By default, the **Create** method creates a **DeviceClient** that uses the AMQP protocol to communicate with IoT Hub. To use the HTTPS protocol, use the override of the **Create** method that enables you to specify the protocol. If you choose to use the HTTPS protocol, you should also add the **Microsoft.AspNet.WebApi.Client** NuGet package to your project to include the **System.Net.Http.Formatting** namespace.

NOTE:

To keep things simple, this tutorial does not implement any retry policy. In production code, you should implement retry policies (such as an exponential backoff), as suggested in the MSDN article [Transient Fault Handling](https://msdn.microsoft.com/en-us/library/hh680901(v=pandp.50).aspx).

In this section, you'll modify the simulated device application you created in the [Get started with IoT Hub](https://azure.microsoft.com/en-us/documentation/articles/iot-hub-csharp-csharp-getstarted/) tutorial to send interactive device-to-cloud messages to the IoT hub.

1. In Visual Studio, in the **SimulatedDevice** project, add the following method to the **Program** class.

Copy

private static async void SendDeviceToCloudInteractiveMessagesAsync()

{

while (true)

{

var interactiveMessageString = "Alert message!";

var interactiveMessage = new Message(Encoding.ASCII.GetBytes(interactiveMessageString));

interactiveMessage.Properties["messageType"] = "interactive";

interactiveMessage.MessageId = Guid.NewGuid().ToString();

await deviceClient.SendEventAsync(interactiveMessage);

Console.WriteLine("{0} > Sending interactive message: {1}", DateTime.Now, interactiveMessageString);

Thread.Sleep(10000);

}

}

This method is very similar to the **SendDeviceToCloudMessagesAsync** method in **SimulatedDevice** project. The only differences are that you now set the **MessageId** system property, and a user property called **messageType**. The code assigns a globally unique identifier (guid) to the **MessageId** property, that Service Bus can use to deduplicate the messages it receives. The sample uses the **messageType**property to distinguish interactive from data point messages. The application passes this information in message properties instead of in the message body, so that the event processor does not need to deserialize the message to perform message routing.

##### NOTE:

It is important to create the **MessageId** used to deduplicate interactive messages in the device code because intermittent network communications, or other failures, could result in multiple retransmissions of the same message from that device. You can also use a semantic message id - such as a hash of the relevant message data fields - in place of a guid.

1. Add the following method in the **Main** method right before the Console.ReadLine() line:

Copy to clipboardCopy

SendDeviceToCloudInteractiveMessagesAsync();

In this section, you'll modify the simulated device application you created in the [Get started with IoT Hub](https://azure.microsoft.com/en-us/documentation/articles/iot-hub-csharp-csharp-getstarted/) tutorial to send interactive device-to-cloud messages to the IoT hub.

1. In Visual Studio, in the **SimulatedDevice** project, add the following method to the **Program** class.

Copy

private static async void SendDeviceToCloudInteractiveMessagesAsync()

{

while (true)

{

var interactiveMessageString = "Alert message!";

var interactiveMessage = new Message(Encoding.ASCII.GetBytes(interactiveMessageString));

interactiveMessage.Properties["messageType"] = "interactive";

interactiveMessage.MessageId = Guid.NewGuid().ToString();

await deviceClient.SendEventAsync(interactiveMessage);

Console.WriteLine("{0} > Sending interactive message: {1}", DateTime.Now, interactiveMessageString);

Thread.Sleep(10000);

}

}

This method is very similar to the **SendDeviceToCloudMessagesAsync** method in **SimulatedDevice** project. The only differences are that you now set the **MessageId** system property, and a user property called **messageType**. The code assigns a globally unique identifier (guid) to the **MessageId** property, that Service Bus can use to deduplicate the messages it receives. The sample uses the **messageType**property to distinguish interactive from data point messages. The application passes this information in message properties instead of in the message body, so that the event processor does not need to deserialize the message to perform message routing.

##### NOTE:

It is important to create the **MessageId** used to deduplicate interactive messages in the device code because intermittent network communications, or other failures, could result in multiple retransmissions of the same message from that device. You can also use a semantic message id - such as a hash of the relevant message data fields - in place of a guid.

1. Add the following method in the **Main** method right before the Console.ReadLine() line:

Copy to clipboardCopy

SendDeviceToCloudInteractiveMessagesAsync();

## Processing device-to-cloud messages

In this section, you will create a Windows console app that processes device-to-cloud messages from IoT Hub. Iot Hub exposes an [Event Hubs](https://azure.microsoft.com/en-us/documentation/articles/event-hubs-overview/)-compatible endpoint to enable an application to read device-to-cloud messages. This tutorial uses the [EventProcessorHost](http://msdn.microsoft.com/library/azure/microsoft.servicebus.messaging.eventprocessorhost(v=azure.95).aspx) class to process these messages in a console app. For more information on how to process messages from Event Hubs you can refer to the [Get Started with Event Hubs](https://azure.microsoft.com/en-us/documentation/articles/event-hubs-csharp-ephcs-getstarted/) tutorial.

The main challenge you face when you implement reliable storage of data point messages or forwarding of interactive messages, is that Event Hubs event processing relies on the message consumer to checkpoint its progress. Moreover, in order to achieve a high throughput, when you read from Event Hubs you should checkpoint in large batches. This creates the possibility of duplicate processing for a large number of messages if there is a failure and you revert to the previous checkpoint. In this tutorial you will see how to synchronize Azure storage writes and Service Bus deduplication windows with **EventProcessorHost** checkpoints.

To reliably write messages to Azure storage, the sample uses the individual block commit feature of [block blobs](https://msdn.microsoft.com/library/azure/ee691964.aspx). The event processor accumulates messages in memory until it is time to perform a checkpoint, such as after the accummulated buffer of messages reaches the maximum block size of 4Mb, or after the Service Bus deduplication time window elapses. Then, before checkpointing, the code commits a new block to the blob.

The event processor uses Event Hubs message offsets as block ids. This allows it to perform a deduplication check before it commits the new block to storage, taking care of a possible crash between committing a block and the checkpoint.

##### NOTE:

This tutorial uses a single storage account to write all the messages retrieved from IoT Hub. Refer to [Azure Storage scalability Guidelines](https://azure.microsoft.com/en-us/documentation/articles/storage-scalability-targets/) to decide if you need to use multiple Azure Storage accounts in your solution.

The application makes use of the Service Bus deduplication feature to avoid duplicates when it processes interactive messages. The simulated device stamps each interactive message with a unique **MessageId** to enable Service Bus to ensure that, in the specified deduplication time window, no two messages with the same **MessageId** are delivered to the receivers. This deduplication, together with the per-message completion semantics provided by Service Bus queues, makes it easy to implement the reliable processing of interactive messages.

To make sure that no message is resubmitted outside of the deduplication window, the code synchronizes the **EventProcessorHost**checkpointing mechanism with the Service Bus queue deduplication window. This is done by forcing a checkpoint at least once every time the deduplication window elapses (one hour in this tutorial).

##### NOTE:

This tutorial uses a single partitioned Service Bus queue to process all the interactive messages retrieved from IoT Hub. Refer to the [Service Bus documentation](https://azure.microsoft.com/documentation/services/service-bus/) for more information on how to use Service Bus Queues to meet the scalability requirements of your solution.

### **Provision an Azure Storage account and a Service Bus queue**

In order to use the [EventProcessorHost](http://msdn.microsoft.com/library/azure/microsoft.servicebus.messaging.eventprocessorhost(v=azure.95).aspx) class, you must have an Azure Storage account to enable the **EventProcessorHost** to record checkpoint information. You can use an existing storage account, or follow the instructions in [About Azure Storage](https://azure.microsoft.com/en-us/documentation/articles/storage-create-storage-account/#create-a-storage-account) to create a new one. Make a note of the storage account connection string.

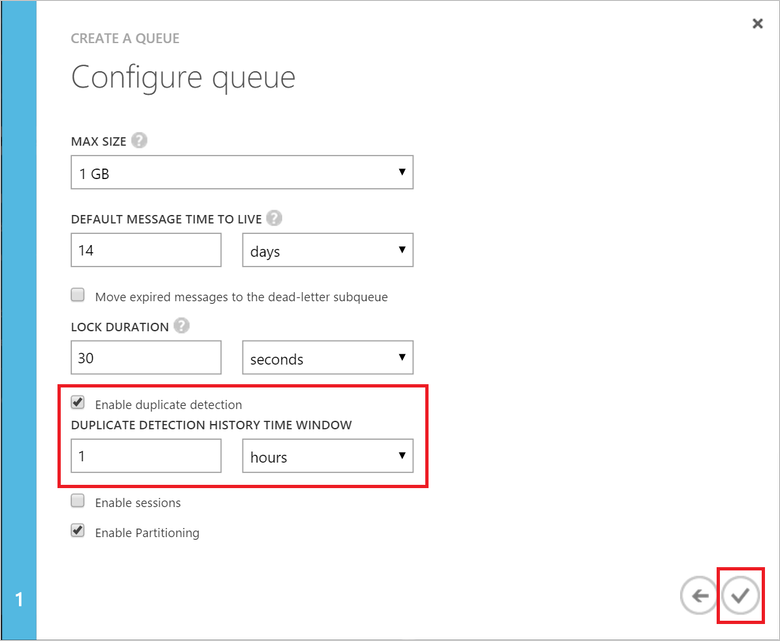
You also need a Service Bus queue to enable reliable processing of interactive messages. You can create a queue programmatically with a 1 hour deduplication window, as explained in [How to use Service Bus Queues](https://azure.microsoft.com/en-us/documentation/articles/service-bus-dotnet-how-to-use-queues/), or use the [Azure portal](https://manage.windowsazure.com/), following these steps:

### **Provision an Azure Storage account and a Service Bus queue**

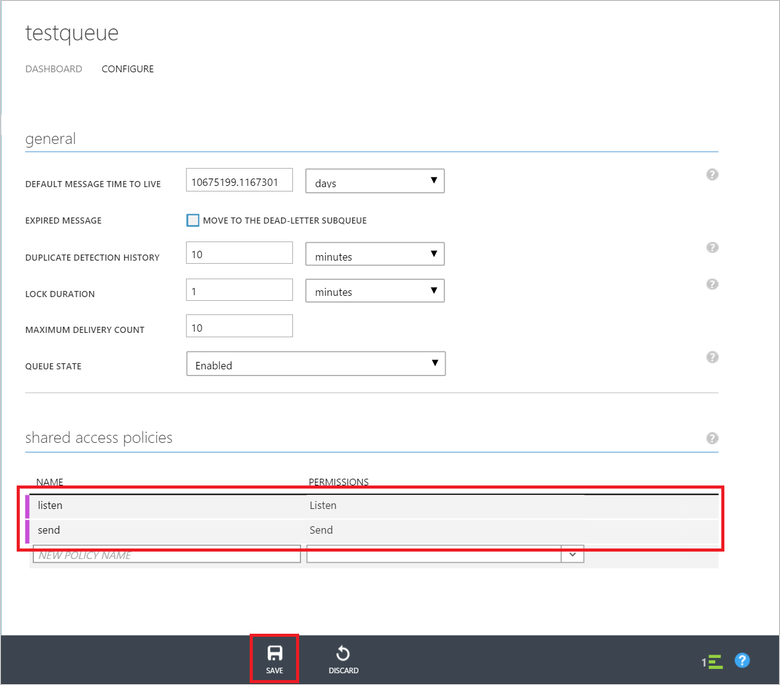
In order to use the [EventProcessorHost](http://msdn.microsoft.com/library/azure/microsoft.servicebus.messaging.eventprocessorhost(v=azure.95).aspx) class, you must have an Azure Storage account to enable the **EventProcessorHost** to record checkpoint information. You can use an existing storage account, or follow the instructions in [About Azure Storage](https://azure.microsoft.com/en-us/documentation/articles/storage-create-storage-account/#create-a-storage-account) to create a new one. Make a note of the storage account connection string.

You also need a Service Bus queue to enable reliable processing of interactive messages. You can create a queue programmatically with a 1 hour deduplication window, as explained in [How to use Service Bus Queues](https://azure.microsoft.com/en-us/documentation/articles/service-bus-dotnet-how-to-use-queues/), or use the [Azure portal](https://manage.windowsazure.com/), following these steps:

1. Click **NEW** in the bottom left corner, then **App Services**, then **Service Bus**, then **Queue**, then **Custom Create**, enter the name **d2ctutorial**, select a region, use an existing namespace or create a new one, then on the next page select **Enable duplicate detection** and set the **Duplicate detection history time window** to one hour. Then click the check mark to save your queue configuration.

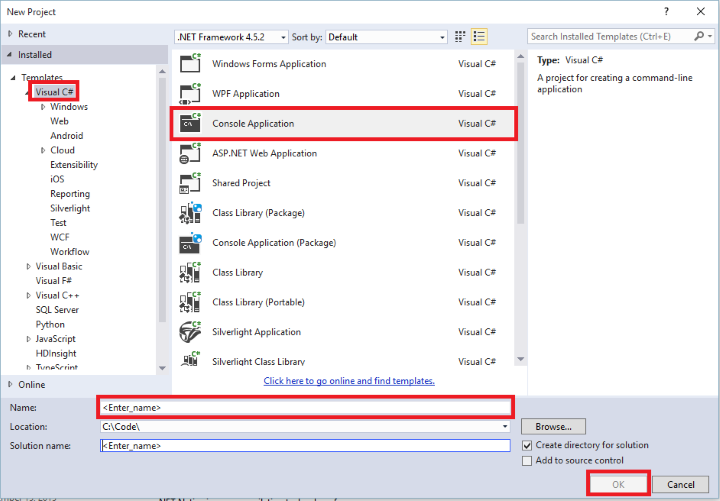
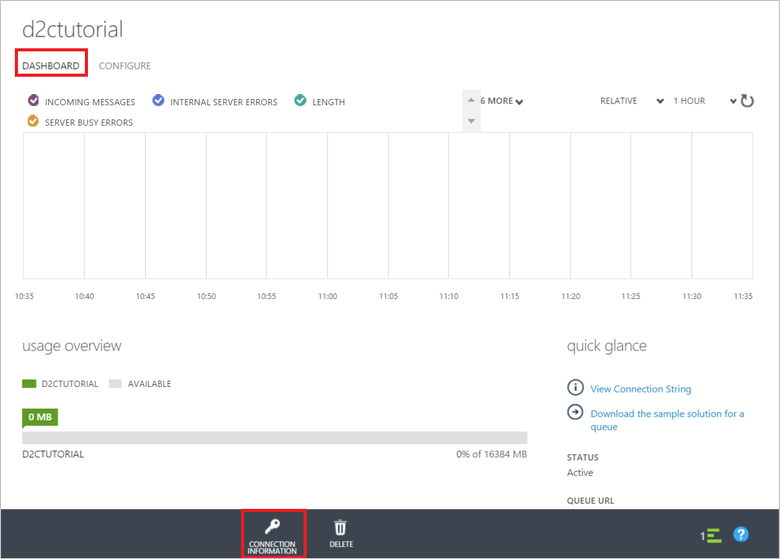


1. In the list of Service Bus queues, click **d2ctutorial**, and then click **Configure**. Create two shared access policies, one called **send** with **Send**permissions, and one called **listen** with **Listen** permissions. Click **Save** at the bottom, when done.



1. Click **Dashboard** at the top, then **Connection information** at the bottom, make a note of the two connection strings.

**Create the event processor**

1. In the current Visual Studio solution, click **File**, then **Add**, and then **New Project** to create a new Visual C# Windows project using the**Console Application** project template. Name the project **ProcessDeviceToCloudMessages**.
2. In Solution Explorer, right-click the **ProcessDeviceToCloudMessages** project, and then click **Manage NuGet Packages**. The **NuGet Package Manager** dialog box appears.
3. Search for **WindowsAzure.ServiceBus**, click **Install**, and accept the terms of use. This downloads, installs, and adds a reference to the[Azure Service Bus NuGet package](https://www.nuget.org/packages/WindowsAzure.ServiceBus), with all its dependencies.
4. Search for **Microsoft Azure Service Bus Event Hub - EventProcessorHost**, click **Install**, and accept the terms of use. This downloads, installs, and adds a reference to the [Azure Service Bus Event Hub - EventProcessorHost NuGet package](https://www.nuget.org/packages/Microsoft.Azure.ServiceBus.EventProcessorHost), with all its dependencies.
5. Right-click the **ProcessDeviceToCloudMessages** project, click **Add**, and then click **Class**. Name the new class **StoreEventProcessor**, and then click **OK** to create the class.
6. Add the following statements at the top of the StoreEventProcessor.cs file:

Copy

using System.IO;

using System.Diagnostics;

using System.Security.Cryptography;

using Microsoft.ServiceBus.Messaging;

using Microsoft.WindowsAzure.Storage;

using Microsoft.WindowsAzure.Storage.Blob;

1. Substitute the following code for the body of the class:

Copy

class StoreEventProcessor : IEventProcessor

{

private const int MAX\_BLOCK\_SIZE = 4 \* 1024 \* 1024;

public static string StorageConnectionString;

public static string ServiceBusConnectionString;

private CloudBlobClient blobClient;

private CloudBlobContainer blobContainer;

private QueueClient queueClient;

private long currentBlockInitOffset;

private MemoryStream toAppend = new MemoryStream(MAX\_BLOCK\_SIZE);

private Stopwatch stopwatch;

private TimeSpan MAX\_CHECKPOINT\_TIME = TimeSpan.FromHours(1);

public StoreEventProcessor()

{

var storageAccount = CloudStorageAccount.Parse(StorageConnectionString);

blobClient = storageAccount.CreateCloudBlobClient();

blobContainer = blobClient.GetContainerReference("d2ctutorial");

blobContainer.CreateIfNotExists();

queueClient = QueueClient.CreateFromConnectionString(ServiceBusConnectionString, "d2ctutorial");

}

Task IEventProcessor.CloseAsync(PartitionContext context, CloseReason reason)

{

Console.WriteLine("Processor Shutting Down. Partition '{0}', Reason: '{1}'.", context.Lease.PartitionId, reason);

return Task.FromResult<object>(null);

}

Task IEventProcessor.OpenAsync(PartitionContext context)

{

Console.WriteLine("StoreEventProcessor initialized. Partition: '{0}', Offset: '{1}'", context.Lease.PartitionId, context.Lease.Offset);

if (!long.TryParse(context.Lease.Offset, out currentBlockInitOffset))

{

currentBlockInitOffset = 0;

}

stopwatch = new Stopwatch();

stopwatch.Start();

return Task.FromResult<object>(null);

}

async Task IEventProcessor.ProcessEventsAsync(PartitionContext context, IEnumerable<EventData> messages)

{

foreach (EventData eventData in messages)

{

byte[] data = eventData.GetBytes();

if (eventData.Properties.ContainsKey("messageType") && (string) eventData.Properties["messageType"] == "interactive")

{

var messageId = (string) eventData.SystemProperties["message-id"];

var queueMessage = new BrokeredMessage(new MemoryStream(data));

queueMessage.MessageId = messageId;

queueMessage.Properties["messageType"] = "interactive";

await queueClient.SendAsync(queueMessage);

WriteHighlightedMessage(string.Format("Received interactive message: {0}", messageId));

continue;

}

if (toAppend.Length + data.Length > MAX\_BLOCK\_SIZE || stopwatch.Elapsed > MAX\_CHECKPOINT\_TIME)

{

await AppendAndCheckpoint(context);

}

await toAppend.WriteAsync(data, 0, data.Length);

Console.WriteLine(string.Format("Message received. Partition: '{0}', Data: '{1}'",

context.Lease.PartitionId, Encoding.UTF8.GetString(data)));

}

}

private async Task AppendAndCheckpoint(PartitionContext context)

{

var blockIdString = String.Format("startSeq:{0}", currentBlockInitOffset.ToString("0000000000000000000000000"));

var blockId = Convert.ToBase64String(ASCIIEncoding.ASCII.GetBytes(blockIdString));

toAppend.Seek(0, SeekOrigin.Begin);

byte[] md5 = MD5.Create().ComputeHash(toAppend);

toAppend.Seek(0, SeekOrigin.Begin);

var blobName = String.Format("iothubd2c\_{0}", context.Lease.PartitionId);

var currentBlob = blobContainer.GetBlockBlobReference(blobName);

if (await currentBlob.ExistsAsync())

{

await currentBlob.PutBlockAsync(blockId, toAppend, Convert.ToBase64String(md5));

var blockList = await currentBlob.DownloadBlockListAsync();

var newBlockList = new List<string>(blockList.Select(b => b.Name));

if (newBlockList.Count() > 0 && newBlockList.Last() != blockId)

{

newBlockList.Add(blockId);

WriteHighlightedMessage(String.Format("Appending block id: {0} to blob: {1}", blockIdString, currentBlob.Name));

}

else

{

WriteHighlightedMessage(String.Format("Overwriting block id: {0}", blockIdString));

}

await currentBlob.PutBlockListAsync(newBlockList);

}

else

{

await currentBlob.PutBlockAsync(blockId, toAppend, Convert.ToBase64String(md5));

var newBlockList = new List<string>();

newBlockList.Add(blockId);

await currentBlob.PutBlockListAsync(newBlockList);

WriteHighlightedMessage(String.Format("Created new blob", currentBlob.Name));

}

toAppend.Dispose();

toAppend = new MemoryStream(MAX\_BLOCK\_SIZE);

// checkpoint.

await context.CheckpointAsync();

WriteHighlightedMessage(String.Format("Checkpointed partition: {0}", context.Lease.PartitionId));

currentBlockInitOffset = long.Parse(context.Lease.Offset);

stopwatch.Restart();

}

private void WriteHighlightedMessage(string message)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine(message);

Console.ResetColor();

}

}

The **EventProcessorHost** class calls this class to process device-to-cloud messages received from IoT Hub. The code in this class implements the logic to reliably store messages in a blob container and forward interactive messages to the Service Bus queue. The**OpenAsync** method, initializes the **currentBlockInitOffset** variable which tracks the current offset of the first message read by this event processor. Remember that each processor is responsible for a single partition.

The **ProcessEventsAsync** method receives a batch of messages from IoT Hub and processes them as follows: it sends interactive messages to the Service Bus queue and appends data point messages to the memory buffer called **toAppend**. If the memory buffer reaches the 4Mb block limit, or the Service Bus deduplication time windows has elapsed since the last checkpoint (one hour in this tutorial), then it triggers a checkpoint.

The method **AppendAndCheckpoint** first generates a blockId for the block to append. Azure storage requires all block ids to have the same length, so the method pads the offset with leading zeroes - currentBlockInitOffset.ToString("0000000000000000000000000"). Then, if a block with this id is already in the blob, the method overwrites it with the current contents of the buffer.

##### NOTE:

To simplify the code, this tutorial uses a single blob file per partition to store the messages. A real solution would implement file rolling by creating additional files when they reach a certain size (note Azure block blob can be at most 195Gb in size), or after a certain amount of time.

1. In the **Program** class, add the following **using** statements at the top:

Copy

using Microsoft.ServiceBus.Messaging;

1. Modify the **Main** method in the **Program** class as shown below, substituting the IoT Hub **iothubowner** connection string (from the [Get started with IoT Hub](https://azure.microsoft.com/en-us/documentation/articles/iot-hub-csharp-csharp-getstarted/) tutorial), the storage connection string, and the Service Bus connection string with **Send** permissions for the queue named **d2ctutorial**:

Copy

static void Main(string[] args)

{

string iotHubConnectionString = "{iot hub connection string}";

string iotHubD2cEndpoint = "messages/events";

StoreEventProcessor.StorageConnectionString = "{storage connection string}";

StoreEventProcessor.ServiceBusConnectionString = "{service bus send connection string}";

string eventProcessorHostName = Guid.NewGuid().ToString();

EventProcessorHost eventProcessorHost = new EventProcessorHost(eventProcessorHostName, iotHubD2cEndpoint, EventHubConsumerGroup.DefaultGroupName, iotHubConnectionString, StoreEventProcessor.StorageConnectionString, "messages-events");

Console.WriteLine("Registering EventProcessor...");

eventProcessorHost.RegisterEventProcessorAsync<StoreEventProcessor>().Wait();

Console.WriteLine("Receiving. Press enter key to stop worker.");

Console.ReadLine();

eventProcessorHost.UnregisterEventProcessorAsync().Wait();

}

##### NOTE:

For the sake of simplicity, this tutorial uses a single instance of the [EventProcessorHost](http://msdn.microsoft.com/library/azure/microsoft.servicebus.messaging.eventprocessorhost(v=azure.95).aspx) class. Please refer to [Event Hubs Programming Guide](https://azure.microsoft.com/en-us/documentation/articles/event-hubs-programming-guide/) for more information.

## 

## Receive interactive messages

In this section, you'll write a Windows console app that receives the interactive messages from the Service Bus queue. Refer to [Build multi-tier applications with Service Bus](https://azure.microsoft.com/en-us/documentation/articles/service-bus-dotnet-multi-tier-app-using-service-bus-queues/) for more information on how to architect a solution using Service Bus.

1. In the current Visual Studio solution, create a new Visual C# Windows project using the **Console Application** project template. Name the project **ProcessD2CInteractiveMessages**.
2. In Solution Explorer, right-click the **ProcessD2CInteractiveMessages** project, and then click **Manage NuGet Packages**. This displays the**NuGet Package Manager** window.
3. Search for **WindowsAzure.Service Bus**, click **Install**, and accept the terms of use. This downloads, installs, and adds a reference to the[Azure Service Bus](https://www.nuget.org/packages/WindowsAzure.ServiceBus), with all its dependencies.
4. Add the following **using** statement at the top of the **Program.cs** file:

Copy

using System.IO;

using Microsoft.ServiceBus.Messaging;

1. Finally, add the following lines to the **Main** method, substituting the connection string with **Listen** permissions for the queue named**d2ctutorial**:

Copy

Console.WriteLine("Process D2C Interactive Messages app\n");

string connectionString = "{service bus listen connection string}";

QueueClient Client = QueueClient.CreateFromConnectionString(connectionString, "d2ctutorial");

OnMessageOptions options = new OnMessageOptions();

options.AutoComplete = false;

options.AutoRenewTimeout = TimeSpan.FromMinutes(1);

Client.OnMessage((message) =>

{

try

{

var bodyStream = message.GetBody<Stream>();

bodyStream.Position = 0;

var bodyAsString = new StreamReader(bodyStream, Encoding.ASCII).ReadToEnd();

Console.WriteLine("Received message: {0} messageId: {1}", bodyAsString, message.MessageId);

message.Complete();

}

catch (Exception)

{

message.Abandon();

}

}, options);

Console.WriteLine("Receiving interactive messages from SB queue...");

Console.WriteLine("Press any key to exit.");

Console.ReadLine();

## Run the applications

Now you are ready to run the applications.

1. In Visual Studio, in Solution Explorer, right-click your solution and select **Set StartUp Projects**. Select **Multiple startup projects**, then select **Start** as the action for the **ProcessDeviceToCloudMessages**, **SimulatedDevice**, and **ProcessD2CInteractiveMessages** projects.
2. Press **F5** to start the three console applications. The **ProcessD2CInteractiveMessages** application should process every interactive message sent from the **SimulatedDevice** application.

##### NOTE:

In order to see updates in your blob file, you may need to reduce the **MAX\_BLOCK\_SIZE** constant in the **StoreEventProcessor** class to a smaller value such as **1024**. This is because it takes some time to reach the block size limit with the data sent by the simulated device. With a smaller block size, you will not need to wait so long to see the blob being created and updated. However, using a larger block size makes the application more scalable.

## Next steps

In this tutorial, you learned how to reliably process data point and interactive device-to-cloud messages using the [EventProcessorHost](http://msdn.microsoft.com/library/azure/microsoft.servicebus.messaging.eventprocessorhost(v=azure.95).aspx) class.

The [Uploading files from devices](https://azure.microsoft.com/en-us/documentation/articles/iot-hub-csharp-csharp-file-upload/) tutorial builds on this tutorial using analagous message processing logic and describes a pattern that makes use of cloud-to-device messages to facilitate file uploads from devices

Additional information on IoT Hub:

* [IoT Hub Overview](https://azure.microsoft.com/en-us/documentation/articles/iot-hub-what-is-iot-hub/)
* [IoT Hub Developer Guide](https://azure.microsoft.com/en-us/documentation/articles/iot-hub-devguide/)
* [IoT Hub Guidance](https://azure.microsoft.com/en-us/documentation/articles/iot-hub-guidance/)
* [Supported device platforms and languages](https://github.com/Azure/azure-iot-sdks/blob/master/doc/tested_configurations.md)
* [Azure IoT Developer Center](https://azure.microsoft.com/develop/iot)