Module 04: Access and manage data with the Azure Cosmos DB SQL API SDKs

Implement Azure Cosmos DB SQL API point operations

Understand point operations

The Microsoft.Azure.Cosmos library includes first-class support for generics in the C# language. At the most foundational level, you can create a C# class that represents an item in your container that, at a minimum, contains two members:

- a string property named id with a public getter and setter
- a string property with the same name as your partition key path with a public getter and setter

In [ ]:
```
// Container class at the most foundational level

public class item
{
    public string id { get; set; }
    public string partitionKey { get; set; }
}
```

In [ ]:
```
// Example of a Product class for a Product container

public class Product
{
    public string id { get; set; }
    public string name { get; set; }
    public string categoryId { get; set; } // Assume categoryId is the partition key
    public double price { get; set; }
    public string[] tags { get; set; }
}
```

In [ ]:
```
// You can disassociate the id property if needed

using Newtonsoft.Json;

public class item
{
    [JsonProperty(PropertyName = "id")]
    public string InternalId { get; set; }

    public string partitionKey { get; set; }
}
```

Create documents

Let's add a new item of the previously defined Product class type.

In [ ]:
```
Product saddle = new()
{
    id = "027D0B9A-F9D9-4C96-8213-C8546C4AAE71",
    categoryId = "26C74104-40BC-4541-8EF5-9892F7F03D72",
    name = "LL Road Seat/Saddle",
    price = 27.12d,
    tags = new string[] { "brown", "weathered" }
};
```

In [ ]:

```
using Microsoft.Azure.Cosmos;
using System.Net;

CosmosClient client = new (connectionString);
Database database = await client.CreateDatabaseIfNotExistsAsync("cosmicworks");
ContainerProperties properties = new ContainerProperties("cosmicworks", "/categoryId");
properties.DefaultTimeToLive = 1000;  // we will configure the default time to live for the
Container container = await database.CreateContainerIfNotExistsAsync(properties,ThroughputP

try
{
    await container.CreateItemAsync<Product>(saddle);
}
catch(CosmosException ex) when (ex.StatusCode == HttpStatusCode.Conflict)
{
    // Add logic to handle conflicting ids
    // Console.WriteLine(ex.ToString());
}
catch(CosmosException ex)
{
    // Add general exception handling logic
}
```

Code	Title	Reason
400	Bad Request	Something was wrong with the item in the body of the request
403	Forbidden	Container was likely full
409	Conflict	Item in container likely already had a matching id
413	RequestEntityTooLarge	Item exceeds max entity size
429	TooManyRequests	Current request exceeds the maximum RU/s provisioned for the container

Read an existing document

In [ ]:

```
// We first need the Unique Id of the document we are searching for.
string id = "027D0B9A-F9D9-4C96-8213-C8546C4AAE71";

// We then need the partition key of the document we are searching for.
string categoryId = "26C74104-40BC-4541-8EF5-9892F7F03D72";
PartitionKey partitionKey = new (categoryId);

// With both the id and the partition key we can now search for the document.
Product saddle = await container.ReadItemAsync<Product>(id, partitionKey);

// If we find the document, we can now do something with its data like display it to the co
string formattedName = $"New Product [{saddle.name}]";
Console.WriteLine(formattedName);

saddle
```

Update a document

In [ ]:

```
// You can modify the saddle variable we defined earlier.
saddle.price = 35.00d;

// We can persist the change invoking the asynchronous UpsertItemAsync<> method passing in
await container.UpsertItemAsync<Product>(saddle);

// We can modify other properties of the saddle variable.
saddle.tags = new string[] { "brown", "new", "crisp" };
```

```
// Even though we upserted the document already, we don't have to read a new item before up
await container.UpsertItemAsync<Product>(saddle);

saddle = await container.ReadItemAsync<Product>(id, partitionKey);

saddle
```

Delete documents

In [ ]:

```
// We first need the Unique Id of the document we want to delete.
string id = "027D0B9A-F9D9-4C96-8213-C8546C4AAE71";

// We then need the partition key of the document we want to delete.
string categoryId = "26C74104-40BC-4541-8EF5-9892F7F03D72";
PartitionKey partitionKey = new (categoryId);

// With the id and partition key, you invoke the asynchronous DeleteItemAsync<> method in a
// the ReadItemAsync<> method.
await container.DeleteItemAsync<Product>(id, partitionKey);
```

Perform cross-document transactional operations with the Azure Cosmos DB SQL API

Create a transactional batch with the SDK

In [ ]:

```
using System.Collections.Generic;

// The transactional batch supports operations with the same logical partition key.
public record Product(string id, string name, string categoryId);

Product saddle = new("0120", "Worn Saddle", "accessories-used");
Product handlebar = new("012A", "Rusty Handlebar", "accessories-used");

PartitionKey partitionKey = new ("accessories-used");

TransactionalBatch batch = container.CreateTransactionalBatch(partitionKey)
    .CreateItem<Product>(saddle)
    .CreateItem<Product>(handlebar);

TransactionalBatchResponse response = await batch.ExecuteAsync();
for(int i=0; i<response.Count; i++) { Console.WriteLine($"{i}, {response[i].StatusCode}");
```

In [ ]:

```
// Operations with different logical partition keys will FAIL batch operation.
public record Product(string id, string name, string categoryId);

Product saddle = new("0240", "Worn Saddle", "accessories-used");
Product handlebar = new("024A", "Rusty Handlebar", "accessories-new");  // new logical part

PartitionKey partitionKey = new ("accessories-used");

TransactionalBatch batch = container.CreateTransactionalBatch(partitionKey)
    .CreateItem<Product>(saddle)
    .CreateItem<Product>(handlebar);

TransactionalBatchResponse response = await batch.ExecuteAsync();
for(int i=0; i<response.Count; i++) { Console.WriteLine($"{i}, {response[i].StatusCode}");
```

Review batch operation results with the SDK

In [ ]:

```
if (response.IsSuccessStatusCode)
{
```

```
    Console.WriteLine("That was a success!");
}

TransactionalBatchOperationResult<Product> result0 = response.GetOperationResultAtIndex<Pro
Product firstProductResult = result0.Resource;

TransactionalBatchOperationResult<Product> result1 = response.GetOperationResultAtIndex<Pro
Product secondProductResult = result1.Resource;

Console.WriteLine(firstProductResult);
Console.WriteLine(secondProductResult);
```

Implement optimistic concurrency control

Using the SDK to read an item and then update the same item in a subsequent operation carries some inherent risk.

Another operation could potentially come in from a separate client and change the underlying document before the first client's update operation is finalized. This conflict could create a "lost update" situation.

In [ ]:

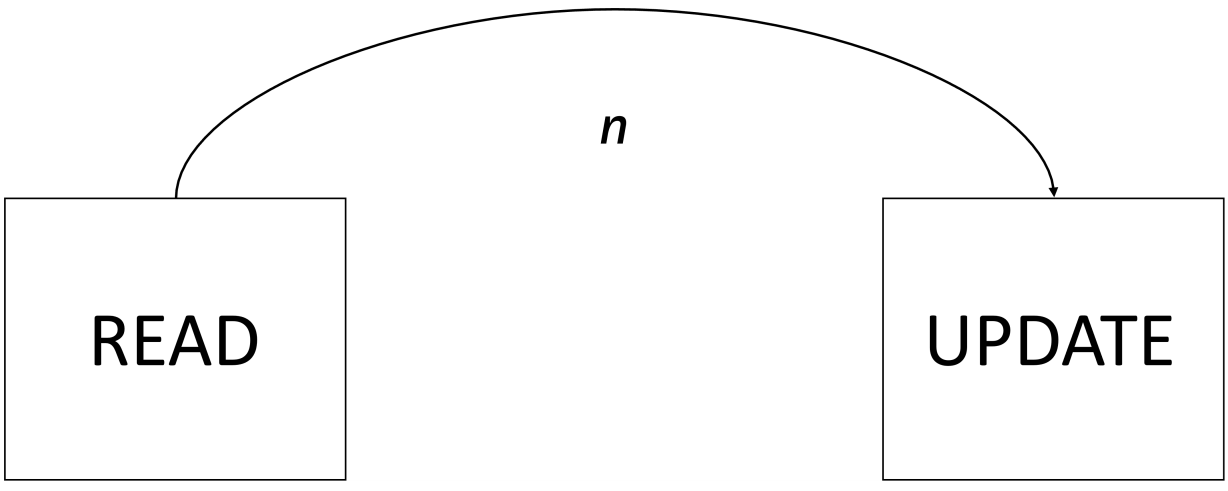
```
public class Product
{
    public string id { get; set; }
    public string name { get; set; }
    public string categoryId { get; set; } // Assume categoryId is the partition key
    public double price { get; set; }
}

string categoryId = "accessories-used";
PartitionKey partitionKey = new (categoryId);

Product product = await container.ReadItemAsync<Product>("0120", partitionKey);

product.price = 50d;

await container.UpsertItemAsync<Product>(product, partitionKey);
```



In [ ]:

```
// The C# code only required minor changes to implement optimistic concurrency control.
string categoryId = "accessories-used";

PartitionKey partitionKey = new (categoryId);
ItemResponse<Product> response = await container.ReadItemAsync<Product>("0120", partitionKe

Product product = response.Resource;
string eTag = response.ETag;
product.price = 50d;

// To prevent lost updates, you can use the if-match rule to see if the ETag still matches
ItemRequestOptions options = new ItemRequestOptions { IfMatchEtag = eTag };
```

```
await container.UpsertItemAsync<Product>(product, partitionKey, requestOptions: options);
```

In [ ]:

```
// Let's see if we can break the ETag
ItemResponse<Product> response1 = await container.ReadItemAsync<Product>("0120", partitionK
ItemResponse<Product> response2 = await container.ReadItemAsync<Product>("0120", partitionK

Product product1 = response1.Resource;
Product product2 = response2.Resource;

string eTag1 = response1.ETag;
string eTag2 = response2.ETag;
product1.price = 20d;
product2.price = 40d;

ItemRequestOptions options1 = new ItemRequestOptions { IfMatchEtag = eTag1 };
ItemRequestOptions options2 = new ItemRequestOptions { IfMatchEtag = eTag2 };

await container.UpsertItemAsync<Product>(product1, partitionKey, requestOptions: options1);
```

In [ ]:

```
// Will FAIL (because the eTag is still the old value)
await container.UpsertItemAsync<Product>(product2, partitionKey, requestOptions: options2);
```

Process bulk data in Azure Cosmos DB SQL API

In [ ]:

```
CosmosClientOptions options = new ()
{
    AllowBulkExecution = true
};

CosmosClient client = new (connectionString, options);
```

In [ ]:

```
using Bogus;
public record Product(string id, string name, string categoryId);
var productsToInsert = new List<Product>();
var faker = new Faker("en");

for (int i=0; i<2500; i++)
{
    productsToInsert.Add(new Product(Guid.NewGuid().ToString(), faker.Name.FullName(), fake
}

// there are about 20 categories/departments...

productsToInsert
```

In [ ]:

```
await container.ReplaceThroughputAsync(4000); // Response status code does not indicate suc

List<Task> concurrentTasks = new List<Task>();

foreach(Product product in productsToInsert)
{
    concurrentTasks.Add(
        container.CreateItemAsync<Product>(
            product,
            new PartitionKey(product.categoryId)
        )
    );
}

// The Task.WhenAll will create batches to group our operations by physical partition,
// then distribute the requests to run concurrently.
await Task.WhenAll(concurrentTasks);
```