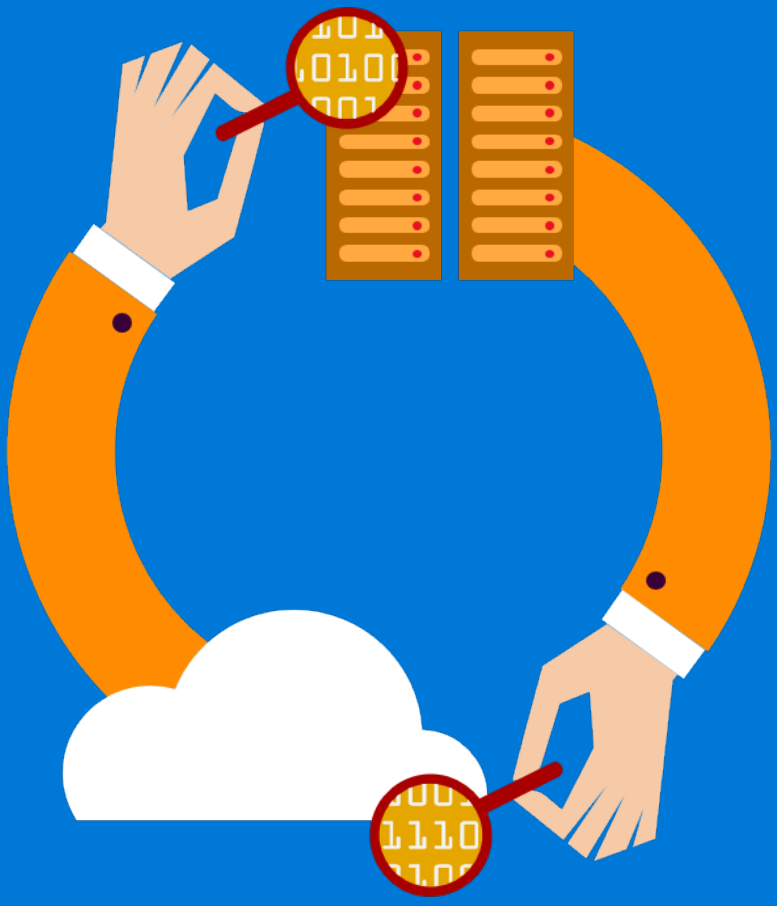
Microsoft



# Create event-driven serverless apps with Azure Event Grid and Azure Functions

By Paolo Salvatori
Azure Global Customer Engineering

January 2020

# Contents

Overview ..... 3
How the solution works ..... 3
Message flow ..... 5
GitHub contents ..... 5
Blob trigger versus Azure Event Grid trigger ..... 6
Sample configuration ..... 6
Azure Function code ..... 7
Bash scripts ..... 12
Alternative: Functions as event handlers ..... 24
Upload files to the container ..... 26
View messages from the queue in Service Bus Explorer ..... 27
Query metrics in Application Insights ..... 28
Learn more ..... 29

## List of figures

Figure 1. When a file is uploaded as a blog, the storage account topic sends an event to the event grid subscribers, including Azure Function ..... 4
Figure 2. You can upload files to the solution using Storage Explorer. ..... 26
Figure 3. You can use Service Bus Explorer to view messages sent by the function to the queue in the target Service Bus namespace. ..... 27
Figure 4. You can use Kusto Query Language to query metrics in Application Insights. ..... 28

Authored by Paolo Salvatori. Edited by Nanette Ray, Resources Online. Reviewed by the Azure Event Grid team.

# Overview

You can build and run serverless apps at scale using Azure managed services that handle the infrastructure for you. A team of AzureCAT advisors recently adapted this approach for engineers working on an event-driven, big data project, and their solution is presented here. This solution was originally designed to process data captured during tests of a self-driving car, but it can be used to handle events generated by many types of Azure services.

The key is to use Azure Event Grid, a fully managed intelligent event routing service that makes it easy to build event-driven solutions. It uses a publish and subscribe model, which means that Azure resources, such as storage, can publish events to an event grid. In turn, the event grid pushes those events to subscribers, which can then react.

A GitHub sample is included. It demonstrates how to create a serverless application using Azure Functions to receive and process events any time a blob is created, updated, or deleted in a given container inside an Azure storage account. You can generalize this approach to handle any event generated by an Azure Event Grid event producer, including Azure subscriptions, resource groups, and Azure Event Hubs. Many event sources and event handlers can be used to build an event-driven serverless or microservice-based solution. For more information, see other code samples for Azure Event Grid.

Check out the GitHub sample!

### Event Grid concepts

**Events:** What happened. In this solution, the event occurs when a blob is created, updated, or deleted in a given container.

**Event sources:** Where the event took place—for example, Azure Storage.

**Topics:** The endpoint where publishers—for example, Azure Event Grid—send events.

**Event subscriptions:** The endpoint or built-in mechanism to route events, sometimes to more than one handler. Subscriptions are also used by handlers to intelligently filter incoming events.

**Event handlers:** The app or service reacting to the event—for example, Azure Functions.

For more information, see Concepts in Azure Event Grid.

# How the solution works

The engineers for the self-driving car project designed an app that was installed on a test car. It captured data from the vehicle's GPS, gyroscope, and camera. At the end of a test drive, the data was sent to the cloud to be processed by a big data solution on Azure. This solution composed the individual images from the vehicle's camera into frames that were rendered in a video. That, along with the associated GPS and gyroscope data, was used to analyze the test car's behavior during a drive.

As Figure 1 shows, the solution on Azure triggers the execution of a blob processing solution as soon as new files are written to a given storage account. Azure Function sends a message containing the data of the new blob to an Azure Service Bus queue to trigger the execution of a hypothetical big data solution. The events are also sent to Azure App Service Web Apps, where the messages are collected and displayed.

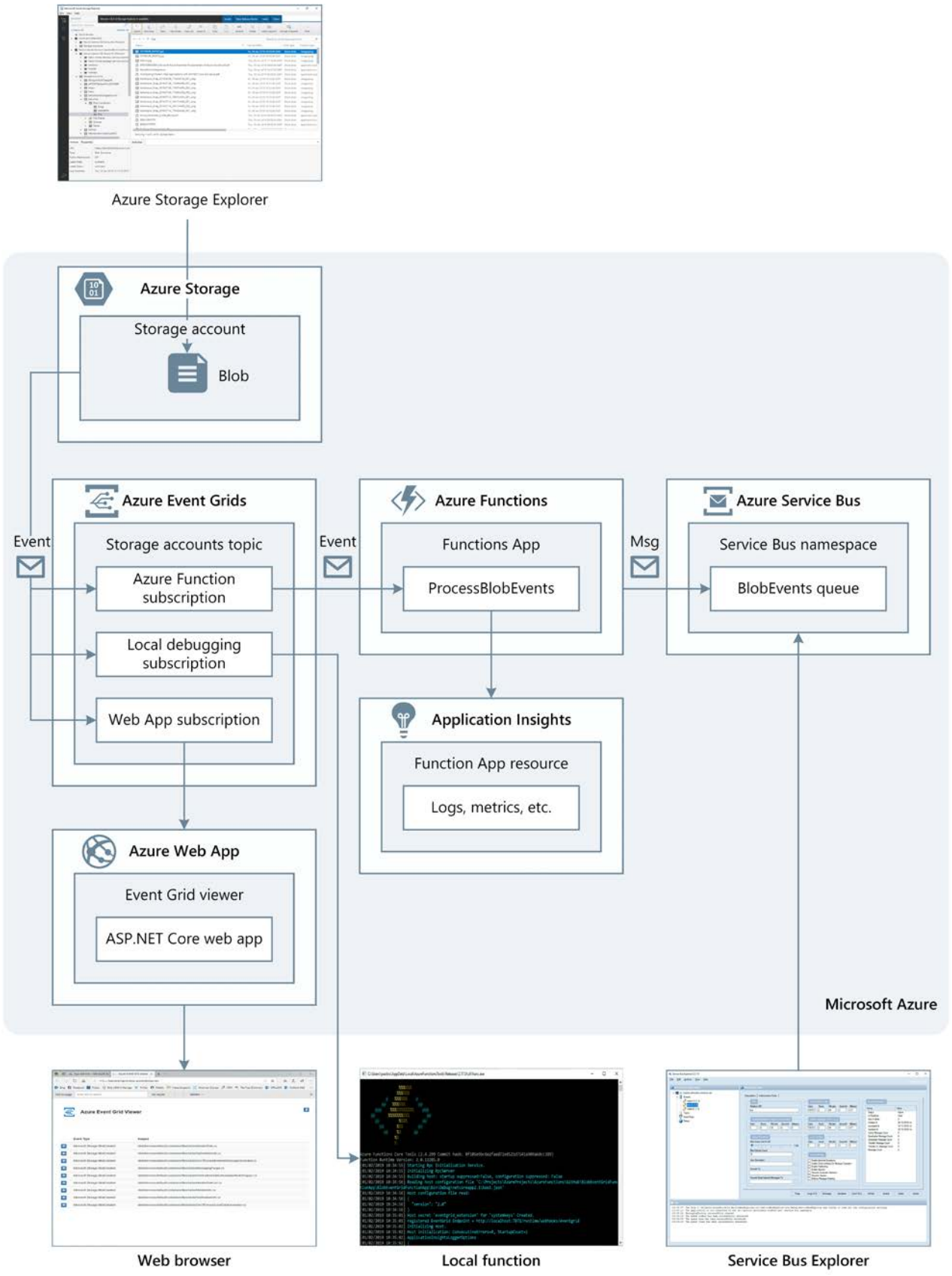



Figure 1. When a file is uploaded as a blog, the storage account topic sends an event to the event grid subscribers, including Azure Function.

This project uses Azure Functions to create small pieces of code, known as *functions*, to handle the events. This sample uses C#, but you can build functions using your programming language of choice, such as F#, Node.js, Java, or PHP. You pay only for the time your code runs, and you can trust Azure to scale your functions as needed.

This solution also tracks the telemetry produced by Azure Function and shows you how to use Kusto Query Language to query Azure Application Insights. You can then analyze the metrics and logs generated by the Azure Function.

## Message flow

The message flow begins when you use Azure Storage Explorer to upload one or more files as blobs to a container in an Azure Storage account. In this context, the Azure Storage Explorer is used to simulate an upstream application that periodically uploads files to a storage account to be processed by a big data solution. The file upload operation triggers an event that is pushed through Azure Event Grid to three event grid subscribers:

- **Azure Function subscription:** This subscription is configured to send events to the webhook endpoint exposed by a function that makes use of an Event Grid trigger. The function receives events from the Event Grid subscription and sends events to a queue in a Service Bus namespace. Azure Function is also instrumented to collect and send metrics and logs to Application Insights.
- **Local debugging subscription:** This subscription is configured to send events to a local function via a public HTTP/S endpoint exposed by ngrok on your development machine. For more information, see Azure Function Event Grid Trigger Local Debugging.
- **Web App subscription:** This subscription is configured to send events to an Azure web app that displays the event messages. For more information about how to use this web app to see the events generated by an event grid topic, see Azure Event Grid Viewer. For more information about sending storage events to a web endpoint with Azure CLI, see the QuickStart: Route storage events to web endpoint with Azure CLI.

ⓘ **NOTE:** Another option is to configure the functions as event handlers instead of pointing at the function's webhook endpoint. This option was not available at the time this solution was created for the customer, but it is discussed later in this guide.

## GitHub contents

The GitHub sample includes all the bash scripts required to provision the event grid subscriptions and topics using Azure CLI commands and shows you how to:

- Use bash scripts to create event grid subscriptions for Azure Functions, the Event Grid viewer web app, and the function app running on the developer virtual machine (VM). Endpoints are exposed using ngrok. For more information, see Azure Function Event Grid Trigger Local Debugging.
- Instrument the Azure Function to generate custom metrics.
- Use Application Insights to analyze Azure Function logs, metrics, and custom metrics.

The GitHub solution also demonstrates how to do local testing and debug an Azure Function that uses the event grid trigger on your local machine before publishing the application to Azure.

 Check out the GitHub sample!

## Blob trigger versus Azure Event Grid trigger

A function in Azure Function can use the Blob storage trigger or the Event Grid trigger to receive events generated by blob storage. By default, a Blob storage trigger is used to start a function when a new or updated blob is detected. The blob contents are provided as input to the function. When using this type of trigger, you can directly access the content of the new or updated blob.

An Event Grid trigger also supports blob events and can be used to start a function when a new or updated blob is detected. You should use this type of trigger instead of a blob storage trigger when your solution:

- Uses blob storage accounts, which are supported for blob input and output bindings but not for blob triggers. Blob storage triggers require a general-purpose storage account.
- Requires high scale, which can be loosely defined as containers that have more than 100,000 blobs in them or storage accounts that have more than 100 blob updates per second.
- Needs to minimize latency. If your function app is on the Azure Consumption plan, and the function goes idle, there can be up to a 10-minute delay in processing new blobs. To avoid this latency, you can switch to an App Service plan with **Always On** enabled. You can also use an Event Grid trigger with your blob storage account. For more information, see Azure Functions scale and hosting.
- Uses blob delete events, which cannot be handled with a blob storage trigger.

## Sample configuration

Make sure to specify the following data in the **local.settings.json** file:

- AzureWebJobStorage. Use this value so that the Azure Functions runtime uses this storage account connection string for all functions except for HTTP triggered functions.
- APPINSIGHTS_INSTRUMENTATIONKEY. If you're using Application Insights, use this instrumentation key. For more information, see Monitor Azure Functions.
- ServiceBusConnectionString. Use this variable to specify the connection string of the Service Bus namespace containing the queue to which Azure Function sends messages.
- QueueName. Use this variable to specify the names of the queue.

```
{
      "IsEncrypted": false,
      "Values": {
            "AzureWebJobsStorage": "<storage-account-connection-string>",
            "FUNCTIONS_WORKER_RUNTIME": "dotnet",
            "APPINSIGHTS_INSTRUMENTATIONKEY": "<app-insights-instrumentation-
key>",
            "ServiceBusConnectionString": "<service-bus-connection-string>",
            "QueueName": "<queue-name>"
```

```
        }
}
```

## Azure Function code

Here's the code of the Azure Function:

```
#region Using Directives
using System;
using System.Threading.Tasks;
using Microsoft.Azure.EventGrid.Models;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.ServiceBus;
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.Extensibility;
using Microsoft.Extensions.Logging;
using Microsoft.Azure.ServiceBus;
using System.Text;
using Microsoft.Azure.WebJobs.Extensions.EventGrid;
using Newtonsoft.Json.Linq;
using System.Collections.Generic;
#endregion

namespace BlobEventGridFunctionApp
{
    public static class ProcessBlobEvents
    {
        #region Private Constants
        private const string BlobCreatedEvent =
"Microsoft.Storage.BlobCreated";
        private const string BlobDeletedEvent =
"Microsoft.Storage.BlobDeleted";
        #endregion

        #region Private Static Fields
        private static readonly string key =
TelemetryConfiguration.Active.InstrumentationKey =
Environment.GetEnvironmentVariable("APPINSIGHTS_INSTRUMENTATIONKEY",
EnvironmentVariableTarget.Process);
        private static readonly TelemetryClient telemetry = new
TelemetryClient() { InstrumentationKey = key };
        #endregion

        #region Azure Functions
        [FunctionName("ProcessBlobEvents")]
        public static async Task Run([EventGridTrigger]EventGridEvent
eventGridEvent,
                                     [ServiceBus("%QueueName%", Connection =
"ServiceBusConnectionString", EntityType = EntityType.Queue)]
IAsyncCollector<Message> asyncCollector,
```

```
                                ExecutionContext context,
                                ILogger log)
        {
            try
            {
                if (eventGridEvent == null &&
string.IsNullOrWhiteSpace(eventGridEvent.EventType))
                {
                    throw new ArgumentNullException("Null or Invalid Event
Grid Event");
                }

                log.LogInformation($@"New Event Grid Event:
    - Id=[{eventGridEvent.Id}]
    - EventType=[{eventGridEvent.EventType}]
    - EventTime=[{eventGridEvent.EventTime}]
    - Subject=[{eventGridEvent.Subject}]
    - Topic=[{eventGridEvent.Topic}]");

                if (eventGridEvent.Data is JObject jObject)
                {
                    // Create message
                    var message = new
Message(Encoding.UTF8.GetBytes(jObject.ToString()))
                    {
                        MessageId = eventGridEvent.Id
                    };

                    switch (eventGridEvent.EventType)
                    {
                        case BlobCreatedEvent:
                            {
                                var blobCreatedEvent =
jObject.ToObject<StorageBlobCreatedEventData>();
                                var storageDiagnostics =
JObject.Parse(blobCreatedEvent.StorageDiagnostics.ToString()).ToString(Newtons
oft.Json.Formatting.None);

                                log.LogInformation($@"Received
{BlobCreatedEvent} Event:
    - Api=[{blobCreatedEvent.Api}]
    - BlobType=[{blobCreatedEvent.BlobType}]
    - ClientRequestId=[{blobCreatedEvent.ClientRequestId}]
    - ContentLength=[{blobCreatedEvent.ContentLength}]
    - ContentType=[{blobCreatedEvent.ContentType}]
    - ETag=[{blobCreatedEvent.ETag}]
    - RequestId=[{blobCreatedEvent.RequestId}]
    - Sequencer=[{blobCreatedEvent.Sequencer}]
    - StorageDiagnostics=[{storageDiagnostics}]
```

```
    - Url=[{blobCreatedEvent.Url}]
");

                                // Set message label
                                message.Label = "BlobCreatedEvent";

                                // Add custom properties
                                message.UserProperties.Add("id",
eventGridEvent.Id);
                                message.UserProperties.Add("topic",
eventGridEvent.Topic);
                                message.UserProperties.Add("eventType",
eventGridEvent.EventType);
                                message.UserProperties.Add("eventTime",
eventGridEvent.EventTime);
                                message.UserProperties.Add("subject",
eventGridEvent.Subject);
                                message.UserProperties.Add("api",
blobCreatedEvent.Api);
                                message.UserProperties.Add("blobType",
blobCreatedEvent.BlobType);
                                message.UserProperties.Add("clientRequestId",
blobCreatedEvent.ClientRequestId);
                                message.UserProperties.Add("contentLength",
blobCreatedEvent.ContentLength);
                                message.UserProperties.Add("contentType",
blobCreatedEvent.ContentType);
                                message.UserProperties.Add("eTag",
blobCreatedEvent.ETag);
                                message.UserProperties.Add("requestId",
blobCreatedEvent.RequestId);
                                message.UserProperties.Add("sequencer",
blobCreatedEvent.Sequencer);

message.UserProperties.Add("storageDiagnostics", storageDiagnostics);
                                message.UserProperties.Add("url",
blobCreatedEvent.Url);

                                // Add message to AsyncCollector
                                await asyncCollector.AddAsync(message);

                                // Telemetry
                                telemetry.Context.Operation.Id =
context.InvocationId.ToString();
                                telemetry.Context.Operation.Name =
"BlobCreatedEvent";

telemetry.TrackEvent($"[{blobCreatedEvent.Url}] blob created");
                                var properties = new Dictionary<string,
string>
                                {
```

```
                                    { "BlobType", blobCreatedEvent.BlobType },
                                    { "ContentType ",
blobCreatedEvent.ContentType }
                                };
                                telemetry.TrackMetric("ProcessBlobEvents
Created", 1, properties);
                            }
                            break;

                        case BlobDeletedEvent:
                            {
                                var blobDeletedEvent =
jObject.ToObject<StorageBlobDeletedEventData>();
                                var storageDiagnostics =
JObject.Parse(blobDeletedEvent.StorageDiagnostics.ToString()).ToString(Newtons
oft.Json.Formatting.None);

                                log.LogInformation($@"Received
{BlobDeletedEvent} Event:
    - Api=[{blobDeletedEvent.Api}]
    - BlobType=[{blobDeletedEvent.BlobType}]
    - ClientRequestId=[{blobDeletedEvent.ClientRequestId}]
    - ContentType=[{blobDeletedEvent.ContentType}]
    - RequestId=[{blobDeletedEvent.RequestId}]
    - Sequencer=[{blobDeletedEvent.Sequencer}]
    - StorageDiagnostics=[{storageDiagnostics}]
    - Url=[{blobDeletedEvent.Url}]
");

                                // Set message label
                                message.Label = "BlobDeletedEvent";

                                // Add custom properties
                                message.UserProperties.Add("id",
eventGridEvent.Id);
                                message.UserProperties.Add("topic",
eventGridEvent.Topic);
                                message.UserProperties.Add("eventType",
eventGridEvent.EventType);
                                message.UserProperties.Add("eventTime",
eventGridEvent.EventTime);
                                message.UserProperties.Add("subject",
eventGridEvent.Subject);
                                message.UserProperties.Add("api",
blobDeletedEvent.Api);
                                message.UserProperties.Add("blobType",
blobDeletedEvent.BlobType);
                                message.UserProperties.Add("clientRequestId",
blobDeletedEvent.ClientRequestId);
```

```
                                message.UserProperties.Add("contentType",
blobDeletedEvent.ContentType);
                                message.UserProperties.Add("requestId",
blobDeletedEvent.RequestId);
                                message.UserProperties.Add("sequencer",
blobDeletedEvent.Sequencer);

message.UserProperties.Add("storageDiagnostics", storageDiagnostics);
                                message.UserProperties.Add("url",
blobDeletedEvent.Url);

                                // Add message to AsyncCollector
                                await asyncCollector.AddAsync(message);

                                // Telemetry
                                telemetry.Context.Operation.Id =
context.InvocationId.ToString();
                                telemetry.Context.Operation.Name =
"BlobDeletedEvent";

telemetry.TrackEvent($"[{blobDeletedEvent.Url}] blob deleted");
                                var properties = new Dictionary<string,
string>
                                {
                                    { "BlobType", blobDeletedEvent.BlobType },
                                    { "ContentType ",
blobDeletedEvent.ContentType }
                                };
                                telemetry.TrackMetric("ProcessBlobEvents
Deleted", 1, properties);
                            }
                            break;
                    }
                }
            }
            catch (Exception ex)
            {
                log.LogError(ex, ex.Message);
                throw;
            }
        }
        #endregion
    }
}
```

## Bash scripts

The GitHub sample includes three bash scripts that you can use to create the Azure Event Grid subscriptions. Any time a blob is created, updated or deleted in the file's container in the source storage account, the subscription sends an event to the following:

- Azure Function running on Azure.
- Azure Function running on your development computer.
- Azure Event Grid event viewer web app on Azure.

Make sure to properly assign a value to variables before running the script. Also, make sure to run scripts using an Azure Active Directory account that is the owner or a contributor of the Azure subscription where you intend to deploy the solution.

### create-event-grid-subscription-for-azure-function.sh

```
#!/bin/bash

# variables
location="WestEurope"
storageAccountName="Contoso"
storageAccountResourceGroup="ContosoResourceGroup"
functionAppName="EventGridBlobEvents"
functionAppResourceGroup="EventGridBlobEventsResourceGroup"
functionName="ProcessBlobEvents"
subscriptionName='ContosoAzureFunctionSubscriber'
deadLetterContainerName="deadletter"
filesContainerName="files"
subjectBeginsWith="/blobServices/default/containers/"$filesContainerName

# functions
function getEventGridExtensionKey
{
    # get Kudu username
    echo "Retrieving username from ["$functionAppName"] Azure Function
publishing profile..."
    username=$(az functionapp deployment list-publishing-profiles --name $1 --
resource-group $2 --query '[?publishMethod==`MSDeploy`].userName' --output
tsv)

    if [ -n $username ]; then
        echo "["$username"] username successfully retrieved"
    else
        echo "No username could be retrieved"
        return
    fi

    # get Kudu password
    echo "Retrieving password from ["$functionAppName"] Azure Function
publishing profile..."
```

```
    password=$(az functionapp deployment list-publishing-profiles --name $1 --
resource-group $2 --query '[?publishMethod==`MSDeploy`].userPWD' --output tsv)

    if [ -n $password ]; then
        echo "["$password"] password successfully retrieved"
    else
        echo "No password could be retrieved"
        return
    fi

    # get jwt
    echo "Retrieving JWT token from Azure Function \ Kudu Management API..."
    jwt=$(sed -e 's/^"//' -e 's/"$//' <<< $(curl
https://$functionAppName.scm.azurewebsites.net/api/functions/admin/token --
user $username":"$password --silent))

    if [ -n $jwt ]; then
        echo "JWT token successfully retrieved"
    else
        echo "No JWT token could be retrieved"
        return
    fi

    # get eventgrid_extension key
    echo "Retrieving [eventgrid_extension] key..."
    eventGridExtensionKey=$(sed -e 's/^"//' -e 's/"$//' <<< $(curl -H 'Accept:
application/json' -H "Authorization: Bearer ${jwt}"
https://$functionAppName.azurewebsites.net/admin/host/systemkeys/eventgrid_ext
ension --silent | jq .value))

    if [ -n $eventGridExtensionKey ]; then
        echo "[eventgrid_extension] key successfully retrieved"
    else
        echo "No [eventgrid_extension] key could be retrieved"
        return
    fi
}

# check if the storage account exists
echo "Checking if ["$storageAccountName"] storage account actually exists..."

set +e
(
    az storage account show --name $storageAccountName --resource-group
$storageAccountResourceGroup &> /dev/null
)

if [ $? != 0 ]; then
      echo "No ["$storageAccountName"] storage account actually exists"
```

```
    set -e
    (
        # create the storage account
        az storage account create \
        --name $storageAccountName \
        --resource-group $storageAccountResourceGroup \
        --location $location \
        --sku Standard_LRS \
        --kind BlobStorage \
        --access-tier Hot 1> /dev/null
    )
    echo "["$storageAccountName"] storage account successfully created"
else
       echo "["$storageAccountName"] storage account already exists"
fi

# get storage account connection string
echo "Retrieving the connection string for ["$storageAccountName"] storage
account..."
connectionString=$(az storage account show-connection-string --name
$storageAccountName --resource-group $storageAccountResourceGroup --query
connectionString --output tsv)

if [ -n $connectionString ]; then
    echo "The connection string for ["$storageAccountName"] storage account is
["$connectionString"]"
else
    echo "Failed to retrieve the connection string for ["$storageAccountName"]
storage account"
    return
fi

# checking if deadletter container exists
echo "Checking if ["$deadLetterContainerName"] container already exists..."
set +e
(
    az storage container show --name $deadLetterContainerName --connection-
string $connectionString &> /dev/null
)

if [ $? != 0 ]; then
       echo "No ["$deadLetterContainerName"] container actually exists in
["$storageAccountName"] storage account"
    set -e
    (
        # create deadletter container
        az storage container create \
        --name $deadLetterContainerName \
        --public-access off \
```

```
        --connection-string $connectionString 1> /dev/null
    )
    echo "["$deadLetterContainerName"] container successfully created in
["$storageAccountName"] storage account"
else
      echo "A container called ["$deadLetterContainerName"] already exists in
["$storageAccountName"] storage account"
fi

# checking if files container exists
echo "Checking if ["$filesContainerName"] container already exists..."
set +e
(
    az storage container show --name $filesContainerName --connection-string
$connectionString &> /dev/null
)

if [ $? != 0 ]; then
      echo "No ["$filesContainerName"] container actually exists in
["$storageAccountName"] storage account"
    set -e
    (
        # create files container
        az storage container create \
        --name $filesContainerName \
        --public-access off \
        --connection-string $connectionString 1> /dev/null
    )
    echo "["$filesContainerName"] container successfully created in
["$storageAccountName"] storage account"
else
      echo "A container called ["$filesContainerName"] already exists in
["$storageAccountName"] storage account"
fi

# retrieve resource id for the storage account
echo "Retrieving the resource id for ["$storageAccountName"] storage
account..."
storageAccountId=$(az storage account show --name $storageAccountName --
resource-group $storageAccountResourceGroup --query id --output tsv 2>
/dev/null)

if [ -n $storageAccountId ]; then
    echo "Resource id for ["$storageAccountName"] storage account successfully
retrieved: ["$storageAccountId"]"
else
    echo "Failed to retrieve resource id for ["$storageAccountName"] storage
account"
    return
fi
```

```
# retrieve eventgrid_extensionkey
getEventGridExtensionKey $functionAppName $functionAppResourceGroup

if [ -z $eventGridExtensionKey ]; then
    echo "Failed to retrieve eventgrid_extensionkey"
    return
fi

# creating the endpoint URL for the Azure Function
endpointUrl="https://$functionAppName.azurewebsites.net/runtime/webhooks/event
grid?functionName=$functionName&code=$eventGridExtensionKey"

echo "The endpoint for the ["$functionName"] function in the
["$functionAppName"] function app is ["$endpointUrl"]"

echo "Checking if Azure CLI eventgrid extension is installed..."
set +e
(
    az extension show --name eventgrid --query name --output tsv &> /dev/null
)

if [ $? != 0 ]; then
      echo "The Azure CLI eventgrid extension was not found. Installing the
extension..."
  az extension add --name eventgrid
else
    echo "Azure CLI eventgrid extension successfully found. Updating the
extension to the latest version..."
  az extension update --name eventgrid
fi

# checking if the subscription already exists
echo "Checking if ["$subscriptionName"] Event Grid subscription already exists
for ["$storageAccountName"] storage account..."
set +e
(
    az eventgrid event-subscription show --name $subscriptionName --source-
resource-id $storageAccountId &> /dev/null
)

if [ $? != 0 ]; then
      echo "No ["$subscriptionName"] Event Grid subscription actually exists
for ["$storageAccountName"] storage account"
    echo "Creating a subscription for the ["$endpointUrl"] endpoint of the
["$functionName"] Azure Function..."

    set +e
    (
```

```
        az eventgrid event-subscription create \
        --source-resource-id $storageAccountId \
        --name $subscriptionName \
        --endpoint-type webhook \
        --endpoint $endpointUrl \
        --subject-begins-with $subjectBeginsWith \
        --deadletter-endpoint
$storageAccountId/blobServices/default/containers/$deadLetterContainerName 1>
/dev/null
    )

    if [ $? == 0 ]; then
        echo "["$subscriptionName"] Event Grid subscription successfully
created"
    fi
else
      echo "An Event Grid subscription called ["$subscriptionName"] already
exists for ["$storageAccountName"] storage account"
fi
```

## create-event-grid-subscription-for-local-function.sh

```
#!/bin/bash

# variables
location="WestEurope"
storageAccountName="Contoso"
storageAccountResourceGroup="ContosoResourceGroup"
subscriptionName='ContosoLocalDebugging'
ngrockSubdomain="db1abac5"
functionName="ProcessBlobEvents"
endpointUrl="https://"$ngrockSubdomain".ngrok.io/runtime/webhooks/EventGrid?fu
nctionName="$functionName
deadLetterContainerName="deadletter"
filesContainerName="files"
subjectBeginsWith="/blobServices/default/containers/"$filesContainerName

# check if the storage account exists
echo "Checking if ["$storageAccountName"] storage account actually exists..."

set +e
(
    az storage account show --name $storageAccountName --resource-group
$storageAccountResourceGroup &> /dev/null
)

if [ $? != 0 ]; then
      echo "No ["$storageAccountName"] storage account actually exists"
    set -e
```

```
    (
        # create the storage account
        az storage account create \
        --name $storageAccountName \
        --resource-group $storageAccountResourceGroup \
        --location $location \
        --sku Standard_LRS \
        --kind BlobStorage \
        --access-tier Hot 1> /dev/null
    )
    echo "["$storageAccountName"] storage account successfully created"
else
      echo "["$storageAccountName"] storage account already exists"
fi

# get storage account connection string
echo "Retrieving the connection string for ["$storageAccountName"] storage
account..."
connectionString=$(az storage account show-connection-string --name
$storageAccountName --resource-group $storageAccountResourceGroup --query
connectionString --output tsv)

if [ -n $connectionString ]; then
    echo "The connection string for ["$storageAccountName"] storage account is
["$connectionString"]"
else
    echo "Failed to retrieve the connection string for ["$storageAccountName"]
storage account"
    return
fi

# checking if deadletter container exists
echo "Checking if ["$deadLetterContainerName"] container already exists..."
set +e
(
    az storage container show --name $deadLetterContainerName --connection-
string $connectionString &> /dev/null
)

if [ $? != 0 ]; then
      echo "No ["$deadLetterContainerName"] container actually exists in
["$storageAccountName"] storage account"
    set -e
    (
        # create deadletter container
        az storage container create \
        --name $deadLetterContainerName \
        --public-access off \
        --connection-string $connectionString 1> /dev/null
```

```
    )
    echo "["$deadLetterContainerName"] container successfully created in
["$storageAccountName"] storage account"
else
      echo "A container called ["$deadLetterContainerName"] already exists in
["$storageAccountName"] storage account"
fi

# checking if files container exists
echo "Checking if ["$filesContainerName"] container already exists..."
set +e
(
    az storage container show --name $filesContainerName --connection-string
$connectionString &> /dev/null
)

if [ $? != 0 ]; then
      echo "No ["$filesContainerName"] container actually exists in
["$storageAccountName"] storage account"
    set -e
    (
        # create files container
        az storage container create \
        --name $filesContainerName \
        --public-access off \
        --connection-string $connectionString 1> /dev/null
    )
    echo "["$filesContainerName"] container successfully created in
["$storageAccountName"] storage account"
else
      echo "A container called ["$filesContainerName"] already exists in
["$storageAccountName"] storage account"
fi

# retrieve resource id for the storage account
echo "Retrieving the resource id for ["$storageAccountName"] storage
account..."
storageAccountId=$(az storage account show --name $storageAccountName --
resource-group $storageAccountResourceGroup --query id --output tsv 2>
/dev/null)

if [ -n $storageAccountId ]; then
    echo "Resource id for ["$storageAccountName"] storage account successfully
retrieved: ["$storageAccountId"]"
else
    echo "Failed to retrieve resource id for ["$storageAccountName"] storage
account"
    return
fi
```

```
echo "Checking if Azure CLI eventgrid extension is installed..."
set +e
(
    az extension show --name eventgrid --query name --output tsv &> /dev/null
)

if [ $? != 0 ]; then
       echo "The Azure CLI eventgrid extension was not found. Installing the
extension..."
  az extension add --name eventgrid
else
    echo "Azure CLI eventgrid extension successfully found. Updating the
extension to the latest version..."
  az extension update --name eventgrid
fi

# checking if the subscription already exists
echo "Checking if ["$subscriptionName"] Event Grid subscription already exists
for ["$storageAccountName"] storage account..."
set +e
(
    az eventgrid event-subscription show --name $subscriptionName --source-
resource-id $storageAccountId &> /dev/null
)

if [ $? != 0 ]; then
       echo "No ["$subscriptionName"] Event Grid subscription actually exists
for ["$storageAccountName"] storage account"
    echo "Creating a subscription for the ["$endpointUrl"] ngrock local
endpoint..."

       set +e
       (
              az eventgrid event-subscription create \
         --source-resource-id $storageAccountId \
         --name $subscriptionName \
         --endpoint-type webhook \
         --endpoint $endpointUrl \
         --subject-begins-with $subjectBeginsWith \
         --deadletter-endpoint
$storageAccountId/blobServices/default/containers/$deadLetterContainerName 1>
/dev/null
       )

    if [ $? == 0 ]; then
        echo "["$subscriptionName"] Event Grid subscription successfully
created"
    fi
else
```

```
      echo "An Event Grid subscription called ["$subscriptionName"] already
exists for ["$storageAccountName"] storage account"
fi
```

create-event-grid-subscription-for-web-app.sh

```
#!/bin/bash

# variables
location="WestEurope"
storageAccountName="Contoso"
storageAccountResourceGroup="ContosoResourceGroup"
subscriptionName='ContosoWebApp'
webAppSubdomain="Contosoeventgridviewer"
functionName="ProcessBlobEvents"
endpointUrl="https://"$webAppSubdomain".azurewebsites.net/api/updates"
deadLetterContainerName="deadletter"
filesContainerName="files"

# check if the storage account exists
echo "Checking if ["$storageAccountName"] storage account actually exists..."

set +e
(
    az storage account show --name $storageAccountName --resource-group
$storageAccountResourceGroup &> /dev/null
)

if [ $? != 0 ]; then
      echo "No ["$storageAccountName"] storage account actually exists"
    set -e
    (
        # create the storage account
        az storage account create \
        --name $storageAccountName \
        --resource-group $storageAccountResourceGroup \
        --location $location \
        --sku Standard_LRS \
        --kind BlobStorage \
        --access-tier Hot 1> /dev/null
    )
    echo "["$storageAccountName"] storage account successfully created"
else
      echo "["$storageAccountName"] storage account already exists"
fi

# get storage account connection string
echo "Retrieving the connection string for ["$storageAccountName"] storage
account..."
```

```
connectionString=$(az storage account show-connection-string --name
$storageAccountName --resource-group $storageAccountResourceGroup --query
connectionString --output tsv)

if [ -n $connectionString ]; then
    echo "The connection string for ["$storageAccountName"] storage account is
["$connectionString"]"
else
    echo "Failed to retrieve the connection string for ["$storageAccountName"]
storage account"
    return
fi

# checking if deadletter container exists
echo "Checking if ["$deadLetterContainerName"] container already exists..."
set +e
(
    az storage container show --name $deadLetterContainerName --connection-
string $connectionString &> /dev/null
)

if [ $? != 0 ]; then
      echo "No ["$deadLetterContainerName"] container actually exists in
["$storageAccountName"] storage account"
    set -e
    (
        # create deadletter container
        az storage container create \
        --name $deadLetterContainerName \
        --public-access off \
        --connection-string $connectionString 1> /dev/null
    )
    echo "["$deadLetterContainerName"] container successfully created in
["$storageAccountName"] storage account"
else
      echo "A container called ["$deadLetterContainerName"] already exists in
["$storageAccountName"] storage account"
fi

# checking if files container exists
echo "Checking if ["$filesContainerName"] container already exists..."
set +e
(
    az storage container show --name $filesContainerName --connection-string
$connectionString &> /dev/null
)

if [ $? != 0 ]; then
      echo "No ["$filesContainerName"] container actually exists in
["$storageAccountName"] storage account"
```

```
    set -e
    (
        # create files container
        az storage container create \
        --name $filesContainerName \
        --public-access off \
        --connection-string $connectionString 1> /dev/null
    )
    echo "["$filesContainerName"] container successfully created in
["$storageAccountName"] storage account"
else
      echo "A container called ["$filesContainerName"] already exists in
["$storageAccountName"] storage account"
fi

# retrieve resource id for the storage account
echo "Retrieving the resource id for ["$storageAccountName"] storage
account..."
storageAccountId=$(az storage account show --name $storageAccountName --
resource-group $storageAccountResourceGroup --query id --output tsv 2>
/dev/null)

if [ -n $storageAccountId ]; then
    echo "Resource id for ["$storageAccountName"] storage account successfully
retrieved: ["$storageAccountId"]"
else
    echo "Failed to retrieve resource id for ["$storageAccountName"] storage
account"
    return
fi

echo "Checking if Azure CLI eventgrid extension is installed..."
set +e
(
    az extension show --name eventgrid --query name --output tsv &> /dev/null
)

if [ $? != 0 ]; then
      echo "The Azure CLI eventgrid extension was not found. Installing the
extension..."
  az extension add --name eventgrid
else
    echo "Azure CLI eventgrid extension successfully found. Updating the
extension to the latest version..."
  az extension update --name eventgrid
fi

# checking if the subscription already exists
echo "Checking if ["$subscriptionName"] Event Grid subscription already exists
for ["$storageAccountName"] storage account..."
```

```
set +e
(
    az eventgrid event-subscription show --name $subscriptionName --source-
resource-id $storageAccountId &> /dev/null
)

if [ $? != 0 ]; then
      echo "No ["$subscriptionName"] Event Grid subscription actually exists
for ["$storageAccountName"] storage account"
    echo "Creating the subscription for the ["$endpointUrl"] endpoint of the
Azure Event Grid Viewer web app..."
    set +e
    (
        az eventgrid event-subscription create \
        --source-resource-id $storageAccountId \
        --name $subscriptionName \
        --endpoint-type webhook \
        --endpoint $endpointUrl \
        --deadletter-endpoint
$storageAccountId/blobServices/default/containers/$deadLetterContainerName 1>
/dev/null
    )
    if [ $? == 0 ]; then
        echo "["$subscriptionName"] Event Grid subscription successfully
created"
    fi
else
      echo "An Event Grid subscription called ["$subscriptionName"] already
exists for ["$storageAccountName"] storage account"
fi
```

## Alternative: Functions as event handlers

After this serverless app solution was created for the self-driving car project, a new feature was added to Azure Functions that supports functions as event handlers. Instead of pointing at a function's webhook endpoint, as this solution does, in CLI, you can run the following:

```
az eventgrid event-subscription create \
--name azureFunctionEventSubscription \
--source-resource-id
/subscriptions/{SubID}/resourceGroups/{RG}/providers/Microsoft.EventGrid/topic
s/topic1 \
--endpoint
/subscriptions/{SubID}/resourceGroups/{RG}/providers/Microsoft.Web/sites/{func
tionappname}/functions/{functionname}
```

Event Grid uses role-based access control (RBAC) to ensure you have access to publish to the function, and it takes care of getting the function key and setting up the event subscription. As an added benefit, Event Grid also takes care of keeping the key up to date if you roll keys. This feature is not fully supported by the Azure CLI, so you must explicitly install the Event Grid CLI

extension using the following command:

```
az extension add --name eventgrid
```

If you had an earlier version of the extension, the command uninstalls it before installing the later version. The minimum requirement is version 0.4.4.

To ensure that the right version of the extension is installed on your machine, run the following command:

```
az extension list
```

The command returns a JSON array like the following:

```
[
  ...
  {
    "extensionType": "whl",
    "name": "eventgrid",
    "version": "0.4.4"
  },
  ...
]
```

# Upload files to the container

You can use Azure Storage Explorer to upload files to a container in your storage account. This container is monitored by the event grid storage topic Figure 2 shows.

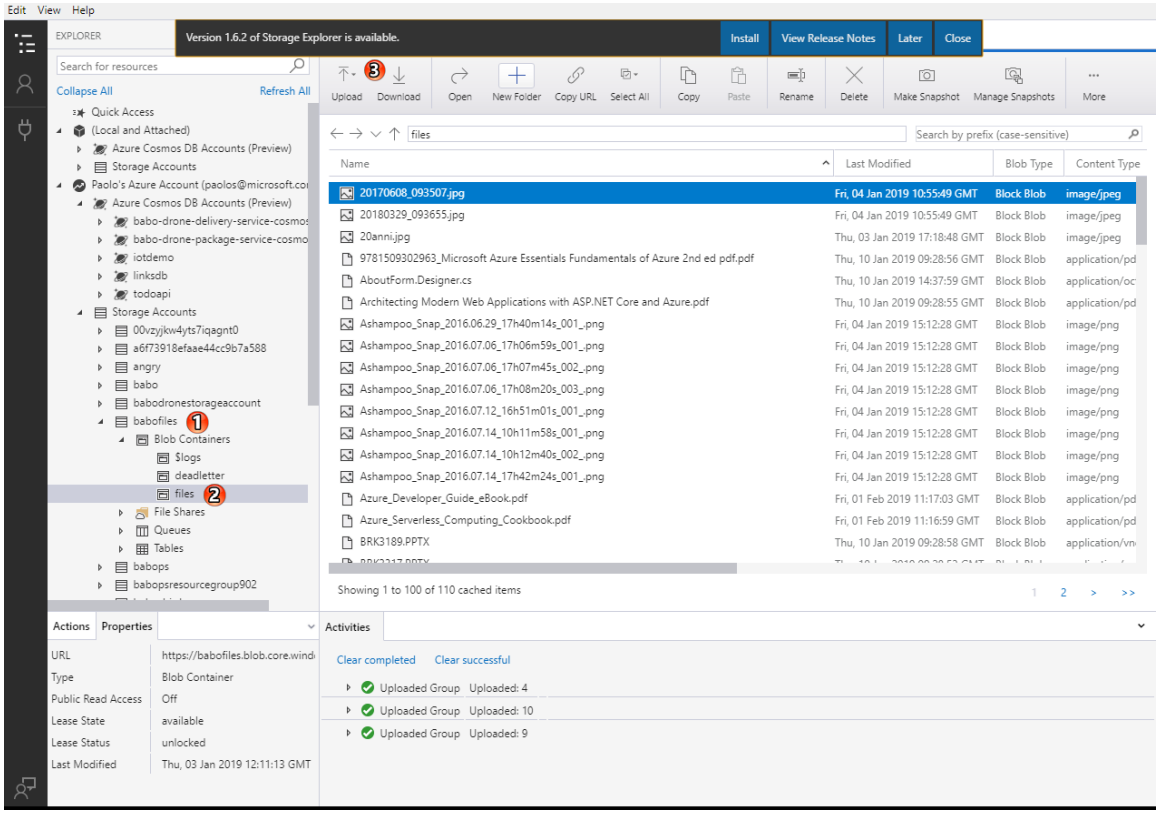



Figure 2. You can upload files to the solution using Storage Explorer.

# View messages from the queue in Service Bus Explorer

To receive and view messages from the queue, you can create a tool using the ServiceBusExplorer GitHub project. As Figure 3 shows, Service Bus Explorer displays the messages sent by the function to the queue in the target Service Bus namespace.

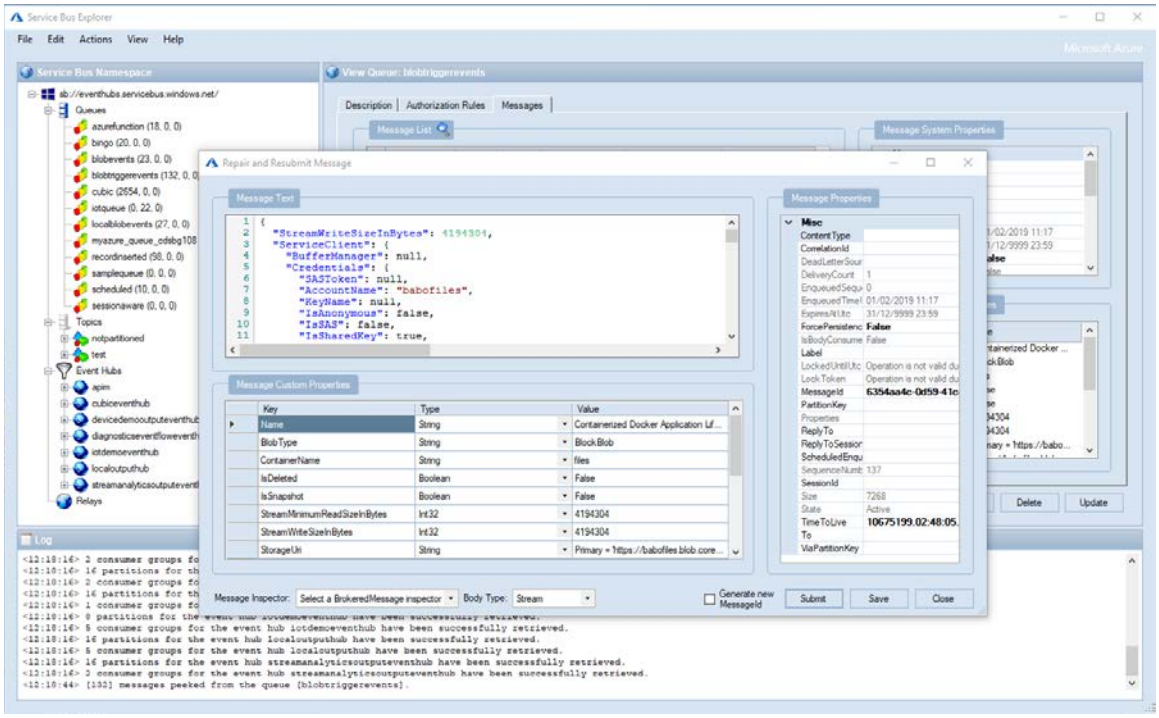



Figure 3. You can use Service Bus Explorer to view messages sent by the function to the queue in the target Service Bus namespace.

# Query metrics in Application Insights

You can use Kusto Query Language to create queries in Application Insights that analyze the metrics and logs generated by the Azure Function. As Figure 4 shows, the **customMetrics** query shows when blobs are created and deleted as a linear progression. The **customEvents** query displays the individual events tracked by the Azure Function.

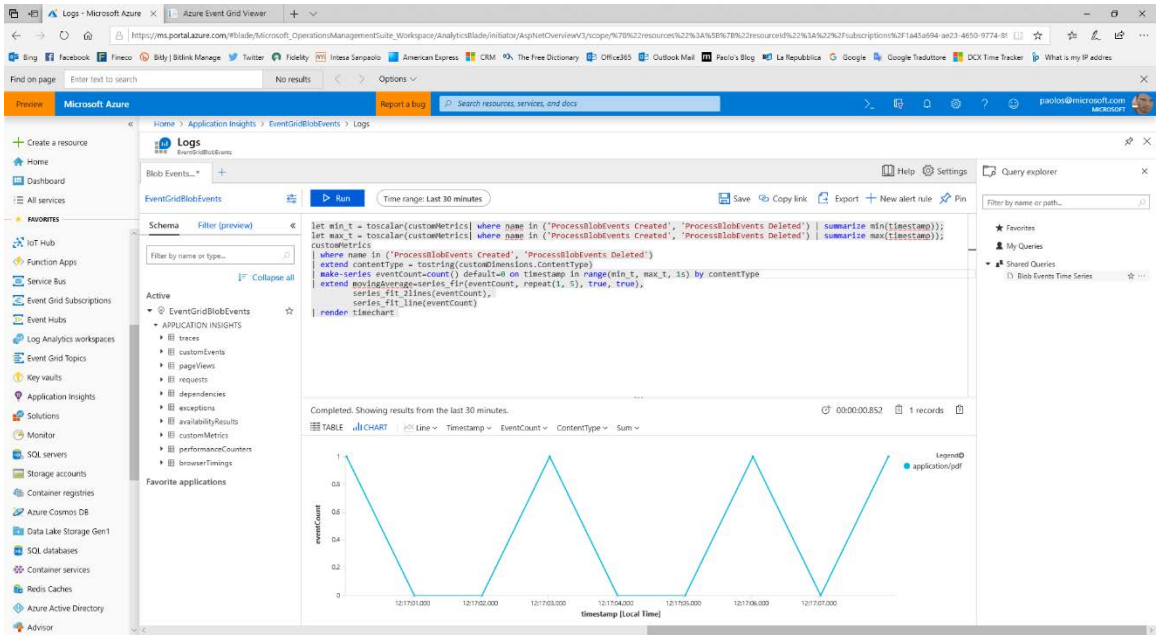


Figure 4. You can use Kusto Query Language to query metrics in Application Insights.

## Sample query: customMetrics

```
let min_t = toscalar(customMetrics| where name in ('ProcessBlobEvents
Created', 'ProcessBlobEvents Deleted') | summarize min(timestamp));
let max_t = toscalar(customMetrics| where name in ('ProcessBlobEvents
Created', 'ProcessBlobEvents Deleted') | summarize max(timestamp));
customMetrics
| where name in ('ProcessBlobEvents Created', 'ProcessBlobEvents Deleted')
| extend contentType = tostring(customDimensions.ContentType)
| make-series eventCount=count() default=0 on timestamp in range(min_t, max_t,
1s) by contentType
| extend movingAverage=series_fir(eventCount, repeat(1, 5), true, true),
         series_fit_2lines(eventCount),
         series_fit_line(eventCount)
| render timechart
```

## Sample query: customEvents

```
customEvents
| project timestamp, message=name, operation=operation_Name
```

# Learn more

For more information, see the following resources:

- Azure Blob Storage
- Azure Functions
- Azure Event Grid
- Azure CLI