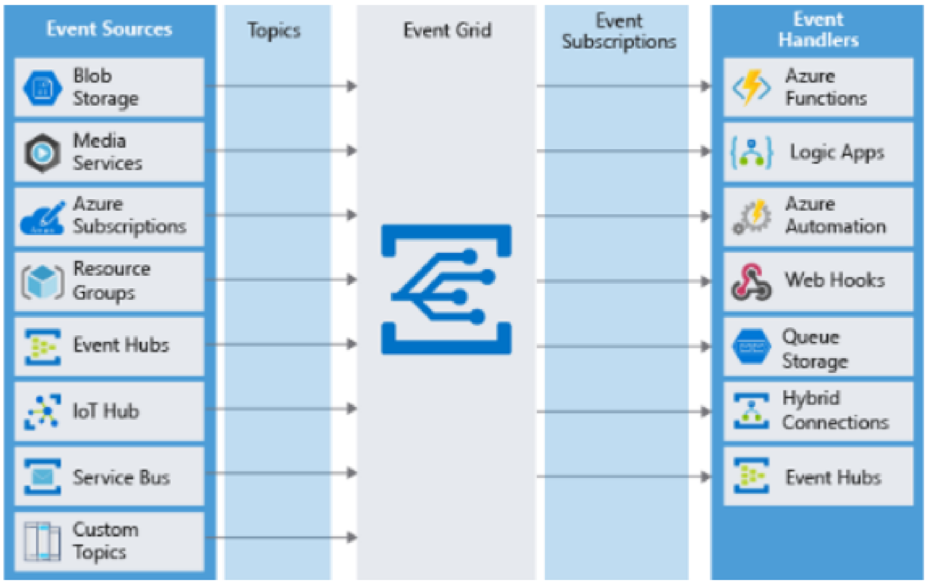
V.2 Develop event-based solutions

jeudi 16 septembre 2021 14:34

Implement solutions that use Azure Event Grid

Definition

- **Events** - What happened.
- **Event sources** - Where the event took place.
- **Topics** - The endpoint where publishers 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.



Composition

- **Events** - What happened. Smallest amount of info (source, time, id, event-related info). 64KB max.
- **Event sources** - Where the event took place. Related to one or more event types. For example, IoT Hub is the event source for device created events, Azure Storage for blob created events.
- **Topics** - The endpoint where publishers send events. The publisher creates the event grid topic and decides whether an event source needs one topic or more than one topic. Subscribers decides which topics to subscribe to. 2 types of topics :
  - System topics : provided by Azure services, owned by the publisher, only subscribable by providing the info about the resource you want to receive events from.
  - Custom topics : app & third-party topics. Unlike system's you can see the custom topic in your subscription.
- **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. It tells Event grid which events on a topic you're interested in receiving. You can filter the event that are sent to the endpoint you provide. Filter by event type, subject pattern or data key.
- **Event handlers** - The app or service reacting to the event. The place where the event is sent.

Event Schema

<= 1MB -> if >, 413 payload too large

```
[
  {
    "topic": string,
    "subject": string,
    "id": string,
    "eventType": string,
    "eventTime": string,
    "data":{
      object-unique-to-each-publisher
    },
    "dataVersion": string,
    "metadataVersion": string
  }
]
```

Event delivery

Order not guaranteed for event delivery.

If event grid receives an error for an event delivery attempt,
    -> if configuration-related error (400, 403, 413)
        -> dead-letter or drop if not configured

-> else
        -> wait 30 sec for a response, and retry if no response
        Delay all subsequent retries

Retry policy
customizable when creating an event subscription

```
az eventgrid event-subscription create \
  -g gridResourceGroup \
  --topic-name <topic_name> \
  --name <event_subscription_name> \
  --endpoint <endpoint_URL> \
  --max-delivery-attempts 18
```

Output batching
can be turned on per-subscription (configure the max event per batch & the preferred batch size in kilobytes)

Control access to events

RBAC Build-in roles :

Role	Description
Event Grid Subscription Reader	Lets you read Event Grid event subscriptions.
Event Grid Subscription Contributor	Lets you manage Event Grid event subscription operations.
Event Grid Contributor	Lets you create and manage Event Grid resources.
Event Grid Data Sender	Lets you send events to Event Grid topics.

The **Microsoft.EventGrid/EventSubscriptions/Write** permission is required on the resource that is the event source in order to subscribe to events in EventGrid

Receive events by using Webhooks

Trusted Azure webhooks :
- Logic App with Event Grid connector
- Azure Automation via Webhook
- Azure function with Event Grid trigger

Receive events in another way
Ex: HTTP trigger based Azure function

-> Synchronous (at the time of event subscription creation) or asynchronous(if third party service used) validation handshake needed

Event filtering

Subscribers use the subject, event type or data key to filter and route events.

By default, all event types are sent. To filter by event type :

```
"filter": {
  "includedEventTypes": [
    "Microsoft.Resources.ResourceWriteFailure",
    "Microsoft.Resources.ResourceWriteSuccess"
  ]
}
```

To **filter by subject**

```
"filter": {
  "subjectBeginsWith": "/blobServices/default/containers/mycontainer/log",
  "subjectEndsWith": ".jpg"
}
```

To filter by data key

```
"filter": {
  "advancedFilters": [
    {
      "operatorType": "NumberGreaterThanOrEquals",
      "key": "Data.Key1",
      "value": 5
    },
    {
      "operatorType": "StringContains",
      "key": "Subject",
      "values": ["container1", "container2"]
    }
```

```
        }
    ]
}
```

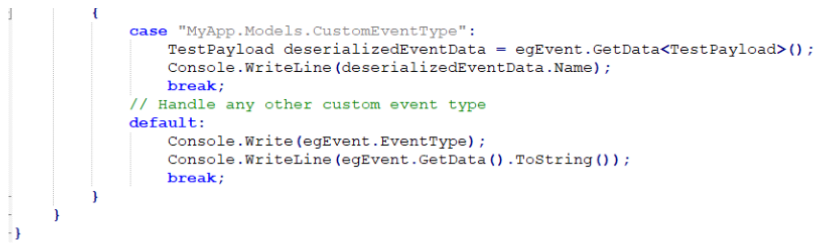
```
# Create resource group
let rNum=$RANDOM*$RANDOM
myLocation=<myLocation>
myTopicName="az204-egtopic-${rNum}"
mySiteName="az204-egsite-${rNum}"
mySiteURL="https://${mySiteName}.azurewebsites.net"

az group create --name az204-evgrid-rg --location $myLocation

# Enable an Event Grid resource provider
az provider register --namespace Microsoft.EventGrid

az provider show --namespace Microsoft.EventGrid --query "registrationState"

# Create a custom topic
az eventgrid topic create --name $myTopicName \
    --location $myLocation \
    --resource-group az204-evgrid-rg

# Create a message endpoint
az deployment group create \
    --resource-group az204-evgrid-rg \
    --template-uri "https://raw.githubusercontent.com/Azure-Samples/azure-event-grid-viewer/main/azuredeploy.json" \
    --parameters siteName=$mySiteName hostingPlanName=viewerhost

echo "Your web app URL: ${mySiteURL}"

# Subscribe to a custom topic
endpoint="${mySiteURL}/api/updates"
subId=$(az account show --subscription "" | jq -r '.id')

az eventgrid event-subscription create \
    --source-resource-id "/subscriptions/$subId/resourceGroups/az204-evgrid-rg/providers/Microsoft.EventGrid/topics/$myTopicName" \
    --name az204ViewerSub \
    --endpoint $endpoint

# Send an event to the topic
topicEndpoint=$(az eventgrid topic show --name $myTopicName -g az204-evgrid-rg --query "endpoint" --output tsv)
key=$(az eventgrid topic key list --name $myTopicName -g az204-evgrid-rg --query "key1" --output tsv)

event='[{
    "id": "'"
    $RANDOM "'",
    "eventType": "recordInserted",
    "subject": "myapp/vehicles/motorcycles",
    "eventTime": "'`date +%Y-%m-%dT%H:%M:%S%z`'",
    "data": {
        "make": "Contoso",
        "model": "Monster"
    },
    "dataVersion": "1.0"
}]'

curl -X POST -H "aeg-sas-key: $key" -d "$event" $topicEndpoint
```

Publish events to event grid in .NET

```
// Create client
EventGridPublisherClient client = new EventGridPublisherClient(
    new Uri("<endpoint>"),
    new AzureKeyCredential("<access-key>"));

// Publish event grid events to an event grid topic
// Add EventGridEvents to a list to publish to the topic
List<EventGridEvent> eventsList = new List<EventGridEvent>
{
    new EventGridEvent(
        "ExampleEventSubject",
        "Example.EventType",
        "1.0",
        "This is the event data")
};

// Send the events
await client.SendEventsAsync(eventsList);
```

Receive events from event grid in .NET

```
// Receive events from event grid
// Parse the JSON payload into a list of events using EventGridEvent.Parse
EventGridEvent[] egEvents = EventGridEvent.Parse(jsonPayloadSampleOne);

foreach (EventGridEvent egEvent in egEvents)
{
    // If the event is a system event, TryGetSystemEventData() will return the deserialized system event
    if (egEvent.TryGetSystemEventData(out object systemEvent))
    {
        switch (systemEvent)
        {
            case SubscriptionValidationEventData subscriptionValidated:
                Console.WriteLine(subscriptionValidated.ValidationCode);
                break;
            case StorageBlobCreatedEventData blobCreated:
                Console.WriteLine(blobCreated.BlobType);
                break;
            // Handle any other system event type
            default:
                Console.WriteLine(egEvent.EventType);
                // we can get the raw Json for the event using GetData()
                Console.WriteLine(egEvent.GetData().ToString());
                break;
        }
    }
    else
    {
        switch (egEvent.EventType)
```



```
            {
                case "MyApp.Models.CustomEventType":
                    TestPayload deserializedEventData = egEvent.GetData<TestPayload>();
                    Console.WriteLine(deserializedEventData.Name);
                    break;
                // Handle any other custom event type
                default:
                    Console.Write(egEvent.EventType);
                    Console.WriteLine(egEvent.GetData().ToString());
                    break;
            }
        }
}
```

Implement solutions that use Azure Event Hubs

Definition

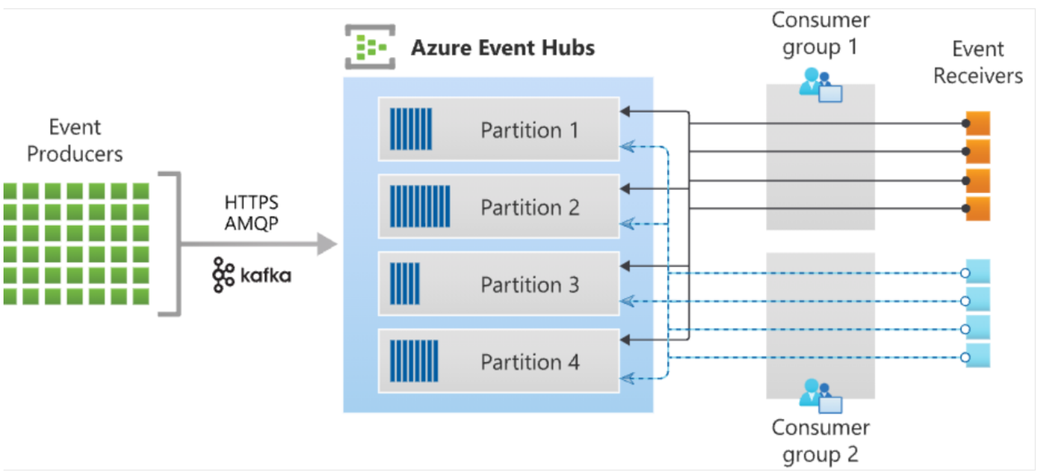
Event Hubs is a intermediary for the pub/sub model. However, unlike Event Grid, it is optimized for a **high throughput**, a **high number of editors, security & resiliency.** It is :

- Fully managed PaaS
- Real-time and batch processing
- Scalable
- Rich ecosystem

Key components

- An **Event Hub client** is the primary interface for developers interacting with the Event Hubs client library. There are several different Event Hub clients, each dedicated to a specific use of Event Hubs, such as publishing or consuming events.
- An **Event Hub producer** is a type of client that serves as a source of telemetry data, diagnostics information, usage logs, or other log data, as part of an embedded device solution, a mobile device application, a game title running on a console or other device, some client or server based business solution, or a web site.
- An **Event Hub consumer** is a type of client which reads information from the Event Hub and allows processing of it. Processing may involve aggregation, complex computation and filtering. Processing may also involve distribution or storage of the information in a raw or transformed fashion. Event Hub consumers are often robust and high-scale platform infrastructure parts with built-in analytics capabilities, like Azure Stream Analytics, Apache Spark, or Apache Storm.
- A **partition** is an ordered sequence of events that is held in an Event Hub. Partitions are a means of data organization associated with the parallelism required by event consumers. Azure Event Hubs provides message streaming through a partitioned consumer pattern in which each consumer only reads a specific subset, or partition, of the message stream. As newer events arrive, they are added to the end of this sequence. The number of partitions is specified at the time an Event Hub is created and cannot be changed.
- A **consumer group** is a view of an entire Event Hub. Consumer groups enable multiple consuming applications to each have a separate view of the event stream, and to read the stream independently at their own pace and from their own position. There can be at most 5 concurrent readers on a partition per consumer group; however it is recommended that there is only one active consumer for a given partition and consumer group pairing. Each active reader receives all of the events from its partition; if there are multiple readers on the same partition, then they will receive duplicate events.
- **Event receivers**: Any entity that reads event data from an event hub. All Event Hubs consumers connect via the AMQP 1.0 session. The Event Hubs service delivers events through a session as they become available. All Kafka consumers connect via the Kafka protocol 1.0 and later.
- **Throughput units** or **processing units**: Pre-purchased units of capacity that control the throughput capacity of Event Hubs.

he following figure shows the Event Hubs stream processing architecture:



Event Hub Capture

Automatically capture streaming/real-time data from Event Hub and put it in Blob or Data lake storage.

Scaling, Thread safety and Checkpointing

In .NET, use **EventProcessorClient** for reading & processing events.

Event processor clients can work cooperatively within the context of a consumer group for a given event hub. Clients will automatically manage distribution and balancing of work as instances become available or unavailable for the group. Functions processes the events is called sequentially for a given partition. Each call to the function delivers a single event from a specific partition. Events from different partitions can be processed concurrently and any shared state that is accessed across partitions have to be synchronized.

Event Processor marks the position of the last successfully processed event within a partition in order to enable another instance to resume processing the partition if the first one disconnects.

Access control

RBAC Build-in roles :

- Azure Event Hubs Data Owner: Use this role to give *complete access* to Event Hubs resources.
- Azure Event Hubs Data Sender: Use this role to give *send access* to Event Hubs resources.
- Azure Event Hubs Data Receiver: Use this role to give *receiving access* to Event Hubs resources.

Authorize access with :

- Managed Identity
- Microsoft Identity Platform
- SAS

Dev

Get partitions

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

await using (var producer = new EventHubProducerClient(connectionString, eventHubName))
{
    string[] partitionIds = await producer.GetPartitionIdsAsync();
}
```

Publish events to an Event Hub

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

await using (var producer = new EventHubProducerClient(connectionString, eventHubName))
{
    using EventDataBatch eventBatch = await producer.CreateBatchAsync();
    eventBatch.TryAdd(new EventData(new BinaryData("First")));
    eventBatch.TryAdd(new EventData(new BinaryData("Second")));

    await producer.SendAsync(eventBatch);
}
```

Read events from an Event Hub

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

string consumerGroup = EventHubConsumerClient.DefaultConsumerGroupName;

await using (var consumer = new EventHubConsumerClient(consumerGroup, connectionString, eventHubName))
{
    using var cancellationSource = new CancellationTokenSource();
    cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));

    await foreach (PartitionEvent receivedEvent in consumer.ReadEventsAsync(cancellationSource.Token))
    {
        // At this point, the loop will wait for events to be available in the Event Hub.  When an event
        // is available, the loop will iterate with the event that was received.  Because we did not
        // specify a maximum wait time, the loop will wait forever unless cancellation is requested using
        // the cancellation token.
    }
}
```

Read events from an Event Hub partition

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

string consumerGroup = EventHubConsumerClient.DefaultConsumerGroupName;

await using (var consumer = new EventHubConsumerClient(consumerGroup, connectionString, eventHubName))
{
    EventPosition startingPosition = EventPosition.Earliest;
    string partitionId = (await consumer.GetPartitionIdsAsync()).First();

    using var cancellationSource = new CancellationTokenSource();
    cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));

    await foreach (PartitionEvent receivedEvent in consumer.ReadEventsFromPartitionAsync(partitionId, startingPosition, cancellationSource.Token))
    {
        // At this point, the loop will wait for events to be available in the partition.  When an event
        // is available, the loop will iterate with the event that was received.  Because we did not
        // specify a maximum wait time, the loop will wait forever unless cancellation is requested using
        // the cancellation token.
    }
}
```

Process events using an Event Processor client

```
var cancellationSource = new CancellationTokenSource();
cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));

var storageConnectionString = "<< CONNECTION STRING FOR THE STORAGE ACCOUNT >>";
var blobContainerName = "<< NAME OF THE BLOB CONTAINER >>";

var eventHubsConnectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";
var consumerGroup = "<< NAME OF THE EVENT HUB CONSUMER GROUP >>";

Task processEventHandler(ProcessEventArgs eventArgs) => Task.CompletedTask;
Task processErrorHandler(ProcessErrorEventArgs eventArgs) => Task.CompletedTask;

var storageClient = new BlobContainerClient(storageConnectionString, blobContainerName);
var processor = new EventProcessorClient(storageClient, consumerGroup, eventHubsConnectionString, eventHubName);

processor.ProcessEventAsync += processEventHandler;
processor.ProcessErrorAsync += processErrorHandler;

await processor.StartProcessingAsync();

try
{
    // The processor performs its work in the background; block until cancellation
    // to allow processing to take place.

    await Task.Delay(Timeout.Infinite, cancellationSource.Token);
}
catch (TaskCanceledException)
{
    // This is expected when the delay is canceled.
}

try
{
    await processor.StopProcessingAsync();
}
finally
{
    // To prevent leaks, the handlers should be removed when processing is complete.

    processor.ProcessEventAsync -= processEventHandler;
    processor.ProcessErrorAsync -= processErrorHandler;
}
```