100 XP

Introduction

3 minutes

Azure Event Hubs is a big data streaming platform and event ingestion service. It can receive and process millions of events per second. Data sent to an event hub can be transformed and stored by using any real-time analytics provider or batching/storage adapters.

After completing this module, you'll be able to:

- Describe the benefits of using Event Hubs and how it captures streaming data.
- Explain how to process events.
- Perform common operations with the Event Hubs client library.

Next unit: Discover Azure Event Hubs

Continue

100 XP

Discover Azure Event Hubs

3 minutes

Azure Event Hubs represents the "front door" for an event pipeline, often called an *event ingestor* in solution architectures. An event ingestor is a component or service that sits between event publishers and event consumers to decouple the production of an event stream from the consumption of those events. Event Hubs provides a unified streaming platform with time retention buffer, decoupling event producers from event consumers.

The table below highlights key features of the Azure Event Hubs service:

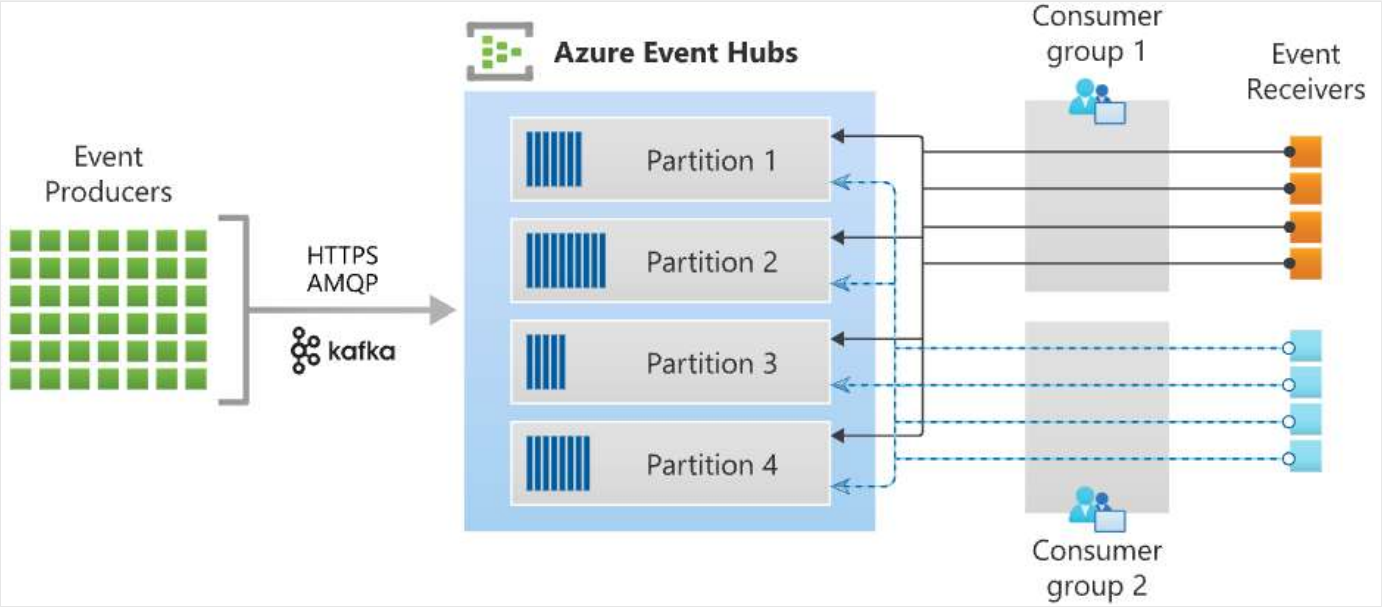
Feature	Description
Fully managed PaaS	Event Hubs is a fully managed Platform-as-a-Service (PaaS) with little configuration or management overhead, so you focus on your business solutions. Event Hubs for Apache Kafka ecosystems gives you the PaaS Kafka experience without having to manage, configure, or run your clusters.
Real-time and batch processing	Event Hubs uses a partitioned consumer model, enabling multiple applications to process the stream concurrently and letting you control the speed of processing.
Scalable	Scaling options, like Auto-inflate, scale the number of throughput units to meet your usage needs.
Rich ecosystem	Event Hubs for Apache Kafka ecosystems enables Apache Kafka (1.0 and later) clients and applications to talk to Event Hubs. You do not need to set up, configure, and manage your own Kafka clusters.

Key concepts

Event Hubs contains the following 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.

The following figure shows the Event Hubs stream processing architecture:



Next unit: Explore Event Hubs Capture

Continue >

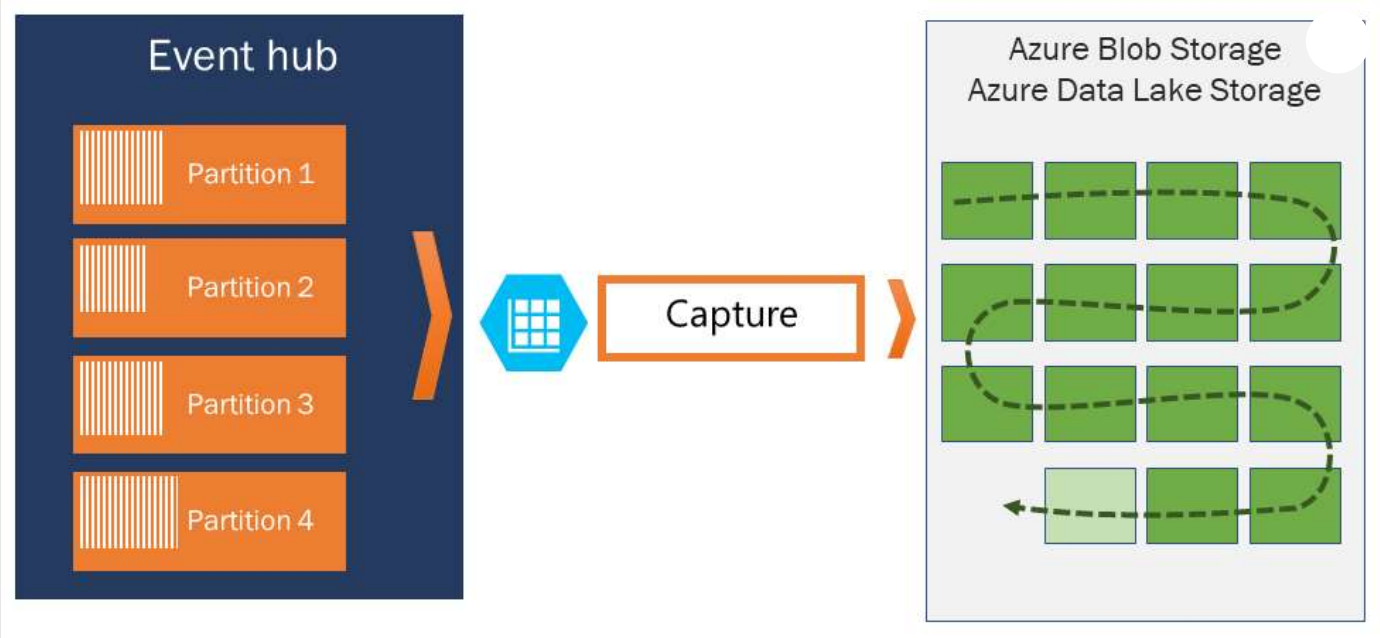
How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Explore Event Hubs Capture

3 minutes

Azure Event Hubs enables you to automatically capture the streaming data in Event Hubs in an Azure Blob storage or Azure Data Lake Storage account of your choice, with the added flexibility of specifying a time or size interval. Setting up Capture is fast, there are no administrative costs to run it, and it scales automatically with Event Hubs throughput units in the standard tier or processing units in the premium tier.



Event Hubs Capture enables you to process real-time and batch-based pipelines on the same stream. This means you can build solutions that grow with your needs over time.

How Event Hubs Capture works

Event Hubs is a time-retention durable buffer for telemetry ingress, similar to a distributed log. The key to scaling in Event Hubs is the partitioned consumer model. Each partition is an independent segment of data and is consumed independently. Over time this data ages off, based on the configurable retention period. As a result, a given event hub never gets "too full."

Event Hubs Capture enables you to specify your own Azure Blob storage account and container, or Azure Data Lake Store account, which are used to store the captured data. These accounts can be in the same region as your event hub or in another region, adding to the flexibility of the Event Hubs Capture feature.

Captured data is written in Apache Avro format: a compact, fast, binary format that provides rich data structures with inline schema. This format is widely used in the Hadoop ecosystem, Stream Analytics, and Azure Data Factory. More information about working with Avro is available later in this article.

Capture windowing

Event Hubs Capture enables you to set up a window to control capturing. This window is a minimum size and time configuration with a "first wins policy," meaning that the first trigger encountered causes a capture operation. Each partition captures independently and writes a completed block blob at the time of capture, named for the time at which the capture interval was encountered. The storage naming convention is as follows:

Copy

```
{Namespace}/{EventHub}/{PartitionId}/{Year}/{Month}/{Day}/{Hour}/{Minute}/{Second}
```

Note that the date values are padded with zeroes; an example filename might be:

Copy

```
https://mystorageaccount.blob.core.windows.net/mycontainer/mynamespace/myeventhub/0/2017/12/08/03/03/17.avro
```

Scaling to throughput units

Event Hubs traffic is controlled by throughput units. A single throughput unit allows 1 MB per second or 1000 events per second of ingress and twice that amount of egress. Standard Event Hubs can be configured with 1-20 throughput units, and you can purchase more with a quota increase support request. Usage beyond your purchased throughput units is throttled. Event Hubs Capture copies data directly from the internal Event Hubs storage, bypassing throughput unit egress quotas and saving your egress for other processing readers, such as Stream Analytics or Spark.

Once configured, Event Hubs Capture runs automatically when you send your first event, and continues running. To make it easier for your downstream processing to know that the process is working, Event Hubs writes empty files when there is no data. This process provides a predictable cadence and marker that can feed your batch processors.

Next unit: Scale your processing application

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Scale your processing application

3 minutes

To scale your event processing application, you can run multiple instances of the application and have it balance the load among themselves. In the older versions, **EventProcessorHost** allowed you to balance the load between multiple instances of your program and checkpoint events when receiving. In the newer versions (5.0 onwards), **EventProcessorClient** (.NET and Java), or **EventHubConsumerClient** (Python and JavaScript) allows you to do the same.

> **Note**
>
> The key to scale for Event Hubs is the idea of partitioned consumers. In contrast to the competing consumers pattern, the partitioned consumer pattern enables high scale by removing the contention bottleneck and facilitating end to end parallelism.

Example scenario

As an example scenario, consider a home security company that monitors 100,000 homes. Every minute, it gets data from various sensors such as a motion detector, door/window open sensor, glass break detector, and so on, installed in each home. The company provides a web site for residents to monitor the activity of their home in near real time.

Each sensor pushes data to an event hub. The event hub is configured with 16 partitions. On the consuming end, you need a mechanism that can read these events, consolidate them, and dump the aggregate to a storage blob, which is then projected to a user-friendly web page.

When designing the consumer in a distributed environment, the scenario must handle the following requirements:

- **Scale:** Create multiple consumers, with each consumer taking ownership of reading from a few Event Hubs partitions.
- **Load balance:** Increase or reduce the consumers dynamically. For example, when a new sensor type (for example, a carbon monoxide detector) is added to each home, the number

of events increases. In that case, the operator (a human) increases the number of consumer instances. Then, the pool of consumers can rebalance the number of partitions they own, to share the load with the newly added consumers.

- **Seamless resume on failures:** If a consumer (**consumer A**) fails (for example, the virtual machine hosting the consumer suddenly crashes), then other consumers can pick up the partitions owned by **consumer A** and continue. Also, the continuation point, called a *checkpoint* or *offset*, should be at the exact point at which **consumer A** failed, or slightly before that.
- **Consume events:** While the previous three points deal with the management of the consumer, there must be code to consume the events and do something useful with it. For example, aggregate it and upload it to blob storage.

Event processor or consumer client

You don't need to build your own solution to meet these requirements. The Azure Event Hubs SDKs provide this functionality. In .NET or Java SDKs, you use an event processor client (`EventProcessorClient`), and in Python and JavaScript SDKs, you use `EventHubConsumerClient`.

For most production scenarios, we recommend that you use the event processor client for reading and 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.

Partition ownership tracking

An event processor instance typically owns and processes events from one or more partitions. Ownership of partitions is evenly distributed among all the active event processor instances associated with an event hub and consumer group combination.

Each event processor is given a unique identifier and claims ownership of partitions by adding or updating an entry in a checkpoint store. All event processor instances communicate with this store periodically to update its own processing state as well as to learn about other active instances. This data is then used to balance the load among the active processors.

Receive messages

When you create an event processor, you specify the functions that will process events and errors. Each call to the function that processes events delivers a single event from a specific partition. It's your responsibility to handle this event. If you want to make sure the consumer processes every message at least once, you need to write your own code with retry logic. But be cautious about poisoned messages.

We recommend that you do things relatively fast. That is, do as little processing as possible. If you need to write to storage and do some routing, it's better to use two consumer groups and have two event processors.

Checkpointing

Checkpointing is a process by which an event processor marks or commits the position of the last successfully processed event within a partition. Marking a checkpoint is typically done within the function that processes the events and occurs on a per-partition basis within a consumer group.

If an event processor disconnects from a partition, another instance can resume processing the partition at the checkpoint that was previously committed by the last processor of that partition in that consumer group. When the processor connects, it passes the offset to the event hub to specify the location at which to start reading. In this way, you can use checkpointing to both mark events as "complete" by downstream applications and to provide resiliency when an event processor goes down. It's possible to return to older data by specifying a lower offset from this checkpointing process.

Thread safety and processor instances

By default, the function that processes the events is called sequentially for a given partition. Subsequent events and calls to this function from the same partition queue up behind the scenes as the event pump continues to run in the background on other threads. Events from different partitions can be processed concurrently and any shared state that is accessed across partitions have to be synchronized.

Next unit: Control access to events

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

Control access to events

3 minutes

Azure Event Hubs supports both Azure Active Directory and shared access signatures (SAS) to handle both authentication and authorization. Azure provides the following Azure built-in roles for authorizing access to Event Hubs data using Azure Active Directory and OAuth:

- 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 identities

To authorize a request to Event Hubs service from a managed identity in your application, you need to configure Azure role-based access control settings for that managed identity. Azure Event Hubs defines Azure roles that encompass permissions for sending and reading from Event Hubs. When the Azure role is assigned to a managed identity, the managed identity is granted access to Event Hubs data at the appropriate scope.

Authorize access with Microsoft Identity Platform

A key advantage of using Azure AD with Event Hubs is that your credentials no longer need to be stored in your code. Instead, you can request an OAuth 2.0 access token from Microsoft identity platform. Azure AD authenticates the security principal (a user, a group, or service principal) running the application. If authentication succeeds, Azure AD returns the access token to the application, and the application can then use the access token to authorize requests to Azure Event Hubs.

Authorize access to Event Hubs publishers with shared access signatures

An event publisher defines a virtual endpoint for an Event Hub. The publisher can only be used to send messages to an event hub and not receive messages. Typically, an event hub employs one publisher per client. All messages that are sent to any of the publishers of an event hub are enqueued within that event hub. Publishers enable fine-grained access control.

Each Event Hubs client is assigned a unique token which is uploaded to the client. A client that holds a token can only send to one publisher, and no other publisher. If multiple clients share the same token, then each of them shares the publisher.

All tokens are assigned with shared access signature keys. Typically, all tokens are signed with the same key. Clients aren't aware of the key, which prevents clients from manufacturing tokens. Clients operate on the same tokens until they expire.

Authorize access to Event Hubs consumers with shared access signatures

To authenticate back-end applications that consume from the data generated by Event Hubs producers, Event Hubs token authentication requires its clients to either have the **manage** rights or the **listen** privileges assigned to its Event Hubs namespace or event hub instance or topic. Data is consumed from Event Hubs using consumer groups. While SAS policy gives you granular scope, this scope is defined only at the entity level and not at the consumer level. It means that the privileges defined at the namespace level or the event hub instance or topic level will be applied to the consumer groups of that entity.

Next unit: Perform common operations with the Event Hubs client library

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Perform common operations with the Event Hubs client library

3 minutes

This unit contains examples of common operations you can perform with the Event Hubs client library (`Azure.Messaging.EventHubs`) to interact with an Event Hub.

Inspect an Event Hub

Many Event Hub operations take place within the scope of a specific partition. Because partitions are owned by the Event Hub, their names are assigned at the time of creation. To understand what partitions are available, you query the Event Hub using one of the Event Hub clients. For illustration, the `EventHubProducerClient` is demonstrated in these examples, but the concept and form are common across clients.

C# Copy

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

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

Publish events to an Event Hub

In order to publish events, you'll need to create an `EventHubProducerClient`. Producers publish events in batches and may request a specific partition, or allow the Event Hubs service to decide which partition events should be published to. We recommended using automatic routing when the publishing of events needs to be highly available or when event data should be distributed evenly among the partitions. Our example will take advantage of automatic routing.

C# Copy

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

await using (var producer = new EventHubProducerClient(connectionString, eventHub-
Name))
{
    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

In order to read events from an Event Hub, you'll need to create an `EventHubConsumerClient` for a given consumer group. When an Event Hub is created, it provides a default consumer group that can be used to get started with exploring Event Hubs. In our example, we will focus on reading all events that have been published to the Event Hub using an iterator.

> **Note**
>
> It is important to note that this approach to consuming is intended to improve the experience of exploring the Event Hubs client library and prototyping. It is recommended that it not be used in production scenarios. For production use, we recommend using the **Event Processor Client** , as it provides a more robust and performant experience.

C# Copy

```
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, connection-
String, 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 cancella-
tion is requested using
        // the cancellation token.
    }
}
```

Read events from an Event Hub partition

To read from a specific partition, the consumer will need to specify where in the event stream to begin receiving events; in our example, we will focus on reading all published events for the first partition of the Event Hub.

C# Copy

```
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, connection-
String, 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 parti-
tion.  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 cancella-
tion is requested using
        // the cancellation token.
    }
}
```

Process events using an Event Processor client

For most production scenarios, it is recommended that the `EventProcessorClient` be used for reading and processing events. Since the `EventProcessorClient` has a dependency on Azure Storage blobs for persistence of its state, you'll need to provide a `BlobContainerClient` for the processor, which has been configured for the storage account and container that should be used.

C# Copy

```
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, blobContainer-
Name);
var processor = new EventProcessorClient(storageClient, consumerGroup, eventHubsCon-
nectionString, 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;
}
```

Next unit: Knowledge check

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

200 XP

Knowledge check

3 minutes

Check your knowledge

1. Which of the following Event Hubs concepts represents an ordered sequence of events that is held in an Event Hub?

- ○ Consumer group
- ◉ Partition

  ✔ That's correct. A partition is an ordered sequence of events that is held in an Event Hub.
- ○ Event Hub producer

2. Which of the following represents when an event processor marks or commits the position of the last successfully processed event within a partition?

- ◉ Checkpointing

  ✔ That's correct. Checkpointing is a process by which an event processor marks or commits the position of the last successfully processed event within a partition.
- ○ Scale
- ○ Load balance

Next unit: Summary

Continue

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Summary

3 minutes

In this module, you learned how to:

- Describe the benefits of using Event Hubs and how it captures streaming data.
- Explain how to process events.
- Perform common operations with the Event Hubs client library.

Module complete:

Unlock achievement

How are we doing? ☆ ☆ ☆ ☆ ☆