Microsoft

Integrate Azure Cosmos DB for NoSQL with Azure services

Agenda

- Consume an Azure Cosmos DB for NoSQL change feed using the SDK
- Handle events with Azure Functions and Azure Cosmos DB for NoSQL change feed
- Search Azure Cosmos DB for NoSQL data with Azure Cognitive Search

Consume an Azure Cosmos DB for NoSQL change feed using the SDK

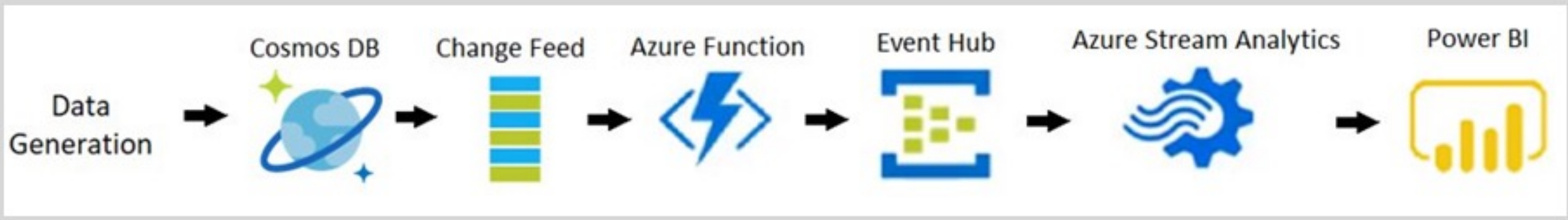
Understand change feed features in the SDK

The .NET SDK for Azure Cosmos DB for NoSQL ships with a change feed processor that simplifies the task of reading changes from the feed.

Change feed processor core components

Monitored container	Lease container	Host	Delegate
This container is monitored for any insert or update operations. These changes are then reflected in the feed.	The lease container serves as a storage mechanism to manage state across multiple change feed consumers (clients).	The host is a client application instance that listens for and reacts to changes from the change feed.	The delegate is code within the client application that will implement business logic for each batch of changes.

예시



1.Azure Cosmos DB: 생성된 데이터는 Azure Cosmos DB 컨테이너에 저장됩니다.
2.변경 피드: Azure Cosmos DB 컨테이너의 변경 내용을 수신 대기합니다. 새 문서가 컬렉션에 추가될 때마다, 즉 사용자가 항목을 조회하거나 자신의 카트에 항목을 추가하거나 항목을 구입하는 것과 같은 이벤트가 발생하면 변경 피드에서 Azure Function을 트리거합니다.
3.Azure Function: 새 데이터를 처리하여 Azure Event Hubs로 보냅니다.
4.Azure Event Hubs: 이러한 이벤트를 저장하고 추가 분석을 수행하기 위해 Azure Stream Analytics로 보냅니다.
5.Azure Stream Analytics: 이벤트를 처리하고 실시간 데이터 분석을 수행하기 위한 쿼리를 정의합니다. 그런 다음, 이 데이터는 Microsoft Power BI로 보내집니다.

https://learn.microsoft.com/ko-kr/azure/cosmos-db/nosql/changefeed-ecommerce-solution

Implement a delegate for the change feed processor

For the change feed processor, the library expects a delegate of type *ChangesHandler<>*.

This delegate will handle a list of changes on the feed.

```
// The delegate includes two parameters; a read-only list of changes and a cancellation token.

ChangesHandler<Product> changeHandlerDelegate = async (
    IReadOnlyCollection<Product> changes,
    CancellationToken cancellationToken ) =>
        {
            // A foreach loop is used to iterate through the current batch of changes,
            // and print each item to the console window
            foreach(Product product in changes)
            {
                await Console.Out.WriteLineAsync($"Detected Operation:\t[{product.id}]\t{product.name}");
                // Do something else with each change
            }
         };
```

Implement the change feed processor

The processor needs to be built with a source and lease container, and then needs to be started.

```
// Create an instance for both the source and lease container.
Container sourceContainer = client.GetContainer("cosmicworks", "products");
Container leaseContainer = client.GetContainer("cosmicworks", "productslease");

// Use the GetChangeFeedProcessorBuilder method from the source container instance to create a builder.
var builder = sourceContainer.GetChangeFeedProcessorBuilder<Product>(
    processorName: "productItemProcessor",
    onChangesDelegate: changeHandlerDelegate // Delegate defined in previous slide
);

// Build the change feed processor with the defined builder and lease container
ChangeFeedProcessor processor = builder
    .WithInstanceName("desktopApplication")
    .WithLeaseContainer(leaseContainer)
    .Build();

// Run processor asynchronously.
await processor.StartAsync();

// Wait while processor handles items

// Terminate processor asynchronously.
await processor.StopAsync();
```

Implement the change feed estimator

Configured similarly to configuring the processor. Returns the number of unprocessed changes.

```
// Add a delegate using the ChangesEstimationHandler to poll how many changes have not yet been processed in the change feed.
ChangesEstimationHandler changeEstimationDelegate = async
    (long estimation, CancellationToken cancellationToken) =>
{     // Do something with the estimation   };
```

```
// Create an instance for both the source and lease container.
Container sourceContainer = client.GetContainer("cosmicworks", "products");
Container leaseContainer = client.GetContainer("cosmicworks", "productslease");

// Use the GetChangeFeedEstimatorBuilder method from the source container instance to create a builder.
var builder = sourceContainer.GetChangeFeedEstimatorBuilder(
    processorName: "productItemEstimator",
    estimationDelegate: changeEstimationDelegate) // Delegate defined in script above
);

// Build the change feed estimator with the defined builder and lease container
ChangeFeedProcessor estimator = builder
   .WithLeaseContainer(leaseContainer)
   .Build();

// Run processor asynchronously.
await processor.StartAsync();

// Wait while processor handles items

// Terminate processor asynchronously.
await processor.StopAsync();
```

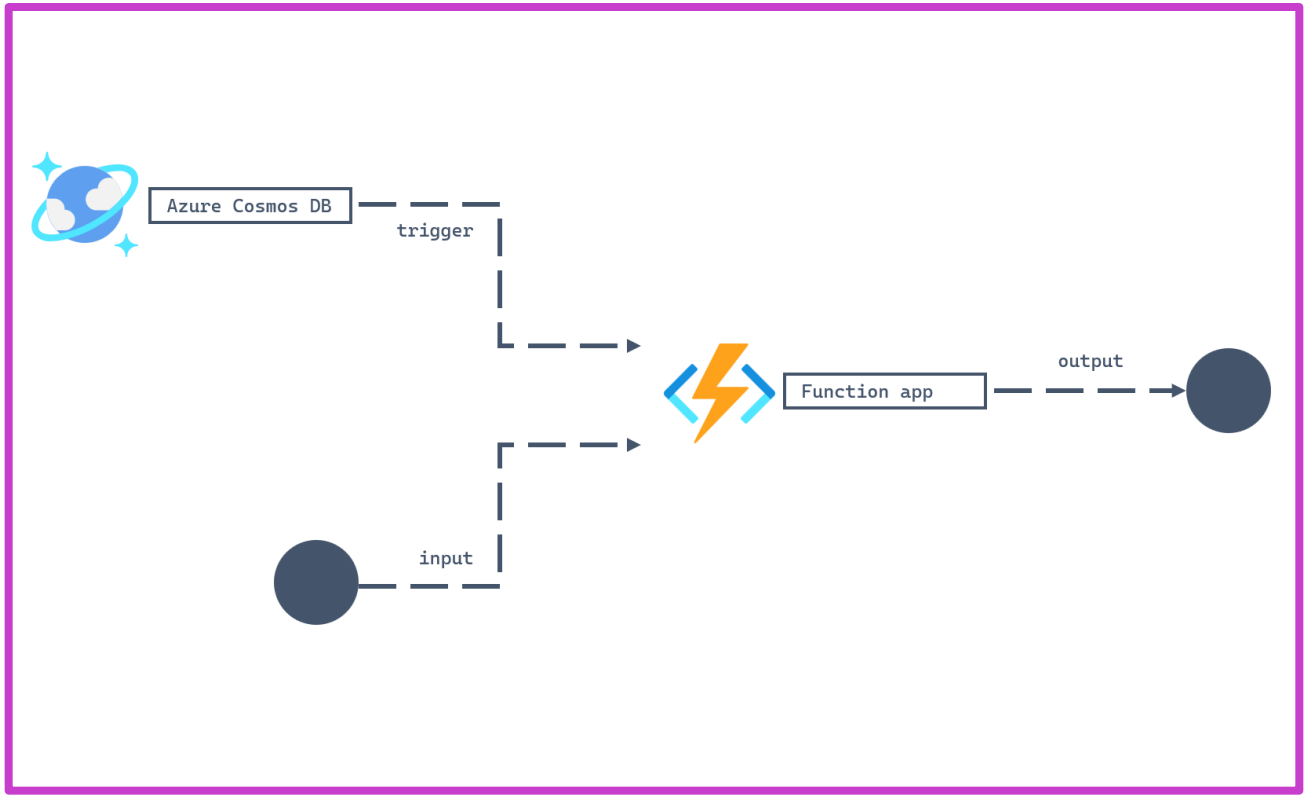
Lab – Process change feed events using the Azure Cosmos DB for NoSQL SDK

- Prepare your development environment
- Create an Azure Cosmos DB for NoSQL account
- Implement the change feed processor in the .NET SDK
- Seed your Azure Cosmos DB for NoSQL account with sample data

Handle events with Azure Functions and Azure Cosmos DB for NoSQL change feed

Understand Azure Function bindings for Azure Cosmos DB for NoSQL

Azure Functions is a service that offers serverless blocks of code that can run logic on-demand.



Azure Function bindings supported by Azure Cosmos DB

Trigger	Input	Output
Start a function's execution when there is a batch of documents ready from the change feed	Read one or more items using either a point read or a SQL query	Write one or more items to a container

Configure function bindings

The ***function.json*** file contains a JSON object with a property named bindings.
The bindings object is an array of trigger, input, and output bindings.

Trigger function on changes in the change feed

```
{ "bindings": [ {
    "type": "cosmosDBTrigger",
    "name": "changes",
    "direction": "in",
    "connectionStringSetting": "cosmosdbsqlconnstr",
    "databaseName": "cosmicworks",
    "collectionName": "products",
    "leaseCollectionName": "productslease",
    "createLeaseCollectionIfNotExists": false
} ] }
```

SQL query input binding

```
{ "bindings": [ {
    "type": "cosmosDB",
    "name": "items",
    "direction": "in",
    "connectionStringSetting": "cosmosdbsqlconnstr",
    "databaseName": "cosmicworks",
    "collectionName": "products",
    "sqlQuery": "SELECT p.id, p.name, p.categoryId FROM products
p WHERE p.price > 500"
} ] }
```

Point read input binding

```
{ "bindings": [ {
    "type": "cosmosDB",
    "name": "item",
    "direction": "in",
    "connectionStringSetting": "cosmosdbsqlconnstr",
    "databaseName": "cosmicworks",
    "collectionName": "products",
    "id": "91AA100C-D092-4190-92A7-7C02410F04EA",
    "partitionKey": "F3FBB167-11D8-41E4-84B4-5AAA92B1E737"
} ] }
```

Output items from the function

```
{ "bindings": [ {
    "type": "cosmosDB",
    "name": "output",
    "direction": "out",
    "connectionStringSetting": "cosmosdbsqlconnstr",
    "databaseName": "cosmicworks",
    "collectionName": "products"
} ] }
```

Develop function

The *name* of the method parameter must match the value provided in the binding configuration for the *name* property.

Product class definitions used in the following examples

```
public class Product
{
    public string id { get; set; }
    public string categoryId { get; set; }
    public string name { get; set; }
}
```

Trigger function on changes in the change feed

```
public static void Run(IReadOnlyList<Product> changes)
{
    foreach(Product doc in changes ?? Enumerable.Empty<Product>())
    { //Do something with each item }
}
```

Point read input binding

```
public static void Run(HttpRequest request, Product item)
{ //Do something with this item }
```

SQL query input binding

```
public static void Run(HttpRequest request, IEnumerable<Product> items)
{ //Do something with these items }
```

Output items from the function

```
public static async Task Run(HttpRequest request,
IAsyncCollector<Product> output)
{
    var firstProduct = new Product()
    { id = "7236DDB5-CFE0-4D3D-8FE5-799B398396B1",
      categoryId = "AE48F0AA-4F65-4734-A4CF-D48B8F82267F",
      name = "Road-650 Black, 48" };
    var secondProduct = new Product()
    { id = "878C50F0-7E29-4D0D-A52E-6D8B063673E3",
      categoryId = "AE48F0AA-4F65-4734-A4CF-D48B8F82267F",
      name = "Road-250 Red, 58" };

    await output.AddAsync(firstProduct);
    await output.AddAsync(secondProduct);
}
```

Lab – Archive Azure Cosmos DB for NoSQL data using Azure Functions

- Create an Azure Cosmos DB for NoSQL account
- Create an Azure Function app and Azure Cosmos DB-triggered function
- Implement function code in .NET
- Seed your Azure Cosmos DB for NoSQL account with sample data

dp420-14

Function App

Search

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Microsoft Defender for Cloud
- Events (preview)

Functions

- App keys
- App files
- Proxies

Deployment

- Deployment slots
- Deployment Center

Settings

- Configuration
- Authentication

Browse | Refresh | Stop | Restart | Swap | Get publish profile | Reset publish profile | Download app content

Essentials

Resource group (move)	: dp-420-lab14	URL	: https://dp420-14.azurewebsites.net
Status	: Running	Operating System	: Windows
Location (move)	: West US	App Service Plan	: ASP-dp420lab14-8ffa (Y1: 0)
Subscription (move)	: MSDN 플랫폼	Runtime version	: 4.28.4.21900
Subscription ID	: 5e47f1bf-145b-45f0-b656-210581ef806e		

Tags (edit) :

Functions | Metrics | Properties | Notifications (0)

Create functions in your preferred environment

Create in Azure portal

Best optimized for:

- Getting started without local setup
- Choose from our Function templates

Create function

VS Code Desktop

Best optimized for:

- Local development within VS Code
- Custom development tool requirements

Create with VS Code Desktop

Other editors or CLI

Best optimized for:

- Using preferred editor fo
- Visual Studio, IntelliJ, cor

Set up your editor

dp420-14 \ ItemsListener \ run.csx

```
#r "Microsoft.Azure.DocumentDB.Core"

using System;
using System.Collections.Generic;
using Microsoft.Azure.Documents;

public static void Run(IReadOnlyList<Document> input, ILogger log)
{
    log.LogInformation($"# Modified Items:\t{input?.Count ?? 0}");
    foreach(Document item in input)
    {
        log.LogInformation($"Detected Operation:\t{item.Id}");
    }
}
```

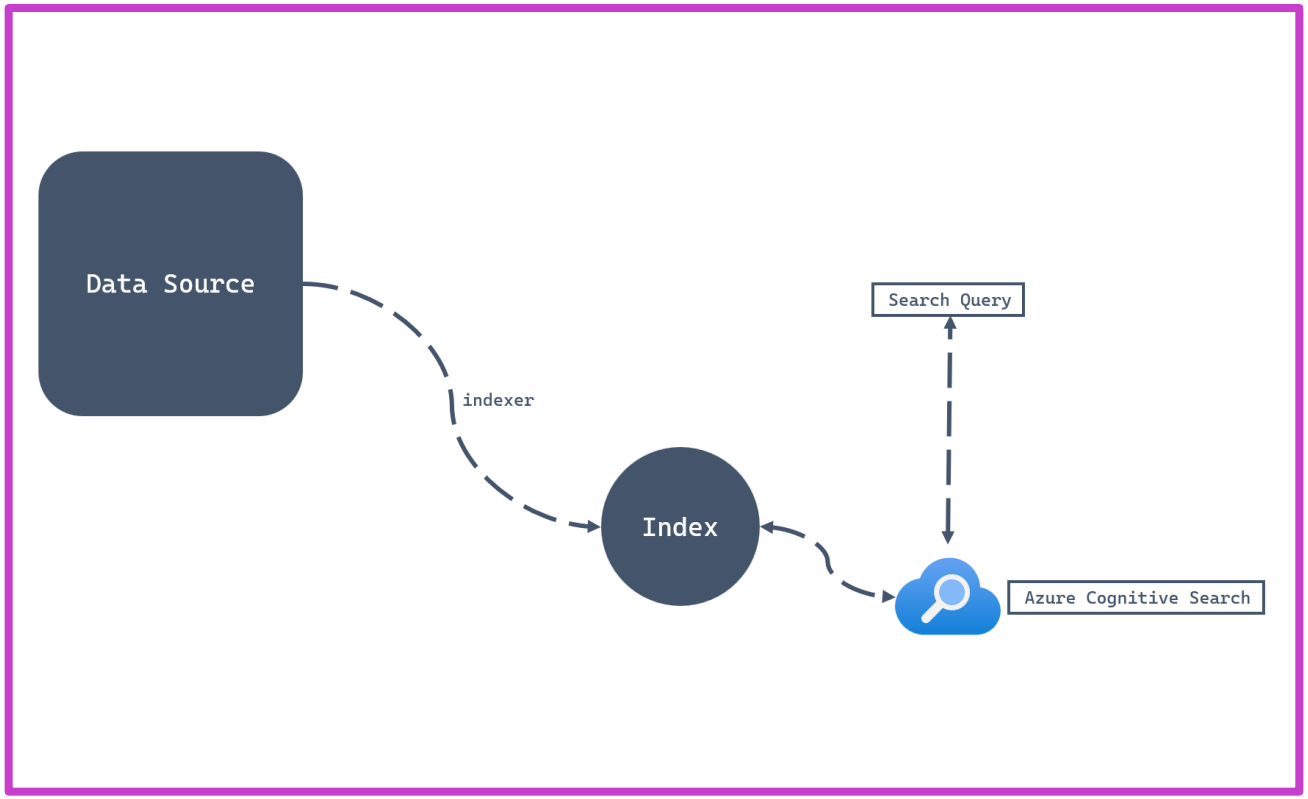
Logs | App Insights Logs | Log Level | Stop | Copy | Clear | Maximize | Lea

```
Connected! You are now viewing logs of Function runs in the current Code + Test panel. To see all the logs for this Function, please go to 'Monitor' from the Function menu.
```

Search Azure Cosmos DB for NoSQL data with Azure Cognitive Search

Create an indexer for data in Azure Cosmos DB for NoSQL

An Azure Cognitive Search instance is comprised of Data Sources, Indexers, and Indexes.



To configure Azure Cosmos DB for NoSQL for an Azure Cognitive Search instance you need to:

Connect a data source	Connection string Database Collection Query
Customize the index	Retrievable Filterable Sortable Facetable Searchable
Configure the indexer	Name Schedule

Implement a change detection policy

To accomplish change detection, the indexer will index all items returned by the query. *

Default Azure Cosmos DB for NoSQL data source query

```
SELECT *
FROM c
WHERE c._ts >= @HighWaterMark
ORDER BY c._ts
```

*If you write a custom query, you must sort using the *_ts* field to enable incremental progress when indexing.

Manage a data deletion detection policy

To enable tracking of deleted items, you must configure a policy to track when an item is deleted.

Set a value to the data source ***softDeleteColumnName***, *_isDeleted* in this example, and set ***softDeleteMarkerValue*** to true.

```
{
    "id": "E08E4507-9666-411B-AAC4-519C00596B0A",
    "categoryId": "86F3CBAB-97A7-4D01-BABB-ADEFFFAED6B4",
    "sku": "TI-R092",
    "name": "LL Road Tire",
    "_isDeleted": true
}
```

Lab – Search data using Azure Cognitive Search and Azure Cosmos DB for NoSQL

- Create an Azure Cosmos DB for NoSQL account
- Seed your Azure Cosmos DB for NoSQL account with sample data
- Create Azure Cognitive Search resource
- Build indexer and index for Azure Cosmos DB for NoSQL data
- Validate index with example search queries