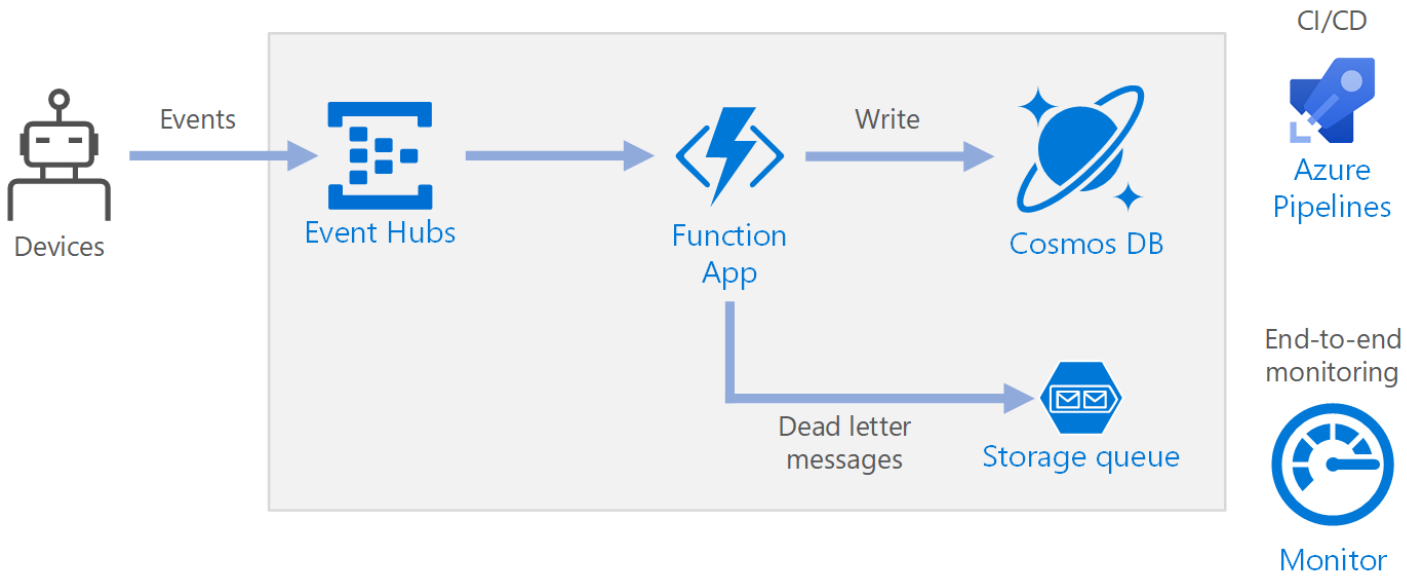
Serverless event processing using Azure Functions

10/16/2018 • 5 minutes to read • Contributors

In this article

This reference architecture shows a serverless, event-driven architecture that ingests a stream of data, processes the data, and writes the results to a back-end database.

A reference implementation for this architecture is available on GitHub.



Architecture

Event Hubs ingests the data stream. Event Hubs is designed for high-throughput data streaming scenarios.

> **Note**
>
> For IoT scenarios, we recommend IoT Hub. IoT Hub has a built-in endpoint that's compatible with the Azure Event Hubs API, so you can use either service in this architecture with no major changes in the backend processing. For more information, see **Connecting IoT Devices to Azure: IoT Hub and Event Hubs**.

Function App. Azure Functions is a serverless compute option. It uses an event-driven model, where a piece of code (a "function") is invoked by a trigger. In this architecture, when events arrive at Event Hubs, they trigger a function that processes the events and writes the results to storage.

Function Apps are suitable for processing individual records from Event Hubs. For more complex stream processing scenarios, consider Apache Spark using Azure Databricks, or Azure Stream Analytics.

Cosmos DB. Cosmos DB is a multi-model database service. For this scenario, the event-processing function stores JSON records, using the Cosmos DB SQL API.

Queue storage. Queue storage is used for dead letter messages. If an error occurs while processing an event, the function stores the event data in a dead letter queue for later processing. For more information, see Resiliency Considerations.

Azure Monitor. Monitor collects performance metrics about the Azure services deployed in the solution. By visualizing these in a dashboard, you can get visibility into the health of the solution.

Azure Pipelines. Pipelines is a continuous integration (CI) and continuous delivery (CD) service that builds, tests, and deploys the application.

Scalability considerations

Event Hubs

The throughput capacity of Event Hubs is measured in throughput units. You can autoscale an event hub by enabling auto-inflate, which automatically scales the throughput units based on traffic, up to a configured maximum.

The Event Hub trigger in the function app scales according to the number of partitions in the event hub. Each partition is assigned one function instance at a time. To maximize throughput, receive the events in a batch, instead of one at a time.

Cosmos DB

Throughput capacity for Cosmos DB is measured in Request Units (RU). In order to scale a Cosmos DB container past 10,000 RU, you must specify a partition key when you create the container, and include the partition key in every document that you create.

Here are some characteristics of a good partition key:

- The key value space is large.
- There will be an even distribution of reads/writes per key value, avoiding hot keys.
- The maximum data stored for any single key value will not exceed the maximum physical partition size (10 GB).
- The partition key for a document won't change. You can't update the partition key on an existing document.

In the scenario for this reference architecture, the function stores exactly one document per device that is sending data. The function continually updates the documents with latest device status, using an upsert operation. Device ID is a good partition key for this scenario, because writes will be evenly distributed across the keys, and the size of each partition will be strictly bounded, because there is a single document for each key value. For more information about partition keys, see Partition and scale in Azure Cosmos DB.

Resiliency considerations

When using the Event Hubs trigger with Functions, catch exceptions within your processing loop. If an unhandled exception occurs, the Functions runtime does not retry the messages. If a message cannot be processed, put the message into a dead letter queue. Use an out-of-band process to examine the messages and determine corrective action.

The following code shows how the ingestion function catches exceptions and puts unprocessed messages onto a dead letter queue.

C# Copy

```
[FunctionName("RawTelemetryFunction")]
[StorageAccount("DeadLetterStorage")]
public static async Task RunAsync(
    [EventHubTrigger("%EventHubName%", Connection = "EventHubConnection", ConsumerGroup
="%EventHubConsumerGroup%")]EventData[] messages,
    [Queue("deadletterqueue")] IAsyncCollector<DeadLetterMessage> deadLetterMessages,
    ILogger logger)
{
    foreach (var message in messages)
    {
        DeviceState deviceState = null;

        try
        {
            deviceState = telemetryProcessor.Deserialize(message.Body.Array, logger);
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Error deserializing message", message.SystemProperties.Parti-
tionKey, message.SystemProperties.SequenceNumber);
            await deadLetterMessages.AddAsync(new DeadLetterMessage { Issue = ex.Message,
EventData = message });
        }

        try
        {
            await stateChangeProcessor.UpdateState(deviceState, logger);
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Error updating status document", deviceState);
            await deadLetterMessages.AddAsync(new DeadLetterMessage { Issue = ex.Message,
EventData = message, DeviceState = deviceState });
        }
    }
}
```

Notice that the function uses the Queue storage output binding to put items in the queue.

The code shown above also logs exceptions to Application Insights. You can use the partition key and sequence number to correlate dead letter messages with the exceptions in the logs.

Messages in the dead letter queue should have enough information so that you can understand the context of error. In this example, the `DeadLetterMessage` class contains the exception message, the original event data, and the deserialized event message (if available).

C# Copy

```
public class DeadLetterMessage
{
    public string Issue { get; set; }
    public EventData EventData { get; set; }
    public DeviceState DeviceState { get; set; }
}
```

Use Azure Monitor to monitor the event hub. If you see there is input but no output, it means that messages are not being processed. In that case, go into Log Analytics and look for exceptions or other errors.

Disaster recovery considerations

The deployment shown here resides in a single Azure region. For a more resilient approach to disaster-recovery, take advantage of geo-distribution features in the various services:

- **Event Hubs**. Create two Event Hubs namespaces, a primary (active) namespace and a secondary (passive) namespace. Messages are automatically routed to the active namespace unless you fail over to the secondary namespace. For more information, see Azure Event Hubs Geo-disaster recovery.
- **Function App**. Deploy a second function app that is waiting to read from the secondary Event Hubs namespace. This function writes to a secondary storage account for dead letter queue.
- **Cosmos DB**. Cosmos DB supports multiple master regions, which enables writes to any region that you add to your Cosmos DB account. If you don't enable multi-master, you can still fail over the primary write region. The Cosmos DB client SDKs and the Azure Function bindings automatically handle the failover, so you don't need to update any application configuration settings.
- **Azure Storage**. Use RA-GRS storage for the dead letter queue. This creates a read-only replica in another region. If the primary region becomes unavailable, you can read the items currently in the queue. In addition, provision another storage account in the secondary region that the function can write to after a fail-over.

Deploy the solution

To deploy this reference architecture, view the GitHub readme.

Next steps

To learn more about the reference implementation, read Show me the code: Serverless application with Azure Functions.