Communication in a microservice architecture

Friday, June 4, 2021 7:22 PM

Azure Service Bus, Event Grid, and Event Hubs.

Client and services can communicate through many different types of communication, each one targeting a different scenario and goals. Initially, those types of communications can be classified in two axes.

The first axis defines if the protocol is synchronous or asynchronous:

• **Synchronous protocol.** HTTP is a synchronous protocol. The client sends a request and waits for a response from the service. That's independent of the client code execution that could be synchronous (thread is blocked) or asynchronous (thread isn't blocked, and the response will reach a callback eventually). The important point here is that the protocol (**HTTP/HTTPS**) is synchronous and the client code can only continue its task when it receives the HTTP server response.

• **Asynchronous protocol.** Other protocols like AMQP (a protocol supported by many operating systems and cloud environments) use asynchronous messages. The client code or message sender usually doesn't wait for a response. It just sends the message as when sending a message to a RabbitMQ queue or any other message broker.

The second axis defines if the communication has a single receiver or multiple receivers:
• **Single receiver**. Each request must be processed by exactly one receiver or service. An example of this communication is the Command pattern.

• **Multiple receivers**. Each request can be processed by zero to multiple receivers. This type of communication must be asynchronous. An example is the publish/subscribe mechanism used in patterns like Event-driven architecture. This is based on an event-bus interface or message broker when propagating data updates between multiple microservices through events; it's usually implemented through a service bus or similar artifact like Azure Service Bus by using topics and subscriptions.52 CHAPTER 4 | Architecting container and microservice-based applications

A microservice-based application will often use a combination of these communication styles. The most common type is single-receiver communication with a synchronous protocol like HTTP/HTTPS when invoking a regular Web API HTTP service. Microservices also typically use messaging protocols for asynchronous communication between microservices.
These axes are good to know so you have clarity on the possible communication mechanisms, but they're not the important concerns when building microservices. Neither the asynchronous nature of client thread execution nor the asynchronous nature of the selected protocol are the important points when integrating microservices. What is important is being able to integrate your microservices asynchronously while maintaining the independence of microservices, as explained in the following section.

If the producer expects an action from the consumer, that message is a command. If the message informs the consumer that an action has taken place, then the message is an event.

Event Based

Friday, June 4, 2021 7:25 PM

Ref: https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/messaging

An event is a type of message that a producer raises to announce facts.

The producer (known as the publisher in this context) **has no expectations that the events will result in any action**.

Interested consumer(s), can subscribe, listen for events, and take actions depending on their consumption scenario. Events can have multiple subscribers or no subscribers at all. Two different subscribers can react to an event with different actions and not be aware of one another.

The producer and consumer are loosely coupled and managed independently. The consumer isn't expected to acknowledge the event back to the producer. A consumer that is no longer interested in the events, can unsubscribe. The consumer is removed from the pipeline without affecting the producer or the overall functionality of the system.

There are two categories of events:

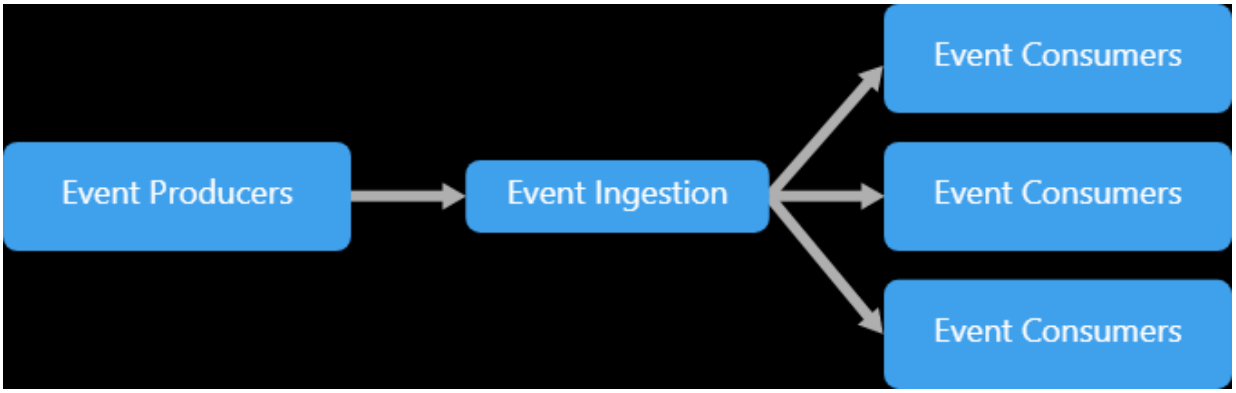
The producer raises events to announce discrete facts. A common use case is event notification. For example, Azure Resource Manager raises events when it creates, modifies, or deletes resources. A subscriber of those events could be a Logic App that sends alert emails.

The producer raises related events in a sequence, or a stream of events, over a period of time. Typically, a stream is consumed for statistical evaluation. The evaluation can be done within a temporal window or as events arrive. Telemetry is a common use case, for example, health and load monitoring of a system. Another case is event streaming from IoT devices.

Event-driven architecture style

Tuesday, July 6, 2021 3:58 PM

An event-driven architecture consists of event producers that generate a stream of events, and event consumers that listen for the events.



Events are delivered in near real time, so consumers can respond immediately to events as they occur. Producers are decoupled from consumers — a producer doesn't know which consumers are listening. Consumers are also decoupled from each other, and every consumer sees all of the events. This differs from a Competing Consumers pattern, where consumers pull messages from a queue and a message is processed just once (assuming no errors). In some systems, such as IoT, events must be ingested at very high volumes.
An event driven architecture can use a pub/sub model or an event stream model.

- Pub/sub: The messaging infrastructure keeps track of subscriptions. When an event is published, it sends the event to each subscriber. After an event is received, it cannot be replayed, and new subscribers do not see the event.
- Event streaming: Events are written to a log. Events are strictly ordered (within a partition) and durable. Clients don't subscribe to the stream, instead a client can read from any part of the stream. The client is responsible for advancing its position in the stream. That means a client can join at any time, and can replay events.

On the consumer side, there are some common variations:

- **Simple event processing**. An event immediately triggers an action in the consumer. For example, you could use Azure Functions with a Service Bus trigger, so that a function executes whenever a message is published to a Service Bus topic.

- **Complex event processing.** A consumer processes a series of events, looking for patterns in the event data, using a technology such as Azure Stream Analytics or Apache Storm. For example, you could aggregate readings from an embedded device over a time window, and generate a notification if the moving average crosses a certain threshold.
- **Event stream processing**. Use a data streaming platform, such as Azure IoT Hub or Apache Kafka, as a pipeline to ingest events and feed them to stream processors. The stream processors act to process or transform the stream. There may be multiple stream processors for different subsystems of the application. This approach is a good fit for IoT workloads.

The source of the events may be external to the system, such as physical devices in an IoT solution. In that case, the system must be able to ingest the data at the volume and throughput that is required by the data source.

In the logical diagram above, each type of consumer is shown as a single box. In practice, it's common to have multiple instances of a consumer, to avoid having the consumer become a single point of failure in system. Multiple instances might also be necessary to handle the volume and frequency of events. Also, a single consumer might process events on multiple threads. This can create challenges if events must be processed in order or require exactly-once semantics. See Minimize Coordination.

When to use this architecture

- Multiple subsystems must process the same events.
- Real-time processing with minimum time lag.
- Complex event processing, such as pattern matching or aggregation over time windows.
- High volume and high velocity of data, such as IoT.

Benefits

- Producers and consumers are decoupled.
- No point-to-point integrations. It's easy to add new consumers to the system.
- Consumers can respond to events immediately as they arrive.
- Highly scalable and distributed.
- Subsystems have independent views of the event stream.

Challenges

- Guaranteed delivery. In some systems, especially in IoT scenarios, it's crucial to guarantee that events are delivered.
- Processing events in order or exactly once. Each consumer type typically runs in multiple instances, for resiliency and scalability. This can create a challenge if the events must be processed in order (within a consumer type), or if the processing logic is not idempotent.

Additional considerations

- The amount of data to include in an event can be a significant consideration that affects both performance and cost. Putting all the relevant information needed for processing in the event itself can simplify the processing code and save additional lookups. Putting the minimal amount of information in an event, like just a couple of identifiers, will reduce transport time and cost, but requires the processing code to look

up any additional information it needs. For more information on this, take a look at this blog post.

From <https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven>

Event Hub

Friday, June 4, 2021 7:29 PM

The Capture feature allows you to store the **event stream** to an Azure Blob storage or Data Lake Storage. This way of storing events is reliable because even if the storage account isn't available, Capture keeps your data for a period, and then writes to the storage after it's available.

S**torage services can also offer additional features for analyzing events.** For example, by taking advantage of the access tiers of a blob storage account, you can store events in a hot tier for data that needs frequent access. You might use that data for visualization. Alternately, you can store data in the archive tier and retrieve it occasionally for auditing purposes.

C**apture stores all events ingested by Event Hubs and is useful for batch processing.** You can generate reports on the data by using a MapReduce function. Captured data can also serve as the source of truth. If certain facts were missed while aggregating the data, you can refer to the captured data.

For details about this feature, see Capture events through Azure Event Hubs in Azure Blob Storage or Azure Data Lake Storage.

Support for Apache Kafka clients
Event Hubs provides an endpoint for Apache Kafka clients. Existing clients can update their configuration to point to the endpoint and start sending events to Event Hubs. No code changes are required.

For more information, see Event Hubs for Apache Kafka.

Crossover scenarios
In some cases, it's advantageous to combine two messaging services.

Combining services can increase the efficiency of your messaging system. For instance, in your business transaction, you use Azure Service Bus queues to handle messages. Queues that are mostly idle and receive messages occasionally are inefficient because the consumer is constantly polling the queue for new messages. You can set up an Event Grid subscription with an Azure Function as the event handler. Each time the queue receives a message and there are no consumers listening, Event Grid sends a notification, which invokes the Azure Function that drains the queue.

Azure Service Bus to Event Grid integration

For details about connecting Service Bus to Event Grid, see Azure Service Bus to Event Grid integration overview.

The Enterprise integration on Azure using message queues and events reference architecture shows an implementation of Service Bus to Event Grid integration.

Here's another example. Event Grid receives a set of events in which some events require a workflow while others are for notification. The message metadata indicates the type of event. One way is to check the metadata by using the filtering feature in the event subscription. If it requires a

workflow, Event Grid sends it to Azure Service Bus queue. The receivers of that queue can take necessary actions. The notification events are sent to Logic Apps to send alert emails.

Event Grid

Friday, June 4, 2021 7:29 PM

Azure Event Grid is recommended for **discrete** events. Event Grid follows the Publisher-Subscriber pattern. When event sources trigger events, they are published to Event grid topics. Consumers of those events create Event Grid subscriptions by specifying event types and event handler that will process the events. If there are no subscribers, the events are discarded. Each event can have multiple subscriptions.

Push Model
Event Grid propagates messages to the subscribers in a push model. Suppose you have an event grid subscription with a webhook. When a new event arrives, Event Grid posts the event to the webhook endpoint.

Integrated with Azure
Choose Event Grid if you want to get notifications about Azure resources. Many Azure services act as event sources that have built-in Event Grid topics. Event Grid also supports various Azure services that can be configured as event handlers. It's easy to subscribe to those topics to route events to event handlers of your choice. For example, you can use Event Grid to invoke an Azure Function when a blob storage is created or deleted.

Custom topics
Create custom Event Grid topics, if you want to send events from your application or an Azure service that isn't integrated with Event Grid.

For example, to see the progress of an entire business transaction, you want the participating services to raise events as they are processing their individual business operations. A web app shows those events. One way is to create a custom topic and add a subscription with your web app registered through an HTTP WebHook. As business services send events to the custom topic, Event Grid pushes them to your web app.

Filtered events
You can specify filters in a subscription to instruct Event Grid to route only a subset of events to a specific event handler. The filters are specified in the subscription schema. Any event sent to the topic with values that match the filter are automatically forwarded to that subscription.

For example, content in various formats are uploaded to Blob Storage. Each time a file is added, an event is raised and published to Event Grid. The event subscription might have a filter that only sends events for images so that an event handler can generate thumbnails.

For more information about filtering, see Filter events for Event Grid.

High throughput
Event Grid can route 10,000,000 events per second per region. The first 100,000 operations per month are free. For cost considerations, see How much does Event Grid cost?

Resilient delivery
Even though successful delivery for events isn't as crucial as commands, you might still want some guarantee depending on the type of event. Event Grid offers features that you can enable and

customize, such as retry policies, expiration time, and dead lettering. For more information, see Delivery and retry.

Event Grid's retry process can help resiliency but it's not fail-safe. In the retry process, Event Grid might deliver the message more than once, skip, or delay some retries if the endpoint is unresponsive for a long time. For more information, see Retry schedule and duration.

You can persist undelivered events to a blob storage account by enabling dead-lettering. There's a delay in delivering the message to the blob storage endpoint and if that endpoint is unresponsive, then Event Grid discards the event. For more information, see Dead letter and retry policies.

Azure Event Hubs
When working with an event stream, Azure Event Hubs is the recommended message broker. Essentially, it's a large buffer that's capable of receiving large volumes of data with low latency. The received data can be read quickly through concurrent operations. You can transform the received data by using any real-time analytics provider. Event Hubs also provides the capability to store events in a storage account.

Fast ingestion
Event Hubs is capable of ingesting millions of events per second. The events are only appended to the stream and are ordered by time.

Pull model
Like Event Grid, Event Hubs also offers Publisher-Subscriber capabilities. A key difference between Event Grid and Event Hubs is in the way event data is made available to the subscribers. Event Grid pushes the ingested data to the subscribers whereas Event Hub makes the data available in a pull model. As events are received, Event Hubs appends them to the stream. A subscriber manages its cursor and can move forward and back in the stream, select a time offset, and replay a sequence at its pace.

Stream processors are subscribers that pull data from Event Hubs for the purposes of transformation and statistical analysis. Use Azure Stream Analytics and Apache Spark for complex processing such as aggregation over time windows or anomaly detection.

If you want to act on each event per partition, you can pull the data by using Event Processing Host or by using built in connector such as Logic Apps to provide the transformation logic. Another option is to use Azure Functions.

Partitioning
A partition is a portion of the event stream. The events are divided by using a partition key. For example, several IoT devices send device data to an event hub. The partition key is the device identifier. As events are ingested, Event Hubs moves them to separate partitions. Within each partition, all events are ordered by time.

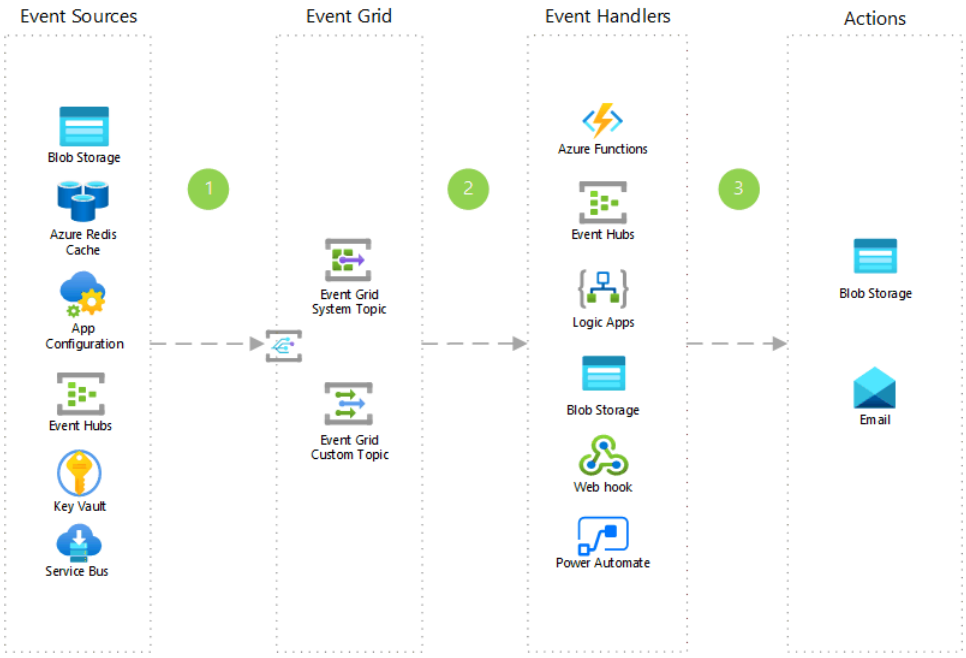
A consumer is an instance of code that processes the event data. Event Hubs follows a partitioned consumer pattern. Each consumer only reads a specific partition. Having multiple partitions results in faster processing because the stream can be read concurrently by multiple consumers.

Instances of the same consumer make up a single consumer group. Multiple consumer groups can read the same stream with different intentions. Suppose an event stream has data from a temperature sensor. One consumer group can read the stream to detect anomalies such as a spike in temperature. Another can read the same stream to calculate a rolling average temperature in a temporal window.

Event Hubs supports the Publisher-Subscriber pattern by allowing multiple consumer groups. Each consumer group is a subscriber.

For more information about Event Hub partitioning, see Partitions.

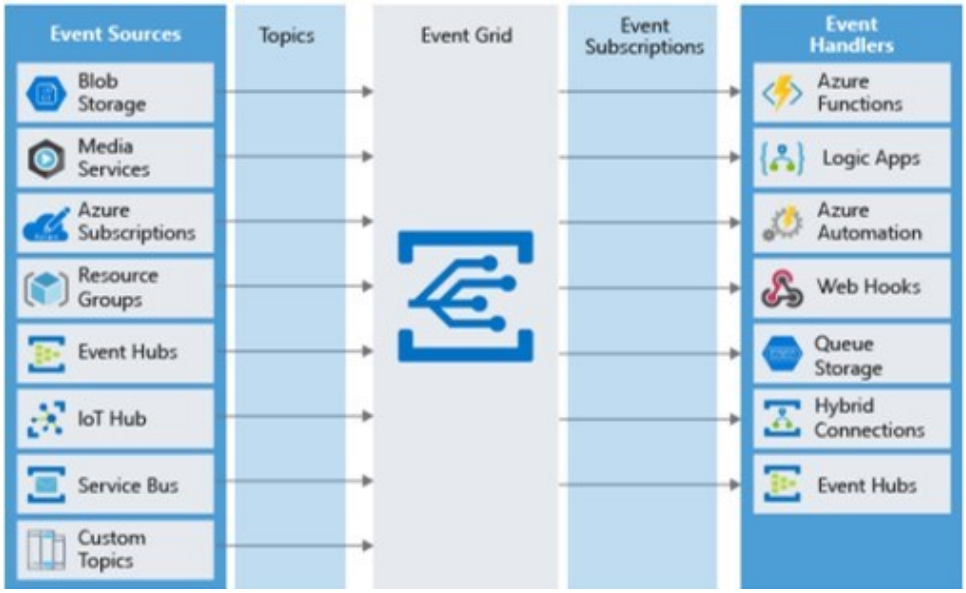
A topic is used for a collection of related events. To respond to certain types of events, subscribers decide which topics to subscribe to.



Event Grid - Good

Friday, June 4, 2021 7:44 PM

https://www.cognizantsoftvision.com/blog/azure-event-grid-vs-event-hubs/



- **Events** → What happened
  The Event is a lightweight notification of a condition or a state change, which means that it doesn't contain the entire object that was changed. The sender of the event is known as ***publisher*** *and the receiver is known as* ***subscriber*** *.*

- **Event Sources** → Where the event took place
  The Event source (publisher) is responsible for sending events to the Event Grid. Each event source is related to one or more event types (for example, Azure Storage is the event source for blob created events).

- **Topics** → The endpoint where publishers send events
  The Event Topic categorizes the events into groups. The Event source sends events to the topics, using a public endpoint.

  There are 2 types of topics:
  1. ***System topics*** *are built-in topics provided by Azure services. We don't see system*

topics in our Azure subscription because the publisher owns the topics, but we can subscribe to them.

2. ***Custom topics*** *are application and third-party topics. We can see custom topics in our subscription if we create or we are assigned access to them.*

- **Event subscriptions** → The endpoint or built-in mechanism to route events, sometimes to multiple handlers

  Event subscription defines which events on a topic an event handler wants to receive. Subscription can also filter events based on their type or subject, so we can ensure that an event handler receives only relevant events.

- **Event handlers** → The app or service reacting to the event

  Event handler (subscriber) is any component that can receive events from Event Grid.

Now we know what Event Grid is and how it can help us. Let's see when we can use it.

For example, we have a blob storage and we want to execute an azure function each time a file is uploaded.

We can do this easily with Azure function:

Azure function (Event Grid Trigger)

1. Create an Event Grid Trigger function.
2. Go to the storage account → events and add an event subscription.
3. At the endpoint **type**, we need to select Azure Function and at the endpoint **field** to select the event grid trigger function that we created earlier.
4. That's it. Once we upload a file to the storage, the event trigger function will be executed.

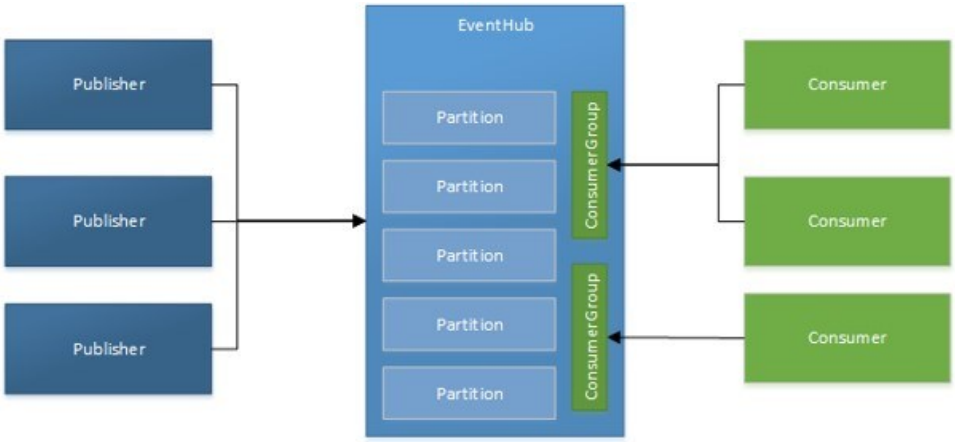
On the other hand, if we want to publish events to an **Event Grid Topic**, in order to accomplish this we can use three different approaches:

- Event Grid SDK
- Sending an HTTP POST with an authentication header
- Durable Functions

From <https://www.cognizantsoftvision.com/blog/azure-event-grid-vs-event-hubs/>

Event Hub - Good

Tuesday, July 6, 2021 7:06 PM



Event Hubs are an intermediary for the publish-subscribe communication pattern. Unlike Event Grid, it **is a service for processing huge amounts of events** (millions of events per second) **with low latency**. We should consider the Event Hubs as the starting point in an event processing pipeline. Furthermore, we can use the Event Hubs as the event source of the Event Grid service.

Partitions

As we can see in the image above, an Event Hub contains multiple partitions. Let's explain what these partitions are, and why an Event Hub contains them. Event Hub receives data and it divides it into partitions. Partitions are buffers into which the data is saved. Because of these buffers, an event isn't missed just because a subscriber is busy or even offline. The subscriber can always use this buffer to get the events. By default, events stay in the buffer for 24 hours before they automatically expire.

These buffers are called partitions because the data is divided amongst them. Every event hub has at least two partitions, and each partition has a separate set of subscribers.

Advantages

One big advantage of Event Hubs is that it can provide a Kafka endpoint that can be used by our existing Kafka with a small configuration change. The big difference between Kafka and Event Hubs is that Event Hubs is a cloud service. There is no need to manage servers or networks.

Another advantage is that it can be used for logging and telemetry. Moreover, it can be integrated with the serverless real-time analytics, Stream Analytics and the business analytics service, Power BI.

Based on the advantages above, we can also understand Event Hubs usage.

From <https://www.cognizantsoftvision.com/blog/azure-event-grid-vs-event-hubs/>

From <https://www.cognizantsoftvision.com/blog/azure-event-grid-vs-event-hubs/>

Event Grid Vs Event Hub

Tuesday, July 6, 2021 7:11 PM

The noticeable difference between them is that Event Hubs are accepting only endpoints for the ingestion of data and they don't provide a mechanism for sending data back to publishers. On the other hand, Event Grid sends HTTP requests to notify events that happen in publishers.

Event Grid can trigger an Azure Function. In the case of Event Hubs, the Azure Function needs to pull and process an event.

Another difference is durability. Event Grid is a distribution system, not a queueing mechanism. If an event is pushed in, it gets pushed out immediately and if it doesn't get handled, it's gone forever. Unless we send the undelivered events to a storage account. This process is known as dead-lettering. By default this option is disabled, so if we need to enable it, we have to specify a storage account at the creation of the Event Grid.

In Event Hubs, publishers and subscribers read and write from durable storage. The data can be kept in the Event Hubs for up to seven days and then replayed. This gives us the ability to resume from a certain point or to restart from an older point in time and reprocess events when we need it.

Moreover, Event Grid doesn't guarantee the order of the events. Contrarily, Event Hubs use partitions as we discussed earlier, and these partitions are ordered sequences, so it can maintain the order of the events in the same partition.

Conclusion

Azure Event Hubs is a more suitable solution when we need a service that can receive and process millions of events per second and provide low-latency event processing. It can handle data from concurrent sources and route it to a variety of stream-processing infrastructure and analytics services, as I have already mentioned. Azure Event Hubs are used more for telemetry scenarios.

On the other hand, Azure Event Grid is ideal for reactive scenarios, like when an item has been shipped or an item has been added or updated on storage. We have to take into account also its native integrations with Functions, Logic Apps and Webhooks. Moreover, Event Grid is cheaper than Event Hubs and more suitable when we don’t have to deal with big data.

From <https://www.cognizantsoftvision.com/blog/azure-event-grid-vs-event-hubs/>

Message Based

Friday, June 4, 2021 7:25 PM

The producer sends a command with the intent that the consumer(s) will perform an operation within the scope of a business transaction.

A command is a high-value message and must be delivered at least once. If a command is lost, the entire business transaction might fail. Also, a command shouldn't be processed more than once. Doing so might cause an erroneous transaction. A customer might get duplicate orders or billed twice.

Commands are often used to manage the workflow of a multistep business transaction. Depending on the business logic, the producer may expect the consumer to acknowledge the message and report the results of the operation. Based on that result, the producer may choose an appropriate course of action.

Azure Service Bus

Friday, June 4, 2021 7:30 PM

Azure Service Bus queues are well suited for transferring commands from producers to consumers. Here are some considerations.

Pull model
A consumer of a Service Bus queue constantly polls Service Bus to check if new messages are available. The client SDKs and Azure Functions trigger for Service Bus abstract that model. When a new message is available, the consumer's callback is invoked, and the message is sent to the consumer.

Guaranteed delivery
Service Bus allows a consumer to peek the queue and lock a message from other consumers.

It's the responsibility of the consumer to report the processing status of the message. Only when the consumer marks the message as consumed, Service Bus removes the message from the queue. If a failure, timeout, or crash occurs, Service Bus unlocks the message so that other consumers can retrieve it. This way messages aren't lost in transfer.

A producer might accidentally send the same message twice. For instance, a producer instance fails after sending a message. Another producer replaces the original instance and sends the message again. Azure Service Bus queues provide a built-in de-duping capability that detects and removes duplicate messages. There's still a chance that a message is delivered twice. For example, if a consumer fails while processing, the message is returned to the queue and is retrieved by the same or another consumer. The message processing logic in the consumer should be idempotent so that even if the work is repeated, the state of the system isn't changed.

Message ordering
If you want consumers to get the messages in the order they are sent, Service Bus queues guarantee first-in-first-out (FIFO) ordered delivery by using sessions. A session can have one or more messages. The messages are correlated with the SessionId property. Messages that are part of a session, never expire. A session can be locked to a consumer to prevent its messages from being handled by a different consumer.

For more information, see Message Sessions.

Message persistence
Service bus queues support temporal decoupling. Even when a consumer isn't available or unable to process the message, it remains in the queue.

Checkpoint long-running transactions
Business transactions can run for a long time. Each operation in the transaction can have multiple messages. Use checkpointing to coordinate the workflow and provide resiliency in case a transaction fails.

Service Bus queues allow checkpointing through the session state capability. State information is incrementally recorded in the queue (SetState) for messages that belong to a session. For example, a consumer can track progress by checking the state (GetState) every now and then. If a consumer fails, another consumer can use state information to determine the last known checkpoint to resume the

session.

Dead-letter queue (DLQ)
A Service Bus queue has a default subqueue, called the dead-letter queue (DLQ) to hold messages that couldn't be delivered or processed. Service Bus or the message processing logic in the consumer can add messages to the DLQ. The DLQ keeps the messages until they are retrieved from the queue.

Here are examples when a message can end up being in the DLQ:

A poison message is a message that cannot be handled because it's malformed or contains unexpected information. In Service Bus queues, you can detect poison messages by setting the MaxDeliveryCount property of the queue. If number of times the same message is received exceeds that property value, Service Bus moves the message to the DLQ.

A message might no longer be relevant if it isn't processed within a period. Service Bus queues allow the producer to post messages with a time-to-live attribute. If this period expires before the message is received, the message is placed in the DLQ.

Examine messages in the DLQ to determine the reason for failure.

Hybrid solution
Service Bus bridges on-premises systems and cloud solutions. On-premises systems are often difficult to reach because of firewall restrictions. Both the producer and consumer (either can be on-premises or the cloud) can use the Service Bus queue endpoint as the pickup and drop off location for messages.

Topics and subscriptions
Service Bus supports the Publisher-Subscriber pattern through Service Bus topics and subscriptions.

This feature provides a way for the producer to broadcast messages to multiple consumers. When a topic receives a message, it's forwarded to all the subscribed consumers. Optionally, a subscription can have filter criteria that allows the consumer to get a subset of messages. Each consumer retrieves messages from a subscription in a similar way to a queue.

For more information, see Azure Service Bus topics.

Choose between Azure messaging services - Event Grid, Event Hubs, and Service Bus

Tuesday, July 6, 2021 6:53 PM

https://docs.microsoft.com/en-us/azure/event-grid/compare-messaging-services

Azure offers three services that assist with delivering events or messages throughout a solution. These services are:

Azure Event Grid

- Dynamically scalable
- Low cost
- Serverless
- At least once delivery of an event

Azure Event Hubs

- Low latency
- Can receive and process millions of events per second
- At least once delivery of an event

Azure Service Bus

- Reliable asynchronous message delivery (enterprise messaging as a service) that requires polling
- Advanced messaging features like first-in and first-out (FIFO), batching/sessions, transactions, dead-lettering, temporal control, routing and filtering, and duplicate detection
- At least once delivery of a message
- Optional ordered delivery of messages

omparison of services

Service	Purpose	Type	When to use
Event Grid	Reactive programming	Event distribution (discrete)	React to status changes
Event Hubs	Big data pipeline	Event streaming (series)	Telemetry and distributed data streaming
Service Bus	High-value enterprise messaging	Message	Order processing and financial transactions

COMPARISON OF SERVICES

From <https://docs.microsoft.com/en-us/azure/event-grid/compare-messaging-services>

Azure Data Studio

Friday, July 16, 2021 7:39 PM