# How to use a Worker Role to read telemetry data from an Event Hub and store it to Azure SQL Database using JSON functionalities

This sample shows how to use the **EventProcessorHost** to retrieve events from an **Event Hub** and store them in a batch mode to an **Azure SQL Database** using the [OPENJSON](https://msdn.microsoft.com/en-us/library/dn921885.aspx) function. The solution demonstrates how the use the following techniques:

* Send events to an [Event Hub](https://msdn.microsoft.com/en-us/library/azure/dn789973.aspx) using both AMQP and HTTPS transport protocols.
* Create an entity level shared access policy with only the Send claim. This key will be used to create SAS tokens, one for each publisher endpoint.
* Issue a SAS token to secure individual publisher endpoints.
* Use a SAS token to authenticate at a publisher endpoint level.
* Use the [EventProcessorHost](https://msdn.microsoft.com/en-us/library/microsoft.servicebus.messaging.eventprocessorhost(v=azure.95).aspx) to retrieve and process events from an event hub.
* Perform structured and sematic logging using a custom [EventSource](https://msdn.microsoft.com/en-us/library/system.diagnostics.tracing.eventsource%28v=vs.110%29.aspx?f=255&MSPPError=-2147217396) class and ETW Logs introduced by Azure SDK 2.5.
* Use the [OPENJSON](https://msdn.microsoft.com/en-us/library/dn921885.aspx) table-value function in a stored procedure to process a batch of rows.

**NOTE**: this article is not intended to provide an exhaustive analysis of the various batching techniques offered by Azure SQL Database. Relying on batching to optimize data ingestion is a topic by itself, if you’re interested in the details take a look at this dedicated article: [How to use batching to improve SQL Database application performance](https://azure.microsoft.com/en-us/documentation/articles/sql-database-use-batching-to-improve-performance/). Also look at [How to store Event Hub events to Azure SQL Database](https://code.msdn.microsoft.com/How-to-integrate-store-828769eb) for a version of the sample where the event processor uses a stored procedure with a [Table-Valued Parameter](https://msdn.microsoft.com/en-us/library/bb675163%28v=vs.110%29.aspx?f=255&MSPPError=-2147217396) to store multiple events in a batch mode to a table on an Azure SQL database.

# Scenario

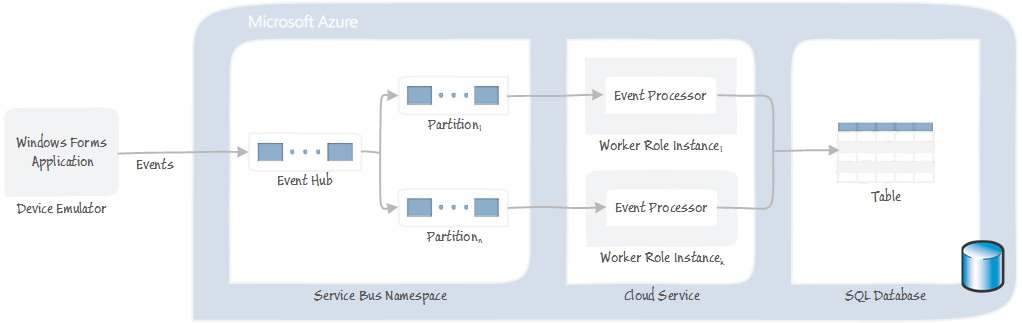
This solution simulates an Internet of Things (IoT) scenario where thousands of devices send events (e.g. sensor readings) to a backend system via a message broker. The backend system retrieves events from the messaging infrastrcure and store them to a persistent repository in a scalable manner.

# Architecture

The sample is structured as follows:

* A Windows Forms application can be used to create an event hub and an entity level shared access policy with only the Send access right.
* The same application can be used to simulate a configurable amount of devices that send readings into the event hub. Each device uses a separate publisher endpoint to send data to the underlying event hub and a separate SAS token to authenticate with the Service Bus namespace.
* An event hub is used to ingest device events.
* A worker role with multiple instances uses an [EventProcessorHost](https://msdn.microsoft.com/en-us/library/microsoft.servicebus.messaging.eventprocessorhost(v=azure.95).aspx) to read and process messages from the partitions of the event hub.
* The custom [EventProcessor](https://msdn.microsoft.com/en-us/library/microsoft.servicebus.messaging.ieventprocessor.aspx) class inserts a collection of events into a table of a SQL Database in a batch mode by invoking a stored procedure.
* The worker role uses a custom [EventSource](https://msdn.microsoft.com/en-us/library/system.diagnostics.tracing.eventsource%28v=vs.110%29.aspx?f=255&MSPPError=-2147217396) class and the Windows Azure Diagnostics support for ETW Events to write log data to table storage.
* The stored procedure uses the [OPENJSON](https://msdn.microsoft.com/en-us/library/dn921885.aspx) table-value function and the [MERGE](https://msdn.microsoft.com/en-us/library/bb510625.aspx) statement to implement an **UPSERT** mechanism.

The following picture shows the architecture of the solution:



# References

JSON Functionalities of Azure SQL Database

* [JSON in SQL Server 2016: Part 1 of 4](https://blogs.technet.microsoft.com/dataplatforminsider/2016/01/05/json-in-sql-server-2016-part-1-of-4/)
* [Channel9 Video: SQL Server 2016 and JSON Support](https://channel9.msdn.com/Shows/Data-Exposed/SQL-Server-2016-and-JSON-Support)
* [Reference Documentation](https://msdn.microsoft.com/en-us/library/dn921897.aspx)

Event Hubs

* [Event Hubs](http://azure.microsoft.com/en-us/services/event-hubs/)
* [Get started with Event Hubs](http://azure.microsoft.com/en-us/documentation/articles/service-bus-event-hubs-csharp-ephcs-getstarted/)
* [Event Hubs Programming Guide](https://msdn.microsoft.com/en-us/library/azure/dn789972.aspx)
* [Service Bus Event Hubs Getting Started](https://code.msdn.microsoft.com/windowsazure/Service-Bus-Event-Hub-286fd097)
* [Event Hubs Authentication and Security Model Overview](https://msdn.microsoft.com/en-us/library/azure/dn789974.aspx)
* [Service Bus Event Hubs Large Scale Secure Publishing](https://code.msdn.microsoft.com/windowsazure/Service-Bus-Event-Hub-99ce67ab)
* [Service Bus Event Hubs Direct Receivers](https://code.msdn.microsoft.com/windowsazure/Event-Hub-Direct-Receivers-13fa95c6)
* [Service Bus Explorer](https://code.msdn.microsoft.com/windowsapps/Service-Bus-Explorer-f2abca5a)
* [Episode 160: Event Hubs with Elio Damaggio](http://channel9.msdn.com/Shows/Cloud+Cover/Episode-160-Event-Hubs-with-Elio-Damaggio) (video)
* [Telemetry and Data Flow at Hyper-Scale: Azure Event Hub](http://channel9.msdn.com/Events/TechEd/Europe/2014/CDP-B307) (video)
* [Data Pipeline Guidance](https://github.com/mspnp/data-pipeline)  (Patterns & Practices solution)
* [Event Processor Host Best Practices Part 1](http://blogs.msdn.com/b/servicebus/archive/2015/01/16/event-processor-host-best-practices-part-1.aspx)
* [Event Processor Host Best Practices Part 2](http://blogs.msdn.com/b/servicebus/archive/2015/01/21/event-processor-host-best-practices-part-2.aspx)
* [How to create a Service Bus Namespace and an Event Hub using a PowerShell script](http://blogs.msdn.com/b/paolos/archive/2014/12/01/how-to-create-a-service-bus-namespace-and-an-event-hub-using-a-powershell-script.aspx)

ETW Logs

* [Diagnostics: Improved diagnostics logging with ETW](http://azure.microsoft.com/blog/2014/11/12/announcing-azure-sdk-2-5-for-net-and-visual-studio-2015-preview/)

# Visual Studio Solution

The Visual Studio solution includes the following projects:

* **CreateIoTDbWithMerge.sql**: this script can be used to create the SQL Server database used to store device events.
* **Entities**: this library contains the **Payload** class. This class defines the structure and content of the [EventData](https://msdn.microsoft.com/en-us/library/microsoft.servicebus.messaging.eventdata.aspx?f=255&MSPPError=-2147217396) message body.
* **EventProcessorHostWorkerRole**: this library defines the worker role used to handle the events from the event hub.
* **Helpers**: this library defines the **TraceEventSource** class used by the worker role to create ETW logs at runtime.
* **Sender**: this Windows Forms application can be used to create the Event Hub used by the sample and simulate a configurable amount of devices sending events to the event hub.
* **StoreEventsToAzureSqlDatabase**: this Azure project defines the cloud service hosting the worker role used to handle the events from the event hub.

**NOTE**: To reduce the size of the zip file, I deleted the NuGet packages. To repair the solution, make sure to right click the solution and select **Enable NuGet Package Restore**. For more information on this topic, see the following [post](http://blogs.4ward.it/enable-nuget-package-restore-in-visual-studio-and-tfs-2012-rc-to-building-windows-8-metro-apps/).

# Solution

This section briefly describes the individual components of the solution.

## SQL Azure Database

Run the **CreateIoTDbWithMerge.sql** script to create the database used by the solution. In particular, the script create the following artifacts:

* The **Events** table used to store events.
* The **sp\_InsertJsonEvents** stored procedure used to store events.

The stored procedure receives a single input parameter of type **nvarchar(max)** which contains the events to store in **JSON** format and uses the [MERGE](https://msdn.microsoft.com/en-us/library/bb510625.aspx) statement to implement an **UPSERT** mechanism. This technique is commonly used to implement idempotency: if an row already exists in the table with the a given EventId, the store procedure updates its columns, otherwise a new record is created. The stored procedure uses the [OPENJSON](https://msdn.microsoft.com/en-us/library/dn921885.aspx) table-value function that parses JSON text and returns objects and properties in JSON as rows and columns. [OPENJSON](https://msdn.microsoft.com/en-us/library/dn921885.aspx) provides a rowset view over a JSON document, with the ability to explicitly specify the columns in the rowset and the property paths to use to populate the columns. Since OPENJSON returns a set of rows, you can use [OPENJSON](https://msdn.microsoft.com/en-us/library/dn921885.aspx) function in FROM clause of Transact-SQL statements like any other table, view, or table-value function. The [OPENJSON](https://msdn.microsoft.com/en-us/library/dn921885.aspx) function is available only under compatibility level 130. If your database compatibility level is lower than 130, SQL Server will not be able to find and execute OPENJSON function. Other JSON functions are available at all compatibility levels. You can check compatibility level in sys.databases view or in database properties. You can change a compatibility level of database using the following command: **ALTER DATABASE DatabaseName SET COMPATIBILITY\_LEVEL = 130**. Note that compatibility level 120 might be default even in new Azure SQL Databases. For more information on the new JSON support in Azure SQL Database, see [JSON functionalities in Azure SQL Database](https://azure.microsoft.com/en-us/blog/json-functionalities-in-azure-sql-database-public-preview/).

USE IoTDB

GO

-- Drop sp\_InsertJsonEvents stored procedure

DROP PROCEDURE IF EXISTS [dbo].[sp\_InsertJsonEvents]

GO

-- Drop Events table

DROP TABLE IF EXISTS [dbo].[Events]

GO

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

-- Create Events table

CREATE TABLE [dbo].[Events](

[EventId] [int] NOT NULL,

[DeviceId] [int] NOT NULL,

[Value] [int] NOT NULL,

[Timestamp] [datetime2](7) NULL,

PRIMARY KEY CLUSTERED

(

[EventId] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

) ON [PRIMARY]

GO

-- Create sp\_InsertEvents stored procedure

CREATE PROCEDURE dbo.sp\_InsertJsonEvents

@Events NVARCHAR(MAX)

AS

BEGIN

MERGE INTO dbo.Events AS A

USING (SELECT \*

FROM OPENJSON(@Events)

WITH ([eventId] int, [deviceId] int, [value] int, [timestamp] datetime2(7))) B

ON (A.EventId = B.EventId)

WHEN MATCHED THEN

UPDATE SET A.DeviceId = B.DeviceId,

A.Value = B.Value,

A.Timestamp = B.Timestamp

WHEN NOT MATCHED THEN

INSERT (EventId, DeviceId, Value, Timestamp)

VALUES(B.EventId, B.DeviceId, B.Value, B.Timestamp);

END

GO

## Entities

The following table contains the code of the **Payload** class. This class is used to define the body of the messages sent to the event hub. Note that the properties of the class are decorated with the **JsonPropertyAttribute**. In fact, the code of both the client application and worker role use [Json.NET](http://www.newtonsoft.com/json) to serialize and deserialize the message content in JSON format.

#region Using Directives  
using System;  
using Newtonsoft.Json;   
#endregion  
  
namespace Microsoft.AzureCat.Samples.Entities  
{  
 [Serializable]  
 public class Payload  
 {  
 /// <summary>  
 /// Gets or sets the device id.  
 /// </summary>  
 [JsonProperty(PropertyName = "eventId", Order = 1)]  
 public int EventId { get; set; }  
  
 /// <summary>  
 /// Gets or sets the device id.  
 /// </summary>  
 [JsonProperty(PropertyName = "deviceId", Order = 2)]  
 public int DeviceId { get; set; }  
  
 /// <summary>  
 /// Gets or sets the device value.  
 /// </summary>  
 [JsonProperty(PropertyName = "value", Order = 3)]  
 public int Value { get; set; }  
  
 /// <summary>  
 /// Gets or sets the event timestamp.  
 /// </summary>  
 [JsonProperty(PropertyName = "timestamp", Order = 4)]  
 public DateTime Timestamp { get; set; }  
 }  
}

## Helpers

This library defines the **TraceEventSource** class used by the worker role to trace events to ETW logs. The WAD agent running on the worker role instances will read data out of local ETW logs and persist this data to a couple of storage tables (**WASDiagnosticTable** and **WADEventProcessorTable**) in the storage account configured for Windows Azure Diagnostics.

#region Using Directives  
using System;  
using System.Diagnostics.Tracing;  
using System.Runtime.CompilerServices;  
using Microsoft.WindowsAzure.ServiceRuntime;  
  
#endregion  
  
 namespace Microsoft.AzureCat.Samples.Helpers  
 {  
 [EventSource(Name = "TraceEventSource")]  
 public sealed class TraceEventSource : EventSource  
 {  
 #region Internal Enums  
 public class Keywords  
 {  
 public const EventKeywords EventHub = (EventKeywords)1;  
 public const EventKeywords DataBase = (EventKeywords)2;  
 public const EventKeywords Diagnostic = (EventKeywords)4;  
 public const EventKeywords Performance = (EventKeywords)8;  
 }  
 #endregion  
   
 #region Public Static Properties  
 public static readonly TraceEventSource Log = new TraceEventSource();   
 #endregion  
   
 #region Private Methods  
 [Event(1,  
 Message = "TraceIn",  
 Keywords = Keywords.Diagnostic,  
 Level = EventLevel.Verbose)]  
 private void TraceIn(string application,  
 string instance,  
 Guid activityId,  
 string description,  
 string source,  
 string method)  
 {  
 if (string.IsNullOrWhiteSpace(application) ||  
 string.IsNullOrWhiteSpace(instance))  
 {  
 return;  
 }  
 WriteEvent(1, application, instance, activityId, description, source, method);  
 }  
   
 [Event(2,  
 Message = "TraceOut",  
 Keywords = Keywords.Diagnostic,  
 Level = EventLevel.Verbose)]  
 private void TraceOut(string application,  
 string instance,  
 Guid activityId,  
 string description,  
 string source,  
 string method)  
 {  
 if (string.IsNullOrWhiteSpace(application) ||  
 string.IsNullOrWhiteSpace(instance))  
 {  
 return;  
 }  
 WriteEvent(2, application, instance, activityId, description, source, method);  
 }  
   
 [Event(3,  
 Message = "TraceApi",  
 Keywords = Keywords.Diagnostic,  
 Level = EventLevel.Informational)]  
 private void TraceExec(string application,  
 string instance,  
 Guid activityId,  
 double elapsed,  
 string description,  
 string source,  
 string method)  
 {  
 if (string.IsNullOrWhiteSpace(application) ||  
 string.IsNullOrWhiteSpace(instance) ||  
 string.IsNullOrWhiteSpace(description))  
 {  
 return;  
 }  
 WriteEvent(3, application, instance, activityId, elapsed, description, source, method);  
 }  
   
 [Event(4,  
 Message = "TraceInfo",  
 Keywords = Keywords.Diagnostic,  
 Level = EventLevel.Informational)]  
 private void TraceInfo(string application,  
 string instance,  
 string description,  
 string source,  
 string method)  
 {  
 if (string.IsNullOrWhiteSpace(application) ||  
 string.IsNullOrWhiteSpace(instance) ||  
 string.IsNullOrWhiteSpace(description))  
 {  
 return;  
 }  
 WriteEvent(4, application, instance, description, source, method);  
 }  
   
 [Event(5,  
 Message = "TraceError",  
 Keywords = Keywords.Diagnostic,  
 Level = EventLevel.Error)]  
 private void TraceError(string application,  
 string instance,  
 Guid activityId,  
 string exception,  
 string innerException,  
 string source,  
 string method)  
 {  
 if (string.IsNullOrWhiteSpace(application) ||  
 string.IsNullOrWhiteSpace(instance) ||  
 string.IsNullOrWhiteSpace(exception))  
 {  
 return;  
 }  
 WriteEvent(5,   
 application,   
 instance,   
 activityId,   
 exception,   
 string.IsNullOrWhiteSpace(innerException) ?   
 string.Empty :   
 innerException,   
 source,   
 method);  
 }  
   
 [Event(6,  
 Message = "OpenPartition",  
 Keywords = Keywords.EventHub,  
 Level = EventLevel.Informational)]  
 private void OpenPartition(string application,   
 string instance,   
 string eventHub,   
 string consumerGroup,   
 string partitionId,   
 string source,   
 string method)  
 {  
 if (string.IsNullOrWhiteSpace(application) ||  
 string.IsNullOrWhiteSpace(instance) ||  
 string.IsNullOrWhiteSpace(eventHub) ||  
 string.IsNullOrWhiteSpace(consumerGroup) ||  
 string.IsNullOrWhiteSpace(partitionId))  
 {  
 return;  
 }  
 WriteEvent(6,   
 application,   
 instance,   
 eventHub,   
 consumerGroup,   
 partitionId,   
 source,   
 method);  
 }  
   
 [Event(7,  
 Message = "ClosePartition",  
 Keywords = Keywords.EventHub,  
 Level = EventLevel.Informational)]  
 private void ClosePartition(string application,   
 string instance,   
 string eventHub,   
 string consumerGroup,   
 string partitionId,   
 string reason,   
 string source,   
 string method)  
 {  
 if (string.IsNullOrWhiteSpace(application) ||  
 string.IsNullOrWhiteSpace(instance) ||  
 string.IsNullOrWhiteSpace(eventHub) ||  
 string.IsNullOrWhiteSpace(consumerGroup) ||  
 string.IsNullOrWhiteSpace(partitionId))  
 {  
 return;  
 }  
 WriteEvent(7,   
 application,   
 instance,   
 eventHub,   
 consumerGroup,   
 partitionId,   
 reason,   
 source,   
 method);  
 }  
   
 [Event(8,  
 Message = "ProcessEvents",  
 Keywords = Keywords.EventHub,  
 Level = EventLevel.Informational)]  
 private void ProcessEvents(string application,   
 string instance,   
 string eventHub,   
 string consumerGroup,   
 string partitionId,   
 int messageCount,   
 string source,   
 string method)  
 {  
 if (string.IsNullOrWhiteSpace(application) ||  
 string.IsNullOrWhiteSpace(instance) ||  
 string.IsNullOrWhiteSpace(eventHub) ||  
 string.IsNullOrWhiteSpace(consumerGroup) ||  
 string.IsNullOrWhiteSpace(partitionId))  
 {  
 return;  
 }  
 if (IsEnabled())  
 {  
 WriteEvent(8,   
 application,   
 instance,   
 eventHub,   
 consumerGroup,   
 partitionId,   
 messageCount,   
 source,   
 method);  
 }  
 }   
 #endregion  
   
 #region Public Methods  
 [NonEvent]  
 public void TraceApi(TimeSpan elapsed,  
 string description,  
 [CallerFilePath] string source = "",  
 [CallerMemberName] string method = "")  
 {  
 if (IsEnabled())  
 {  
 TraceExec(RoleEnvironment.CurrentRoleInstance.Role.Name,  
 RoleEnvironment.CurrentRoleInstance.Id,  
 Guid.NewGuid(),  
 elapsed.TotalMilliseconds,  
 description,  
 source,  
 method);  
 }  
 }  
   
 [NonEvent]  
 public void TraceApi(Guid activityId,  
 TimeSpan elapsed,  
 string description,  
 [CallerFilePath] string source = "",  
 [CallerMemberName] string method = "")  
 {  
 if (IsEnabled())  
 {  
 TraceExec(RoleEnvironment.CurrentRoleInstance.Role.Name,  
 RoleEnvironment.CurrentRoleInstance.Id,  
 activityId,  
 elapsed.TotalMilliseconds,  
 description,  
 source,  
 method);  
 }  
 }  
   
 [NonEvent]  
 public void TraceIn([CallerFilePath] string source = "",  
 [CallerMemberName] string method = "")  
 {  
 if (IsEnabled())  
 {  
 TraceIn(RoleEnvironment.CurrentRoleInstance.Role.Name,  
 RoleEnvironment.CurrentRoleInstance.Id,  
 Guid.NewGuid(),  
 string.Empty,  
 source,  
 method);  
 }  
 }  
   
 [NonEvent]  
 public void TraceOut([CallerFilePath] string source = "",  
 [CallerMemberName] string method = "")  
 {  
 if (IsEnabled())  
 {  
 TraceOut(RoleEnvironment.CurrentRoleInstance.Role.Name,  
 RoleEnvironment.CurrentRoleInstance.Id,  
 Guid.NewGuid(),  
 string.Empty,  
 source,  
 method);  
 }  
 }  
   
 [NonEvent]  
 public void TraceInfo(string description,   
 [CallerFilePath] string source = "",   
 [CallerMemberName] string method = "")  
 {  
 if (IsEnabled())  
 {  
 TraceInfo(RoleEnvironment.CurrentRoleInstance.Role.Name,  
 RoleEnvironment.CurrentRoleInstance.Id,  
 description,  
 source,  
 method);  
 }  
 }  
   
 [NonEvent]  
 public void TraceError(string exception,   
 string innerException,  
 [CallerFilePath] string source = "",   
 [CallerMemberName] string method = "")  
 {  
 if (string.IsNullOrWhiteSpace(exception))  
 {  
 return;  
 }  
 if (IsEnabled())  
 {  
 TraceError(RoleEnvironment.CurrentRoleInstance.Role.Name,  
 RoleEnvironment.CurrentRoleInstance.Id,  
 Guid.NewGuid(),  
 exception,   
 string.IsNullOrWhiteSpace(innerException) ? string.Empty : innerException,  
 source,  
 method);  
 }  
 }  
   
 [NonEvent]  
 public void OpenPartition(string eventHub,   
 string consumerGroup,   
 string partitionId,  
 [CallerFilePath] string source = "",  
 [CallerMemberName] string method = "")  
 {  
 if (string.IsNullOrWhiteSpace(eventHub) ||  
 string.IsNullOrWhiteSpace(consumerGroup) ||  
 string.IsNullOrWhiteSpace(partitionId))  
 {  
 return;  
 }  
 if (IsEnabled())  
 {  
 OpenPartition(RoleEnvironment.CurrentRoleInstance.Role.Name,  
 RoleEnvironment.CurrentRoleInstance.Id,  
 eventHub,  
 consumerGroup,  
 partitionId,  
 source,  
 method);  
 }  
 }  
   
 [NonEvent]  
 public void ClosePartition(string eventHub,   
 string consumerGroup,   
 string partitionId,  
 string reason,  
 [CallerFilePath] string source = "",  
 [CallerMemberName] string method = "")  
 {  
 if (string.IsNullOrWhiteSpace(eventHub) ||  
 string.IsNullOrWhiteSpace(consumerGroup) ||  
 string.IsNullOrWhiteSpace(partitionId))  
 {  
 return;  
 }  
 if (IsEnabled())  
 {  
 ClosePartition(RoleEnvironment.CurrentRoleInstance.Role.Name,  
 RoleEnvironment.CurrentRoleInstance.Id,  
 eventHub,  
 consumerGroup,  
 partitionId,  
 reason,  
 source,  
 method);  
 }  
 }  
   
 [NonEvent]  
 public void ProcessEvents(string eventHub,   
 string consumerGroup,   
 string partitionId,   
 int messageCount,  
 [CallerFilePath] string source = "",  
 [CallerMemberName] string method = "")  
 {  
 if (string.IsNullOrWhiteSpace(eventHub) ||  
 string.IsNullOrWhiteSpace(consumerGroup) ||  
 string.IsNullOrWhiteSpace(partitionId))  
 {  
 return;  
 }  
 if (IsEnabled())  
 {  
 ProcessEvents(RoleEnvironment.CurrentRoleInstance.Role.Name,  
 RoleEnvironment.CurrentRoleInstance.Id,  
 eventHub,  
 consumerGroup,  
 partitionId,  
 messageCount,  
 source,  
 method);  
 }  
 }  
   
 #endregion  
 }  
 }

## EventProcessorHostWorkerRole

The following table contains the code of the **WorkerRole** class. The class reads the following settings from the service configuration file and then create an instance of the **EventProcessorHost** class:

* **SqlDatabaseConnectionString**: connectionstring of the Azure SQL Database where to store events.
* **InsertStoredProcedure**: name of the stored procedure used to insert an array of events expressed as a JSON array.
* **StorageAccountConnectionString**: connectionstring of the **Storage Account** used by the **EventProcessorHost**.
* **ServiceBusConnectionString**: connectionstring of the **Service Bus** namespace hosting the **Event Hub** used by the solution.
* **EventHubName**: the name of the Event Hub used by the solution.
* **ConsumerGroupName**: the name of the **Consumer Grou**p used by the **Worker Role** to read events from the **Event Hub**.
* #region Using Directives  
  using System;  
  using System.Linq;  
  using System.Net;  
  using System.Threading;  
  using System.Threading.Tasks;  
  using Microsoft.ServiceBus.Messaging;  
  using Microsoft.WindowsAzure.ServiceRuntime;  
  using Microsoft.AzureCat.Samples.Helpers;  
  #endregion  
    
  namespace Microsoft.AzureCat.Samples.EventProcessorHostWorkerRole  
  {  
   public class WorkerRole : RoleEntryPoint  
   {  
   #region Private Constants  
   //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
   // Messages & Formats  
   //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
   private const string RoleEnvironmentSettingFormat = "Configuration Setting [{0}] = [{1}].";  
   private const string RoleEnvironmentConfigurationSettingChangedFormat = "The setting [{0}] is changed: new value = [{1}].";  
   private const string RoleEnvironmentConfigurationSettingChangingFormat = "The setting [{0}] is changing: old value = [{1}].";  
   private const string RoleEnvironmentTopologyChangedFormat = "The topology for the [{0}] role is changed.";  
   private const string RoleEnvironmentTopologyChangingFormat = "The topology for the [{0}] role is changing.";  
   private const string RoleInstanceCountFormat = "[Role {0}] instance count = [{1}].";  
   private const string RoleInstanceEndpointCountFormat = "[Role {0}] instance endpoints count = [{1}].";  
   private const string RoleInstanceEndpointFormat = "[Role {0}] instance endpoint [{1}]: protocol = [{2}] address = [{3}] port = [{4}].";  
   private const string RoleInstanceIdFormat = "[Role {0}] instance Id = [{1}].";  
   private const string RoleInstanceStatusFormat = "[Role {0}] instance Id = [{1}] Status = [{2}].";  
   private const string Unknown = "Unknown";  
   private const string RegisteringEventProcessor = "Registering Event Processor [EventProcessor]... ";  
   private const string EventProcessorRegistered = "Event Processor [EventProcessor] successfully registered. ";  
    
   //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
   // Settings  
   //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
   private const string SqlDatabaseConnectionStringSetting = "SqlDatabaseConnectionString";  
   private const string InsertStoredProcedureSetting = "InsertStoredProcedure";  
   private const string StorageAccountConnectionStringSetting = "StorageAccountConnectionString";  
   private const string ServiceBusConnectionStringSetting = "ServiceBusConnectionString";  
   private const string EventHubNameSetting = "EventHubName";  
   private const string ConsumerGroupNameSetting = "ConsumerGroupName";  
   #endregion  
    
   #region Private Fields  
   private string eventHubName;  
   private string consumerGroupName;  
   private string sqlDatabaseConnectionString;  
   private string insertStoredProcedure;  
   private string storageAccountConnectionString;  
   private string serviceBusConnectionString;  
   private EventProcessorHost eventProcessorHost;  
   #endregion  
    
   #region Public Methods  
   public override void Run()  
   {  
   try  
   {  
   // TraceIn  
   TraceEventSource.Log.TraceIn();  
    
   while (true)  
   {  
   Thread.Sleep(10000);  
   }  
   // ReSharper disable once FunctionNeverReturns  
   }  
   catch (Exception ex)  
   {  
   // Trace Exception  
   TraceEventSource.Log.TraceError(ex.Message, ex.InnerException?.Message ?? string.Empty);  
   }  
   finally  
   {  
   // TraceOut  
   TraceEventSource.Log.TraceOut();  
   }  
   }  
    
   public override bool OnStart()  
   {  
   try  
   {  
   // TraceIn  
   TraceEventSource.Log.TraceIn();  
    
   // Set Default values for the ServicePointManager  
   ServicePointManager.UseNagleAlgorithm = false;  
   ServicePointManager.Expect100Continue = false;  
    
   // Setting RoleEnvironment event handlers  
   RoleEnvironment.Changed += RoleEnvironment\_Changed;  
   RoleEnvironment.Changing += RoleEnvironment\_Changing;  
   RoleEnvironment.StatusCheck += RoleEnvironment\_StatusCheck;  
   RoleEnvironment.Stopping += RoleEnvironment\_Stopping;  
    
   // Read Configuration Settings  
   ReadConfigurationSettings();  
    
   // Start Event Processor  
   StartEventProcessorAsync().Wait();  
    
   // Run base.OnStart method  
   return base.OnStart();  
   }  
   catch (Exception ex)  
   {  
   // Trace Exception  
   TraceEventSource.Log.TraceError(ex.Message, ex.InnerException?.Message ?? string.Empty);  
   return false;  
   }  
   finally  
   {  
   // TraceOut  
   TraceEventSource.Log.TraceOut();  
   }  
   }  
   #endregion  
    
   #region Private Methods  
    
   private void ReadConfigurationSettings()  
   {  
   // Read sql database connectionstring setting   
   sqlDatabaseConnectionString = CloudConfigurationHelper.GetSetting(SqlDatabaseConnectionStringSetting);  
   TraceEventSource.Log.TraceInfo(string.Format(RoleEnvironmentSettingFormat,  
   SqlDatabaseConnectionStringSetting,  
   sqlDatabaseConnectionString));  
    
   // Read insert stored procedure setting   
   insertStoredProcedure = CloudConfigurationHelper.GetSetting(InsertStoredProcedureSetting);  
   TraceEventSource.Log.TraceInfo(string.Format(RoleEnvironmentSettingFormat,  
   InsertStoredProcedureSetting,  
   insertStoredProcedure));  
    
   // Read storage account connectionstring setting   
   storageAccountConnectionString = CloudConfigurationHelper.GetSetting(StorageAccountConnectionStringSetting);  
   TraceEventSource.Log.TraceInfo(string.Format(RoleEnvironmentSettingFormat,  
   StorageAccountConnectionStringSetting,  
   storageAccountConnectionString));  
    
   // Read service bus connectionstring setting   
   serviceBusConnectionString = CloudConfigurationHelper.GetSetting(ServiceBusConnectionStringSetting);  
   TraceEventSource.Log.TraceInfo(string.Format(RoleEnvironmentSettingFormat,  
   ServiceBusConnectionStringSetting,  
   serviceBusConnectionString));  
    
   // Read event hub name setting  
   eventHubName = CloudConfigurationHelper.GetSetting(EventHubNameSetting);  
   TraceEventSource.Log.TraceInfo(string.Format(RoleEnvironmentSettingFormat,  
   ServiceBusConnectionStringSetting,  
   serviceBusConnectionString));  
    
   // Read event consumer group name setting  
   consumerGroupName = CloudConfigurationHelper.GetSetting(ConsumerGroupNameSetting);  
   TraceEventSource.Log.TraceInfo(string.Format(RoleEnvironmentSettingFormat,  
   ServiceBusConnectionStringSetting,  
   serviceBusConnectionString));  
   }  
    
   private async Task StartEventProcessorAsync()  
   {  
   try  
   {  
   // TraceIn  
   TraceEventSource.Log.TraceIn();  
   var eventHubClient = EventHubClient.CreateFromConnectionString(serviceBusConnectionString, eventHubName);  
    
   // Get the default Consumer Group  
   eventProcessorHost = new EventProcessorHost(RoleEnvironment.CurrentRoleInstance.Id,  
   eventHubClient.Path.ToLower(),  
   consumerGroupName.ToLower(),  
   serviceBusConnectionString,  
   storageAccountConnectionString)  
   {  
   PartitionManagerOptions = new PartitionManagerOptions  
   {  
   AcquireInterval = TimeSpan.FromSeconds(10), // Default is 10 seconds  
   RenewInterval = TimeSpan.FromSeconds(10), // Default is 10 seconds  
   LeaseInterval = TimeSpan.FromSeconds(30) // Default value is 30 seconds  
   }  
   };  
   TraceEventSource.Log.TraceInfo(RegisteringEventProcessor);  
   var eventProcessorOptions = new EventProcessorOptions  
   {  
   InvokeProcessorAfterReceiveTimeout = true,  
   MaxBatchSize = 100,  
   PrefetchCount = 100,  
   ReceiveTimeOut = TimeSpan.FromSeconds(30),  
   };  
   eventProcessorOptions.ExceptionReceived += eventProcessorOptions\_ExceptionReceived;  
   await eventProcessorHost.RegisterEventProcessorFactoryAsync(  
   new EventProcessorFactory<EventProcessor>(sqlDatabaseConnectionString,   
   insertStoredProcedure),  
   eventProcessorOptions);  
   TraceEventSource.Log.TraceInfo(EventProcessorRegistered);  
   }  
   catch (Exception ex)  
   {  
   // Trace Exception  
   TraceEventSource.Log.TraceError(ex.Message, ex.InnerException?.Message ?? string.Empty);  
   }  
   finally  
   {  
   // TraceOut  
   TraceEventSource.Log.TraceOut();  
   }  
   }  
    
   void eventProcessorOptions\_ExceptionReceived(object sender, ExceptionReceivedEventArgs e)  
   {  
   if (e?.Exception == null)  
   {  
   return;  
   }  
    
   // Trace Exception  
   TraceEventSource.Log.TraceError(e.Exception.Message, e.Exception.InnerException?.Message ?? string.Empty);  
   }  
    
   /// <summary>  
   /// Occurs after a change to the service configuration is applied to the running instances of a role.  
   /// </summary>  
   /// <param name="sender">The source of the event.</param>  
   /// <param name="e">Represents the arguments for the Changed event, which occurs after a configuration change has been applied to a role instance.</param>  
   private static void RoleEnvironment\_Changed(object sender, RoleEnvironmentChangedEventArgs e)  
   {  
   try  
   {  
   // TraceIn  
   TraceEventSource.Log.TraceIn();  
    
   // Get the list of configuration setting changes  
   var settingChanges = e.Changes.OfType<RoleEnvironmentConfigurationSettingChange>();  
    
   foreach (var settingChange in settingChanges)  
   {  
   var value = CloudConfigurationHelper.GetSetting(settingChange.ConfigurationSettingName);  
   TraceEventSource.Log.TraceInfo(string.Format(RoleEnvironmentConfigurationSettingChangedFormat,  
   settingChange.ConfigurationSettingName ?? string.Empty,  
   string.IsNullOrEmpty(value) ? string.Empty : value));  
   }  
    
   // Get the list of configuration changes  
   var topologyChanges = e.Changes.OfType<RoleEnvironmentTopologyChange>();  
    
   foreach (var roleName in topologyChanges.Select(topologyChange => topologyChange.RoleName))  
   {  
   TraceEventSource.Log.TraceInfo(string.Format(RoleEnvironmentTopologyChangedFormat,  
   string.IsNullOrEmpty(roleName) ?   
   Unknown :   
   roleName));  
   if (string.IsNullOrEmpty(roleName))  
   {  
   continue;  
   }  
   var role = RoleEnvironment.Roles[roleName];  
   if (role == null)  
   {  
   continue;  
   }  
   TraceEventSource.Log.TraceInfo(string.Format(RoleInstanceCountFormat,   
   roleName,   
   role.Instances.Count));  
   foreach (var roleInstance in role.Instances)  
   {  
   TraceEventSource.Log.TraceInfo(string.Format(RoleInstanceIdFormat,   
   roleName,   
   roleInstance.Id));  
   TraceEventSource.Log.TraceInfo(string.Format(RoleInstanceEndpointCountFormat,  
   roleName, roleInstance.InstanceEndpoints.Count));  
   foreach (var instanceEndpoint in roleInstance.InstanceEndpoints)  
   {  
   TraceEventSource.Log.TraceInfo(string.Format(RoleInstanceEndpointFormat,  
   roleName,  
   instanceEndpoint.Key,  
   instanceEndpoint.Value.Protocol,  
   instanceEndpoint.Value.IPEndpoint.Address,  
   instanceEndpoint.Value.IPEndpoint.Port));  
   }  
   }  
   }  
   }  
   catch (Exception ex)  
   {  
   // Trace Exception  
   TraceEventSource.Log.TraceError(ex.Message, ex.InnerException?.Message ?? string.Empty);  
   }  
   finally  
   {  
   // TraceOut  
   TraceEventSource.Log.TraceOut();  
   }  
   }  
    
   /// <summary>  
   /// Occurs before a change to the service configuration is applied to the running instances of a role.   
   /// </summary>  
   /// <param name="sender">The source of the event.</param>  
   /// <param name="e">presents the arguments for the Changing event,   
   /// which occurs before a configuration change is applied to a role instance. </param>  
   private static void RoleEnvironment\_Changing(object sender, RoleEnvironmentChangingEventArgs e)  
   {  
   try  
   {  
   // TraceIn  
   TraceEventSource.Log.TraceIn();  
    
   // Get the list of configuration setting changes  
   var settingChanges = e.Changes.OfType<RoleEnvironmentConfigurationSettingChange>();  
    
   foreach (var settingChange in settingChanges)  
   {  
   var value = CloudConfigurationHelper.GetSetting(settingChange.ConfigurationSettingName);  
   TraceEventSource.Log.TraceInfo(string.Format(RoleEnvironmentConfigurationSettingChangingFormat,  
   settingChange.ConfigurationSettingName,  
   string.IsNullOrEmpty(value) ? string.Empty : value));  
   }  
    
   // Get the list of configuration changes  
   var topologyChanges = e.Changes.OfType<RoleEnvironmentTopologyChange>();  
    
   foreach (var roleName in topologyChanges.Select(topologyChange => topologyChange.RoleName))  
   {  
   TraceEventSource.Log.TraceInfo(string.Format(RoleEnvironmentTopologyChangingFormat,  
   string.IsNullOrEmpty(roleName) ?   
   Unknown :   
   roleName));  
   if (string.IsNullOrEmpty(roleName))  
   {  
   continue;  
   }  
   var role = RoleEnvironment.Roles[roleName];  
   if (role == null)  
   {  
   continue;  
   }  
   TraceEventSource.Log.TraceInfo(string.Format(RoleInstanceCountFormat, roleName, role.Instances.Count));  
   foreach (var roleInstance in role.Instances)  
   {  
   TraceEventSource.Log.TraceInfo(string.Format(RoleInstanceIdFormat, roleName, roleInstance.Id));  
   TraceEventSource.Log.TraceInfo(string.Format(RoleInstanceEndpointCountFormat,  
   roleName,   
   roleInstance.InstanceEndpoints.Count));  
   foreach (var instanceEndpoint in roleInstance.InstanceEndpoints)  
   {  
   TraceEventSource.Log.TraceInfo(string.Format(RoleInstanceEndpointFormat,  
   roleName,  
   instanceEndpoint.Key,  
   instanceEndpoint.Value.Protocol,  
   instanceEndpoint.Value.IPEndpoint.Address,  
   instanceEndpoint.Value.IPEndpoint.Port));  
   }  
   }  
   }  
   }  
   catch (Exception ex)  
   {  
   // Trace Exception  
   TraceEventSource.Log.TraceError(ex.Message, ex.InnerException?.Message ?? string.Empty);  
   }  
   finally  
   {  
   // TraceOut  
   TraceEventSource.Log.TraceOut();  
   }  
   }  
    
   /// <summary>  
   /// Occurs at a regular interval to indicate the status of a role instance.   
   /// </summary>  
   /// <param name="sender">The source of the event.</param>  
   /// <param name="e">Represents the arguments for the StatusCheck event,   
   /// which occurs at a regular interval to indicate the status of a role instance.</param>  
   private static void RoleEnvironment\_StatusCheck(object sender, RoleInstanceStatusCheckEventArgs e)  
   {  
   try  
   {  
   // TraceIn  
   TraceEventSource.Log.TraceIn();  
    
   // Write Role Instance Status  
   TraceEventSource.Log.TraceInfo(string.Format(RoleInstanceStatusFormat,  
   RoleEnvironment.CurrentRoleInstance.Role.Name,  
   RoleEnvironment.CurrentRoleInstance.Id,  
   e.Status));  
   }  
   catch (Exception ex)  
   {  
   // Trace Exception  
   TraceEventSource.Log.TraceError(ex.Message, ex.InnerException?.Message ?? string.Empty);  
   }  
   finally  
   {  
   // TraceOut  
   TraceEventSource.Log.TraceOut();  
   }  
   }  
    
   /// <summary>  
   /// Occurs when a role instance is about to be stopped.   
   /// </summary>  
   /// <param name="sender">The source of the event.</param>  
   /// <param name="e">Represents the arguments for the Stopping event,   
   /// which occurs when a role instance is being stopped. </param>  
   private static void RoleEnvironment\_Stopping(object sender, RoleEnvironmentStoppingEventArgs e)  
   {  
   try  
   {  
   // TraceIn  
   TraceEventSource.Log.TraceIn();  
   }  
   catch (Exception ex)  
   {  
   // Trace Exception  
   TraceEventSource.Log.TraceError(ex.Message, ex.InnerException?.Message ?? string.Empty);  
   }  
   finally  
   {  
   // TraceOut  
   TraceEventSource.Log.TraceOut();  
   }  
   }  
   #endregion  
   }  
  }

## EventProcessor

The following table contains the code of the **EventProcessor** class. In particular, the **ProcessEventsAsync** method writes events to an **Azure SQL Database** in a batch mode by invoking a stored procedure. Note how the code first extracts the payload from the **EventData** objects contained in the **events** collection and then serializes the resulting **List** collection into a JSON array using the **JsonConvert.SerializeObject** method. The string returned by this call is used as value of **@Events** parameter of the **sp\_InsertJsonEvents** stored procedure.

#region Using Directives

using System;

using System.Collections.Generic;

using System.Data;

using System.Data.SqlClient;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Microsoft.ServiceBus.Messaging;

using Microsoft.AzureCat.Samples.Helpers;

#endregion

namespace Microsoft.AzureCat.Samples.EventProcessorHostWorkerRole

{

public class EventProcessor : IEventProcessor

{

#region Private Fields

private readonly string sqlDatabaseConnectionString;

private readonly string insertStoredProcedure;

#endregion

#region Public Constructors

public EventProcessor(string sqlDatabaseConnectionString, string insertStoredProcedure)

{

this.sqlDatabaseConnectionString = sqlDatabaseConnectionString;

this.insertStoredProcedure = insertStoredProcedure;

}

#endregion

#region IEventProcessor Methods

public Task OpenAsync(PartitionContext context)

{

try

{

// TraceIn

TraceEventSource.Log.TraceIn();

// Trace Open Partition

TraceEventSource.Log.OpenPartition(context.EventHubPath,

context.ConsumerGroupName,

context.Lease.PartitionId);

}

catch (Exception ex)

{

// Trace Exception

TraceEventSource.Log.TraceError(ex.Message, ex.InnerException?.Message ?? string.Empty);

}

finally

{

// TraceOut

TraceEventSource.Log.TraceOut();

}

return Task.FromResult<object>(null);

}

public async Task ProcessEventsAsync(PartitionContext context, IEnumerable<EventData> events)

{

try

{

// TraceIn

TraceEventSource.Log.TraceIn();

if (events == null)

{

return;

}

var eventList = events.Select(e => Encoding.UTF8.GetString(e.GetBytes())).ToList();

if (!eventList.Any())

{

return;

}

// Trace Process Events

TraceEventSource.Log.ProcessEvents(context.EventHubPath,

context.ConsumerGroupName,

context.Lease.PartitionId,

eventList.Count);

using (var sqlConnection = new SqlConnection(sqlDatabaseConnectionString))

{

await sqlConnection.OpenAsync();

// Create command

var sqlCommand = new SqlCommand(insertStoredProcedure, sqlConnection)

{

CommandType = CommandType.StoredProcedure

};

// Add table-valued parameter

sqlCommand.Parameters.Add(new SqlParameter

{

ParameterName = "@Events",

SqlDbType = SqlDbType.NVarChar,

Size = -1,

Value = GetJsonArray(eventList)

});

// Execute the query

await sqlCommand.ExecuteNonQueryAsync().ConfigureAwait(false);

}

await context.CheckpointAsync();

}

catch (Exception ex)

{

// Trace Exception

TraceEventSource.Log.TraceError(ex.Message, ex.InnerException?.Message ?? string.Empty);

}

finally

{

// TraceOut

TraceEventSource.Log.TraceOut();

}

}

public async Task CloseAsync(PartitionContext context, CloseReason reason)

{

try

{

// TraceIn

TraceEventSource.Log.TraceIn();

// Trace Open Partition

TraceEventSource.Log.ClosePartition(context.EventHubPath,

context.ConsumerGroupName,

context.Lease.PartitionId,

reason.ToString());

if (reason == CloseReason.Shutdown)

{

await context.CheckpointAsync();

}

}

catch (Exception ex)

{

// Trace Exception

TraceEventSource.Log.TraceError(ex.Message, ex.InnerException?.Message ?? string.Empty);

}

finally

{

// TraceOut

TraceEventSource.Log.TraceOut();

}

}

#endregion

#region Private Methods

private string GetJsonArray(IReadOnlyList<string> list)

{

if (list == null || list.Count == 0)

{

return null;

}

var builder = new StringBuilder("[");

for (var i = 0; i < list.Count; i++)

{

builder.Append(i == 0 ? list[0] : $",{list[i]}");

}

builder.Append("]");

return builder.ToString();

}

#endregion

}

}

## StoreEventsToAzureSqlDatabase

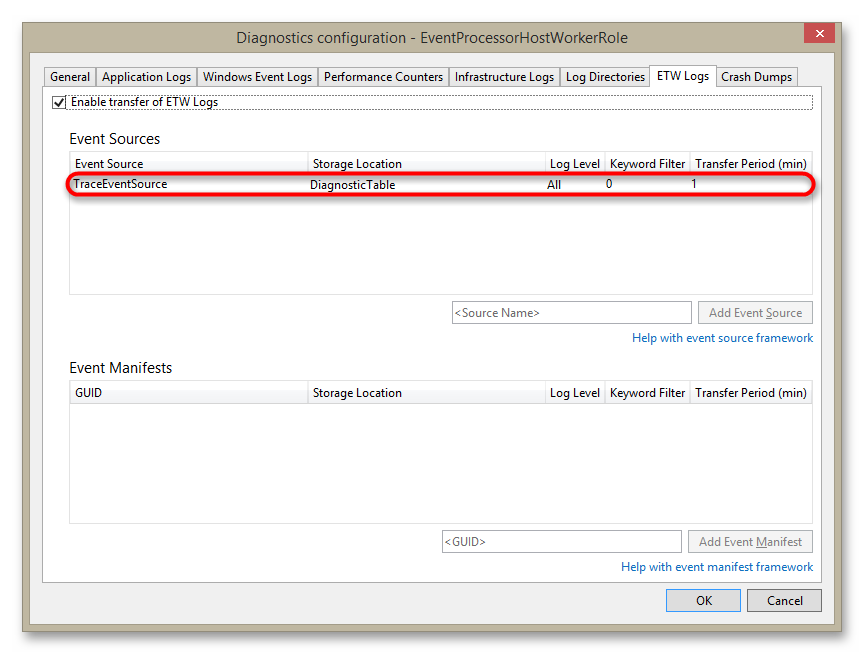
This project defines the cloud service hosting the worker role. The following table contains the service definition file of the cloud service.

<?xml version="1.0" encoding="utf-8"?>  
<ServiceDefinition name="StoreEventsToAzureSqlDatabase"   
 xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/ServiceDefinition"   
 schemaVersion="2014-06.2.4">  
 <WorkerRole name="EventProcessorHostWorkerRole" vmsize="Small">  
 <ConfigurationSettings>  
 <Setting name="SqlDatabaseConnectionString" />  
 <Setting name="InsertStoredProcedure" />  
 <Setting name="StorageAccountConnectionString" />  
 <Setting name="ServiceBusConnectionString" />  
 <Setting name="EventHubName" />  
 <Setting name="ConsumerGroupName" />  
 </ConfigurationSettings>  
 <Imports>  
 <Import moduleName="RemoteAccess" />  
 <Import moduleName="RemoteForwarder" />  
 </Imports>  
 </WorkerRole>  
</ServiceDefinition>

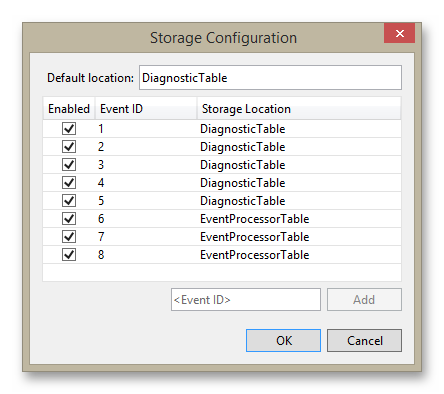
The following table contains the service definition file of the cloud service. Make sure to substitute the placeholders with the expected information before deploying the could service to Azure.

<?xml version="1.0" encoding="utf-8"?>  
<ServiceConfiguration serviceName="StoreEventsToAzureSqlDatabase"   
 xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/ServiceConfiguration"   
 osFamily="4"   
 osVersion="\*"   
 schemaVersion="2014-06.2.4">  
 <Role name="EventProcessorHostWorkerRole">  
 <Instances count="2" />  
 <ConfigurationSettings>  
 <Setting name="SqlDatabaseConnectionString" value="[AZURE SQL DATABASE CONNECTION STRING]" />  
 <Setting name="InsertStoredProcedure" value="sp\_InsertJsonEvents" />  
 <Setting name="StorageAccountConnectionString" value="[STORAGE ACCOUNT CONNECTION STRING]" />  
 <Setting name="ServiceBusConnectionString" value="[SERVICE BUS CONNECTION STRING]" />  
 <Setting name="EventHubName" value="[EVENT HUB NAME]" />  
 <Setting name="ConsumerGroupName" value="[CONSUMER GROUP NAME]" />  
 <Setting name="Microsoft.WindowsAzure.Plugins.RemoteAccess.Enabled" value="true" />  
 <Setting name="Microsoft.WindowsAzure.Plugins.RemoteAccess.AccountUsername" value="..." />  
 <Setting name="Microsoft.WindowsAzure.Plugins.RemoteAccess.AccountEncryptedPassword" value="..." />  
 <Setting name="Microsoft.WindowsAzure.Plugins.RemoteAccess.AccountExpiration" value="..." />  
 <Setting name="Microsoft.WindowsAzure.Plugins.RemoteForwarder.Enabled" value="true" />  
 </ConfigurationSettings>  
 <Certificates>  
 <Certificate name="Microsoft.WindowsAzure.Plugins.RemoteAccess.PasswordEncryption"   
 thumbprint="XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"   
 thumbprintAlgorithm="sha1" />  
 </Certificates>  
 </Role>  
</ServiceConfiguration>

To configure diagnostics configuration and in particular ETW Logs for the worker role, you can proceed as follows: right click on the role and selec **Properties**. On the **Configuration** property page make sure **Enable Diagnostics** is checked and click the **Configure…** button. On the **Diagnostics configuration** dialog go to the **ETW Logs** tab and select **Enable transfer of ETW logs**. Add the appropriate event sources to transfer by specifying the event source name and clicking on the **Add Event Source** button. In the sample, make sure to select the **TraceEventSource** as shown in the picture below.



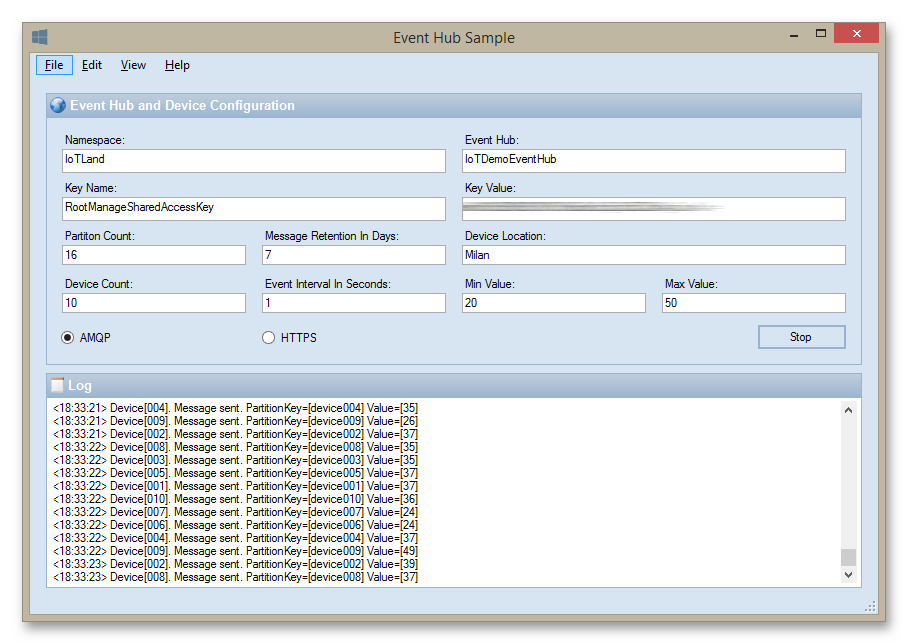
Then, right click the new row and select **Configure event storage...** menu item. In the **Storage Configuration** dialog, specify the name of the default storage table and, optionally, specify the name of the target table for each EventId defined in the EventSource class. In this sample, all diagnostic messages generated by the **TraceEventSource** class are configured to be traced to the **WADDiagnosticTable** (the prefix WAD is automatically added by Windows Azure Diagnostics), while the logs generated by the EventProcessor class are stored in a separate table called **WADEventProcessorTable**.



Once added, configure additional properties like the storage location for the logs, the log level, any keyword filters and transfer frequency.

## Sender

This application can be used to provision the event hub used by the application and simulate a configurable amount of devices.



The following table shows the configuration file of the application. Make sure to substitute the placeholders with the expected information before running the application.

<?xml version="1.0" encoding="utf-8"?>  
<configuration>  
 <appSettings>  
 <add key="namespace" value="[SERVICE BUS NAMESPACE]"/>  
 <add key="keyName" value="[NAMESPACE LEVEL SAS KEY NAME]"/>  
 <add key="keyValue" value="[NAMESPACE LEVEL SAS KEY VALUE]"/>  
 <add key="eventHub" value="[EVENT HUB NAME]"/>  
 <add key="partitionCount" value="16"/>  
 <add key="retentionDays" value="7"/>  
 <add key="location" value="Milan"/>  
 <add key="deviceCount" value="10"/>  
 <add key="eventInterval" value="1"/>  
 <add key="minValue" value="20"/>  
 <add key="maxValue" value="50"/>  
 </appSettings>  
 <system.serviceModel>  
 <extensions>  
 <!-- In this extension section we are introducing all known service bus extensions. User can remove the ones they don't need. -->  
 <behaviorExtensions>  
 <add name="connectionStatusBehavior"  
 type="Microsoft.ServiceBus.Configuration.ConnectionStatusElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="transportClientEndpointBehavior"  
 type="Microsoft.ServiceBus.Configuration.TransportClientEndpointBehaviorElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="serviceRegistrySettings"  
 type="Microsoft.ServiceBus.Configuration.ServiceRegistrySettingsElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 </behaviorExtensions>  
 <bindingElementExtensions>  
 <add name="netMessagingTransport"  
 type="Microsoft.ServiceBus.Messaging.Configuration.NetMessagingTransportExtensionElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="tcpRelayTransport"  
 type="Microsoft.ServiceBus.Configuration.TcpRelayTransportElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="httpRelayTransport"  
 type="Microsoft.ServiceBus.Configuration.HttpRelayTransportElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="httpsRelayTransport"  
 type="Microsoft.ServiceBus.Configuration.HttpsRelayTransportElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="onewayRelayTransport"  
 type="Microsoft.ServiceBus.Configuration.RelayedOnewayTransportElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 </bindingElementExtensions>  
 <bindingExtensions>  
 <add name="basicHttpRelayBinding"  
 type="Microsoft.ServiceBus.Configuration.BasicHttpRelayBindingCollectionElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="webHttpRelayBinding"  
 type="Microsoft.ServiceBus.Configuration.WebHttpRelayBindingCollectionElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="ws2007HttpRelayBinding"  
 type="Microsoft.ServiceBus.Configuration.WS2007HttpRelayBindingCollectionElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="netTcpRelayBinding"  
 type="Microsoft.ServiceBus.Configuration.NetTcpRelayBindingCollectionElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="netOnewayRelayBinding"  
 type="Microsoft.ServiceBus.Configuration.NetOnewayRelayBindingCollectionElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="netEventRelayBinding"  
 type="Microsoft.ServiceBus.Configuration.NetEventRelayBindingCollectionElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 <add name="netMessagingBinding"  
 type="Microsoft.ServiceBus.Messaging.Configuration.NetMessagingBindingCollectionElement, Microsoft.ServiceBus, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>  
 </bindingExtensions>  
 </extensions>  
 </system.serviceModel>  
</configuration>

The following table contains the code of the **MainForm** class. Spend a few minutes to analyze the code of the **btnStart\_Click** method. This method check creates the event hub if it doesn't exist and creates the **SendKey** used to create SAS tokens for individual devices. Then the code creates a separate Task for each device. Each Task start sending events using the selected transport (**AMQP** or **HTTPS**).

#region Using Directives  
using System;  
using System.Configuration;  
using System.Drawing;  
using System.IO;  
using System.Linq;  
using System.Globalization;  
using System.Net.Http;  
using System.Security.Cryptography;  
using System.Text;  
using System.Threading;  
using System.Threading.Tasks;  
using System.Web;  
using System.Windows.Forms;  
using Microsoft.ServiceBus;  
using Microsoft.ServiceBus.Messaging;  
using Newtonsoft.Json;  
using Newtonsoft.Json.Linq;  
using Microsoft.AzureCat.Samples.Entities;  
#endregion  
  
namespace Microsoft.AzureCat.Samples.DeviceSimulator  
{  
 public partial class MainForm : Form  
 {  
 #region Private Constants  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 // Formats  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 private const string DateFormat = "<{0,2:00}:{1,2:00}:{2,2:00}> {3}";  
 private const string ExceptionFormat = "Exception: {0}";  
 private const string InnerExceptionFormat = "InnerException: {0}";  
 private const string LogFileNameFormat = "WADTablesCleaner {0}.txt";  
  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 // Constants  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 private const string SaveAsTitle = "Save Log As";  
 private const string SaveAsExtension = "txt";  
 private const string SaveAsFilter = "Text Documents (\*.txt)|\*.txt";  
 private const string Start = "Start";  
 private const string Stop = "Stop";  
 private const string SenderSharedAccessKey = "SenderSharedAccessKey";  
 private const string DeviceId = "id";  
 private const string DeviceName = "name";  
 private const string DeviceLocation = "location";  
 private const string Value = "value";  
  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 // Configuration Parameters  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 private const string NamespaceParameter = "namespace";  
 private const string KeyNameParameter = "keyName";  
 private const string KeyValueParameter = "keyValue";  
 private const string EventHubParameter = "eventHub";  
 private const string LocationParameter = "location";  
 private const string PartitionCountParameter = "partitionCount";  
 private const string RetentionDaysParameter = "retentionDays";  
 private const string DeviceCountParameter = "deviceCount";  
 private const string EventIntervalParameter = "eventInterval";  
 private const string MinValueParameter = "minValue";  
 private const string MaxValueParameter = "maxValue";  
 private const string ApiVersion = "&api-version=2014-05";  
  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 // Configuration Parameters  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 private const string DefaultEventHubName = "SampleEventHub";  
 private const int DefaultDeviceNumber = 10;  
 private const int DefaultMinValue = 20;  
 private const int DefaultMaxValue = 50;  
 private const int DefaultEventIntervalInSeconds = 1;  
  
  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 // Messages  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 private const string NamespaceCannonBeNull = "The Service Bus namespace cannot be null.";  
 private const string EventHubNameCannonBeNull = "The event hub name cannot be null.";  
 private const string KeyNameCannonBeNull = "The senderKey name cannot be null.";  
 private const string KeyValueCannonBeNull = "The senderKey value cannot be null.";  
 private const string EventHubCreatedOrRetrieved = "Event hub [{0}] created or retrieved.";  
 private const string MessagingFactoryCreated = "Device[{0,3:000}]. MessagingFactory created.";  
 private const string SasToken = "Device[{0,3:000}]. SAS Token created.";  
 private const string EventHubClientCreated = "Device[{0,3:000}]. EventHubClient created: Path=[{1}].";  
 private const string HttpClientCreated = "Device[{0,3:000}]. HttpClient created: BaseAddress=[{1}].";  
 private const string EventSent = "Device[{0,3:000}]. Message sent. PartitionKey=[{1}] Value=[{2}]";  
 private const string SendFailed = "Device[{0,3:000}]. Message send failed: [{1}]";  
 #endregion  
  
 #region Private Fields  
 private CancellationTokenSource cancellationTokenSource;  
 private int eventId;  
 #endregion  
  
 #region Public Constructor  
 /// <summary>  
 /// Initializes a new instance of the MainForm class.  
 /// </summary>  
 public MainForm()  
 {  
 InitializeComponent();  
 ConfigureComponent();  
 ReadConfiguration();  
 }  
 #endregion  
  
 #region Public Methods  
  
 public void ConfigureComponent()  
 {  
 txtNamespace.AutoSize = false;  
 txtNamespace.Size = new Size(txtNamespace.Size.Width, 24);  
 txtKeyName.AutoSize = false;  
 txtKeyName.Size = new Size(txtKeyName.Size.Width, 24);  
 txtKeyValue.AutoSize = false;  
 txtKeyValue.Size = new Size(txtKeyValue.Size.Width, 24);  
 txtEventHub.AutoSize = false;  
 txtEventHub.Size = new Size(txtEventHub.Size.Width, 24);  
 }  
  
 public void HandleException(Exception ex)  
 {  
 if (ex == null || string.IsNullOrEmpty(ex.Message))  
 {  
 return;  
 }  
 WriteToLog(string.Format(CultureInfo.CurrentCulture, ExceptionFormat, ex.Message));  
 if (ex.InnerException != null && !string.IsNullOrEmpty(ex.InnerException.Message))  
 {  
 WriteToLog(string.Format(CultureInfo.CurrentCulture, InnerExceptionFormat, ex.InnerException.Message));  
 }  
 }  
 #endregion  
  
 #region Private Methods  
 public static bool IsJson(string item)  
 {  
 if (item == null)  
 {  
 throw new ArgumentException("The item argument cannot be null.");  
 }  
 try  
 {  
 var obj = JToken.Parse(item);  
 return obj != null;  
 }  
 catch (Exception)  
 {  
 return false;  
 }  
 }  
  
 public string IndentJson(string json)  
 {  
 if (string.IsNullOrWhiteSpace(json))  
 {  
 return null;  
 }  
 dynamic parsedJson = JsonConvert.DeserializeObject(json);  
 return JsonConvert.SerializeObject(parsedJson, Formatting.Indented);  
 }  
  
 private void ReadConfiguration()  
 {  
 try  
 {  
 txtNamespace.Text = ConfigurationManager.AppSettings[NamespaceParameter];  
 txtKeyName.Text = ConfigurationManager.AppSettings[KeyNameParameter];  
 txtKeyValue.Text = ConfigurationManager.AppSettings[KeyValueParameter];  
 txtEventHub.Text = ConfigurationManager.AppSettings[EventHubParameter] ?? DefaultEventHubName;  
 var eventHubDescription = new EventHubDescription(txtEventHub.Text);  
 int value;  
 var setting = ConfigurationManager.AppSettings[PartitionCountParameter];  
 txtPartitionCount.Text = int.TryParse(setting, out value) ?  
 value.ToString(CultureInfo.InvariantCulture) :  
 eventHubDescription.PartitionCount.ToString(CultureInfo.InvariantCulture);  
 setting = ConfigurationManager.AppSettings[RetentionDaysParameter];  
 txtMessageRetentionInDays.Text = int.TryParse(setting, out value) ?  
 value.ToString(CultureInfo.InvariantCulture) :  
 eventHubDescription.MessageRetentionInDays.ToString(CultureInfo.InvariantCulture);  
 txtLocation.Text = ConfigurationManager.AppSettings[LocationParameter];  
 setting = ConfigurationManager.AppSettings[DeviceCountParameter];  
 txtDeviceCount.Text = int.TryParse(setting, out value) ?   
 value.ToString(CultureInfo.InvariantCulture) :   
 DefaultDeviceNumber.ToString(CultureInfo.InvariantCulture);  
 setting = ConfigurationManager.AppSettings[EventIntervalParameter];  
 txtEventIntervalInSeconds.Text = int.TryParse(setting, out value) ?  
 value.ToString(CultureInfo.InvariantCulture) :  
 DefaultEventIntervalInSeconds.ToString(CultureInfo.InvariantCulture);  
 setting = ConfigurationManager.AppSettings[MinValueParameter];  
 txtMinValue.Text = int.TryParse(setting, out value) ?  
 value.ToString(CultureInfo.InvariantCulture) :  
 DefaultMinValue.ToString(CultureInfo.InvariantCulture);  
 setting = ConfigurationManager.AppSettings[MaxValueParameter];  
 txtMaxValue.Text = int.TryParse(setting, out value) ?  
 value.ToString(CultureInfo.InvariantCulture) :  
 DefaultMaxValue.ToString(CultureInfo.InvariantCulture);  
 }  
 catch (Exception ex)  
 {  
 HandleException(ex);  
 }  
 }  
  
 private void WriteToLog(string message)  
 {  
 if (InvokeRequired)  
 {  
 Invoke(new Action<string>(InternalWriteToLog), new object[] { message });  
 }  
 else  
 {  
 InternalWriteToLog(message);  
 }  
 }  
  
 private void InternalWriteToLog(string message)  
 {  
 lock (this)  
 {  
 if (string.IsNullOrEmpty(message))  
 {  
 return;  
 }  
 var lines = message.Split('\n');  
 var now = DateTime.Now;  
 var space = new string(' ', 19);  
  
 for (var i = 0; i < lines.Length; i++)  
 {  
 if (i == 0)  
 {  
 var line = string.Format(DateFormat,  
 now.Hour,  
 now.Minute,  
 now.Second,  
 lines[i]);  
 lstLog.Items.Add(line);  
 }  
 else  
 {  
 lstLog.Items.Add(space + lines[i]);  
 }  
 }  
 lstLog.SelectedIndex = lstLog.Items.Count - 1;  
 lstLog.SelectedIndex = -1;  
 }  
 }  
  
 #endregion  
  
 #region Event Handlers  
  
 private void exitToolStripMenuItem\_Click(object sender, EventArgs e)  
 {  
 Close();  
 }  
  
 private void clearLogToolStripMenuItem\_Click(object sender, EventArgs e)  
 {  
 lstLog.Items.Clear();  
 }  
  
 /// <summary>  
 /// Saves the log to a text file  
 /// </summary>  
 /// <param name="sender">MainForm object</param>  
 /// <param name="e">System.EventArgs parameter</param>  
 private void saveLogToolStripMenuItem\_Click(object sender, EventArgs e)  
 {  
 try  
 {  
 if (lstLog.Items.Count <= 0)  
 {  
 return;  
 }  
 saveFileDialog.Title = SaveAsTitle;  
 saveFileDialog.DefaultExt = SaveAsExtension;  
 saveFileDialog.Filter = SaveAsFilter;  
 saveFileDialog.FileName = string.Format(LogFileNameFormat, DateTime.Now.ToString(CultureInfo.CurrentUICulture).Replace('/', '-').Replace(':', '-'));  
 if (saveFileDialog.ShowDialog() != DialogResult.OK ||   
 string.IsNullOrEmpty(saveFileDialog.FileName))  
 {  
 return;  
 }  
 using (var writer = new StreamWriter(saveFileDialog.FileName))  
 {  
 foreach (var t in lstLog.Items)  
 {  
 writer.WriteLine(t as string);  
 }  
 }  
 }  
 catch (Exception ex)  
 {  
 HandleException(ex);  
 }  
 }  
  
 private void logWindowToolStripMenuItem\_Click(object sender, EventArgs e)  
 {  
 splitContainer.Panel2Collapsed = !((ToolStripMenuItem)sender).Checked;  
 }  
  
 private void aboutToolStripMenuItem\_Click(object sender, EventArgs e)  
 {  
 var form = new AboutForm();  
 form.ShowDialog();  
 }  
  
 private void lstLog\_Leave(object sender, EventArgs e)  
 {  
 lstLog.SelectedIndex = -1;  
 }  
  
 private void button\_MouseEnter(object sender, EventArgs e)  
 {  
 var control = sender as Control;  
 if (control != null)  
 {  
 control.ForeColor = Color.White;  
 }  
 }  
  
 private void button\_MouseLeave(object sender, EventArgs e)  
 {  
 var control = sender as Control;  
 if (control != null)  
 {  
 control.ForeColor = SystemColors.ControlText;  
 }  
 }  
   
 private void MainForm\_Paint(object sender, PaintEventArgs e)  
 {  
 var width = (mainHeaderPanel.Size.Width - 48)/2;  
 var halfWidth = (width - 16)/2;  
  
 txtNamespace.Size = new Size(width, txtNamespace.Size.Height);  
 txtKeyName.Size = new Size(width, txtKeyName.Size.Height);  
 txtKeyValue.Size = new Size(width, txtKeyValue.Size.Height);  
 txtEventHub.Size = new Size(width, txtEventHub.Size.Height);  
 txtLocation.Size = new Size(width, txtLocation.Size.Height);  
 txtPartitionCount.Size = new Size(halfWidth, txtPartitionCount.Size.Height);  
 txtMessageRetentionInDays.Size = new Size(halfWidth, txtMessageRetentionInDays.Size.Height);  
 txtDeviceCount.Size = new Size(halfWidth, txtDeviceCount.Size.Height);  
 txtEventIntervalInSeconds.Size = new Size(halfWidth, txtEventIntervalInSeconds.Size.Height);  
 txtMinValue.Size = new Size(halfWidth, txtMinValue.Size.Height);  
 txtMinValue.Size = new Size(halfWidth, txtMinValue.Size.Height);  
  
 txtEventHub.Location = new Point(32 + width, txtEventHub.Location.Y);  
 txtKeyValue.Location = new Point(32 + width, txtKeyValue.Location.Y);  
 txtLocation.Location = new Point(32 + width, txtLocation.Location.Y);  
 txtMessageRetentionInDays.Location = new Point(32 + halfWidth, txtMessageRetentionInDays.Location.Y);  
 txtEventIntervalInSeconds.Location = new Point(32 + halfWidth, txtEventIntervalInSeconds.Location.Y);  
 txtMinValue.Location = new Point(32 + width, txtMinValue.Location.Y);  
 txtMaxValue.Location = new Point(48 + width + halfWidth, txtMaxValue.Location.Y);  
  
 lblEventHub.Location = new Point(32 + width, lblEventHub.Location.Y);  
 lblKeyValue.Location = new Point(32 + width, lblKeyValue.Location.Y);  
 lblLocation.Location = new Point(32 + width, lblLocation.Location.Y);  
 lblMessageRetentionInDays.Location = new Point(32 + halfWidth, lblMessageRetentionInDays.Location.Y);  
 lblEventIntervalInSeconds.Location = new Point(32 + halfWidth, lblEventIntervalInSeconds.Location.Y);  
 lblMinValue.Location = new Point(32 + width, lblMinValue.Location.Y);  
 lblMaxValue.Location = new Point(48 + width + halfWidth, lblMaxValue.Location.Y);  
 radioButtonHttps.Location = new Point(32 + halfWidth, radioButtonAmqp.Location.Y);  
 }  
  
 private void MainForm\_Shown(object sender, EventArgs e)  
 {  
 txtNamespace.SelectionLength = 0;  
 }  
  
 private async void btnStart\_Click(object sender, EventArgs e)  
 {  
 try  
 {  
 if (string.Compare(btnStart.Text, Start, StringComparison.OrdinalIgnoreCase) == 0)  
 {  
 // Change button text  
 btnStart.Text = Stop;  
  
 // Validate parameters  
 if (!ValidateParameters())  
 {  
 return;  
 }  
  
 // Create namespace manager  
 var namespaceUri = ServiceBusEnvironment.CreateServiceUri("sb", txtNamespace.Text, string.Empty);  
 var tokenProvider = TokenProvider.CreateSharedAccessSignatureTokenProvider(txtKeyName.Text, txtKeyValue.Text);  
 var namespaceManager = new NamespaceManager(namespaceUri, tokenProvider);  
  
 // Check if the event hub already exists, if not, create the event hub.  
 var eventHubDescription = await namespaceManager.EventHubExistsAsync(txtEventHub.Text) ?  
 await namespaceManager.GetEventHubAsync(txtEventHub.Text) :  
 await namespaceManager.CreateEventHubAsync(new EventHubDescription(txtEventHub.Text)  
 {  
 PartitionCount = txtPartitionCount.IntegerValue,  
 MessageRetentionInDays = txtMessageRetentionInDays.IntegerValue  
 });  
 WriteToLog(string.Format(EventHubCreatedOrRetrieved, txtEventHub.Text));  
  
 // Check if the SAS authorization rule used by devices to send events to the event hub already exists, if not, create the rule.  
 var authorizationRule = eventHubDescription.  
 Authorization.  
 FirstOrDefault(r => string.Compare(r.KeyName,   
 SenderSharedAccessKey,   
 StringComparison.InvariantCultureIgnoreCase)   
 == 0) as SharedAccessAuthorizationRule;  
 if (authorizationRule == null)  
 {  
 authorizationRule = new SharedAccessAuthorizationRule(SenderSharedAccessKey,   
 SharedAccessAuthorizationRule.GenerateRandomKey(),   
 new[]  
 {  
 AccessRights.Send  
 });  
 eventHubDescription.Authorization.Add(authorizationRule);  
 await namespaceManager.UpdateEventHubAsync(eventHubDescription);  
 }  
   
 cancellationTokenSource = new CancellationTokenSource();  
 var serviceBusNamespace = txtNamespace.Text;  
 var eventHubName = txtEventHub.Text;  
 var senderKey = authorizationRule.PrimaryKey;  
 var location = txtLocation.Text;  
 var eventInterval = txtEventIntervalInSeconds.IntegerValue \* 1000;  
 var minValue = txtMinValue.IntegerValue;  
 var maxValue = txtMaxValue.IntegerValue;  
 var cancellationToken = cancellationTokenSource.Token;  
  
 // Create one task for each device  
 for (var i = 1; i <= txtDeviceCount.IntegerValue; i++)  
 {  
 var deviceId = i;  
 #pragma warning disable 4014  
 #pragma warning disable 4014  
 Task.Run(async () =>  
 #pragma warning restore 4014  
 {  
 var deviceName = $"device{deviceId:000}";  
 var random = new Random((int)DateTime.Now.Ticks);  
  
 if (radioButtonAmqp.Checked)  
 {  
 // The token has the following format:   
 // SharedAccessSignature sr={URI}&sig={HMAC\_SHA256\_SIGNATURE}&se={EXPIRATION\_TIME}&skn={KEY\_NAME}  
 var token = CreateSasTokenForAmqpSender(SenderSharedAccessKey,  
 senderKey,  
 serviceBusNamespace,  
 eventHubName,  
 deviceName,  
 TimeSpan.FromDays(1));  
 WriteToLog(string.Format(SasToken, deviceId));  
  
 var messagingFactory = MessagingFactory.Create(ServiceBusEnvironment.CreateServiceUri("sb", serviceBusNamespace, ""), new MessagingFactorySettings  
 {  
 TokenProvider = TokenProvider.CreateSharedAccessSignatureTokenProvider(token),  
 TransportType = TransportType.Amqp  
 });  
 WriteToLog(string.Format(MessagingFactoryCreated, deviceId));  
  
 // Each device uses a different publisher endpoint: [EventHub]/publishers/[PublisherName]  
 var eventHubClient = messagingFactory.CreateEventHubClient($"{eventHubName}/publishers/{deviceName}");  
 WriteToLog(string.Format(EventHubClientCreated, deviceId, eventHubClient.Path));  
  
 while (!cancellationToken.IsCancellationRequested)  
 {  
 // Create random value  
 var value = random.Next(minValue, maxValue + 1);  
  
 // Create EventData object with the payload serialized in JSON format   
 using (var eventData = new EventData(Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(new Payload  
 {  
 EventId = eventId++,  
 DeviceId = deviceId,  
 Value = value,  
 Timestamp = DateTime.UtcNow  
 })))  
 {  
 PartitionKey = deviceName  
 })  
 {  
 // Create custom properties  
 eventData.Properties.Add(DeviceId, deviceId);  
 eventData.Properties.Add(DeviceName, deviceName);  
 eventData.Properties.Add(DeviceLocation, location);  
 eventData.Properties.Add(Value, value);  
  
 // Send the event to the event hub  
 await eventHubClient.SendAsync(eventData);  
 WriteToLog(string.Format(EventSent, deviceId, deviceName, value));  
 }  
  
 // Wait for the event time interval  
 Thread.Sleep(eventInterval);  
 }  
 }  
 else  
 {  
 // The token has the following format:   
 // SharedAccessSignature sr={URI}&sig={HMAC\_SHA256\_SIGNATURE}&se={EXPIRATION\_TIME}&skn={KEY\_NAME}  
 var token = CreateSasTokenForHttpsSender(SenderSharedAccessKey,  
 senderKey,  
 serviceBusNamespace,  
 eventHubName,  
 deviceName,  
 TimeSpan.FromDays(1));  
 WriteToLog(string.Format(SasToken, deviceId));  
  
 // Create HttpClient object used to send events to the event hub.  
 var httpClient = new HttpClient  
 {  
 BaseAddress = new Uri($"https://{serviceBusNamespace}.servicebus.windows.net/{eventHubName}/publishers/{deviceName}".ToLower())  
 };  
 httpClient.DefaultRequestHeaders.TryAddWithoutValidation("Authorization", token);  
 httpClient.DefaultRequestHeaders.Add("ContentType", "application/json;type=entry;charset=utf-8");  
 WriteToLog(string.Format(HttpClientCreated, deviceId, httpClient.BaseAddress));  
  
 while (!cancellationToken.IsCancellationRequested)  
 {  
 // Create random value  
 var value = random.Next(minValue, maxValue + 1);  
  
 // Create HttpContent  
 var postContent = new ByteArrayContent(Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(new Payload  
 {  
 EventId = eventId++,  
 DeviceId = deviceId,  
 Value = value,  
 Timestamp = DateTime.UtcNow  
 })));  
  
 // Create custom properties  
   
 postContent.Headers.Add(DeviceId, deviceId.ToString(CultureInfo.InvariantCulture));  
 postContent.Headers.Add(DeviceName, deviceName);  
 //postContent.Headers.Add(DeviceLocation, location);  
 postContent.Headers.Add(Value, value.ToString(CultureInfo.InvariantCulture));  
  
 try  
 {  
 var response = await httpClient.PostAsync(httpClient.BaseAddress + "/messages" + "?timeout=60" + ApiVersion, postContent, cancellationToken);  
 response.EnsureSuccessStatusCode();  
 WriteToLog(string.Format(EventSent, deviceId, deviceName, value));  
 }  
 catch (HttpRequestException ex)  
 {  
 WriteToLog(string.Format(SendFailed, deviceId, ex.Message));  
 }  
 }  
 }  
 },  
 cancellationToken).ContinueWith(t =>  
#pragma warning restore 4014  
 #pragma warning restore 4014  
 {  
 if (t.IsFaulted && t.Exception != null)  
 {  
 HandleException(t.Exception);  
 }  
 }, cancellationToken);  
 }  
  
 }  
 else  
 {  
 // Change button text  
 btnStart.Text = Start;  
 cancellationTokenSource?.Cancel();  
 }  
 }  
 catch (Exception ex)  
 {  
 HandleException(ex);  
 }  
 }  
  
 private bool ValidateParameters()  
 {  
 if (string.IsNullOrWhiteSpace(txtNamespace.Text))  
 {  
 WriteToLog(NamespaceCannonBeNull);  
 return false;  
 }  
 if (string.IsNullOrWhiteSpace(txtEventHub.Text))  
 {  
 WriteToLog(EventHubNameCannonBeNull);  
 return false;  
 }  
 if (string.IsNullOrWhiteSpace(txtKeyName.Text))  
 {  
 WriteToLog(KeyNameCannonBeNull);  
 return false;  
 }  
 if (string.IsNullOrWhiteSpace(txtKeyValue.Text))  
 {  
 WriteToLog(KeyValueCannonBeNull);  
 return false;  
 }  
 return true;  
 }  
  
 public static string CreateSasTokenForAmqpSender(string senderKeyName,   
 string senderKey,   
 string serviceNamespace,   
 string hubName,   
 string publisherName,   
 TimeSpan tokenTimeToLive)  
 {  
 // This is the format of the publisher endpoint. Each device uses a different publisher endpoint.  
 // sb://<NAMESPACE>.servicebus.windows.net/<EVENT\_HUB\_NAME>/publishers/<PUBLISHER\_NAME>.   
 var serviceUri = ServiceBusEnvironment.CreateServiceUri("sb",   
 serviceNamespace,  
 $"{hubName}/publishers/{publisherName}")  
 .ToString()  
 .Trim('/');  
 // SharedAccessSignature sr=<URL-encoded-resourceURI>&sig=<URL-encoded-signature-string>&se=<expiry-time-in-ISO-8061-format. >&skn=<senderKeyName>  
 return SharedAccessSignatureTokenProvider.GetSharedAccessSignature(senderKeyName, senderKey, serviceUri, tokenTimeToLive);  
 }  
  
 // Create a SAS token for a specified scope. SAS tokens are described in http://msdn.microsoft.com/en-us/library/windowsazure/dn170477.aspx.  
 private static string CreateSasTokenForHttpsSender(string senderKeyName,   
 string senderKey,   
 string serviceNamespace,   
 string hubName,   
 string publisherName,   
 TimeSpan tokenTimeToLive)  
 {  
 // Set token lifetime. When supplying a device with a token, you might want to use a longer expiration time.  
 var origin = new DateTime(1970, 1, 1, 0, 0, 0, 0);  
 var difference = DateTime.Now.ToUniversalTime() - origin;  
 var tokenExpirationTime = Convert.ToUInt32(difference.TotalSeconds) + tokenTimeToLive.Seconds;  
  
 // https://<NAMESPACE>.servicebus.windows.net/<EVENT\_HUB\_NAME>/publishers/<PUBLISHER\_NAME>.   
 var uri = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespace,  
 $"{hubName}/publishers/{publisherName}")  
 .ToString()  
 .Trim('/');  
 var stringToSign = HttpUtility.UrlEncode(uri) + "\n" + tokenExpirationTime;  
 var hmac = new HMACSHA256(Encoding.UTF8.GetBytes(senderKey));  
  
 var signature = Convert.ToBase64String(hmac.ComputeHash(Encoding.UTF8.GetBytes(stringToSign)));  
  
 // SharedAccessSignature sr=<URL-encoded-resourceURI>&sig=<URL-encoded-signature-string>&se=<expiry-time-in-ISO-8061-format. >&skn=<senderKeyName>  
 var token = String.Format(CultureInfo.InvariantCulture, "SharedAccessSignature sr={0}&sig={1}&se={2}&skn={3}",  
 HttpUtility.UrlEncode(uri), HttpUtility.UrlEncode(signature), tokenExpirationTime, senderKeyName);  
 return token;  
 }  
  
 private void btnClear\_Click(object sender, EventArgs e)  
 {  
 lstLog.Items.Clear();  
 }  
 #endregion  
 }  
}