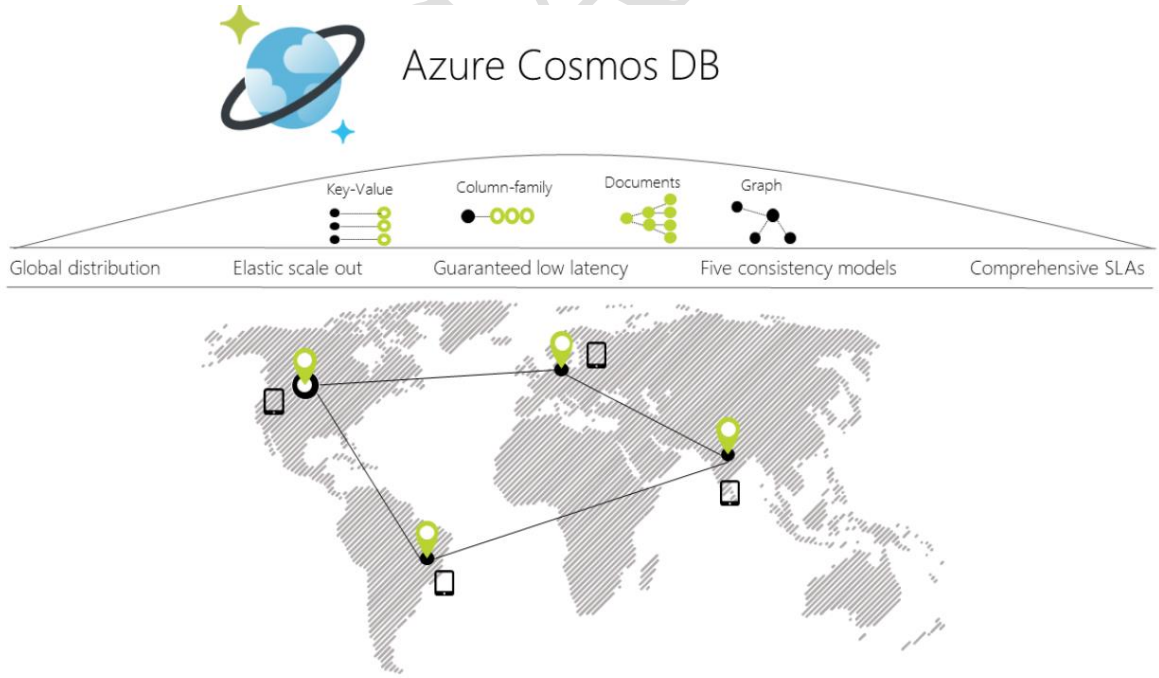
Agenda: Azure CosmosDB Service

- What is CosmosDB
- Understanding DocumentDB database Schema
- Creating and Saving Documents using Portal
- Programming Schema
- Adding / Editing / Deleting and Reading Documents

What is Cosmos DB?

- Azure Cosmos DB is **Microsoft's globally distributed, multi-model database**.
- Azure Cosmos DB enables you to elastically and independently scale throughput and storage across any number of Azure's geographic regions.
- **It offers throughput, latency, availability, and consistency guarantees** with comprehensive service level agreements (SLAs), something no other database service can offer. 99.99% availability SLA for all single region database accounts, and all 99.999% read availability on all multi-region database accounts.
- For a typical 1KB item, Cosmos DB guarantees end-to-end latency of reads under 10 ms and indexed writes under 15 ms at the 99th percentile, within the same Azure region. The median latencies are significantly lower (under 5 ms).
- Five to ten times more cost effective than a non-managed solution or an on-prem NoSQL solution.Three times cheaper than AWS DynamoDB or Google Spanner.



Multiple data models and popular APIs for accessing and querying data

- The **atom-record-sequence** (ARS) based data model that Azure Cosmos DB is built on natively supports multiple data models, including but not limited to document, graph, key-value, table, and column-family data models.
- APIs for the following data models are supported with SDKs available in multiple languages:
    1. **SQL API**: A schema-less JSON database engine with rich SQL querying capabilities.
    2. **MongoDB API**: A massively scalable *MongoDB-as-a-Service* powered by Azure Cosmos DB platform. Compatible with existing MongoDB libraries, drivers, tools, and applications.
    3. **Cassandra API**: A globally distributed Cassandra-as-a-Service powered by Azure Cosmos DB platform. Compatible with existing Apache Cassandra libraries, drivers, tools, and applications.
    4. **Gremlin API**: A fully managed, horizontally scalable graph database service that makes it easy to build and run applications that work with highly connected datasets supporting Open Gremlin APIs
    5. **Table API**: A key-value database service built to provide premium capabilities (for example, automatic indexing, guaranteed low latency, global distribution) to existing Azure Table storage applications without making any app changes.

Capability Comparison:

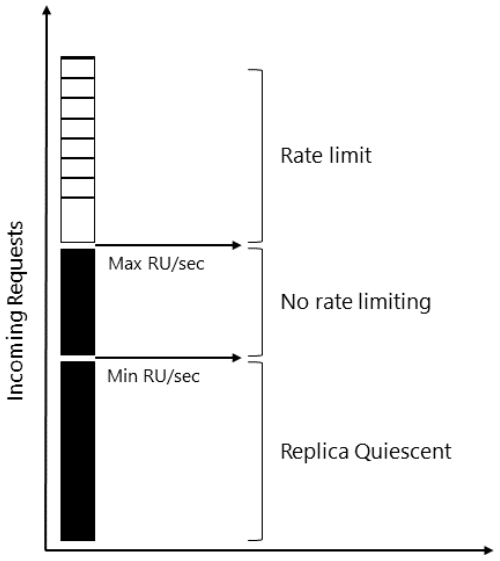
Azure Cosmos DB provides the best capabilities of relational and non-relational databases.

Capabilities	Relational databases	Non-relational (NoSQL) databases	Azure Cosmos DB
Global distribution	No	No	Yes, turnkey distribution in 30+ regions, with multi-homing APIs
Horizontal scale	No	Yes	Yes, you can independently scale storage and throughput
Latency guarantees	No	Yes	Yes, 99% of reads in <10 ms and writes in <15 ms
High availability	No	Yes	Yes, Azure Cosmos DB is always on, has well-defined PACELC tradeoffs, and offers automatic and manual failover options
Data model + API	Relational + SQL	Multi-model + OSS API	Multi-model + SQL + OSS API (more coming soon)
SLAs	Yes	No	Yes, comprehensive SLAs for latency, throughput, consistency, availability

RUs – Azure Cosmos DB

Azure Cosmos DB reserves resources to manage the throughput of an application. Because, application load and access patterns change over time, Azure Cosmos DB has support built-in to increase or decrease the amount of reserved throughput available at any time.

With Azure Cosmos DB, reserved throughput is specified in terms of **request unit processing per second (RU/s)**. You reserve several guaranteed request units to be available to your application on a per-second basis. Each operation in Azure Cosmos DB, including writing a document, performing a query, and updating a document, consumes CPU, memory, and Input/output operations per second (IOPS). That is, each operation incurs a request charge, which is expressed in request units.



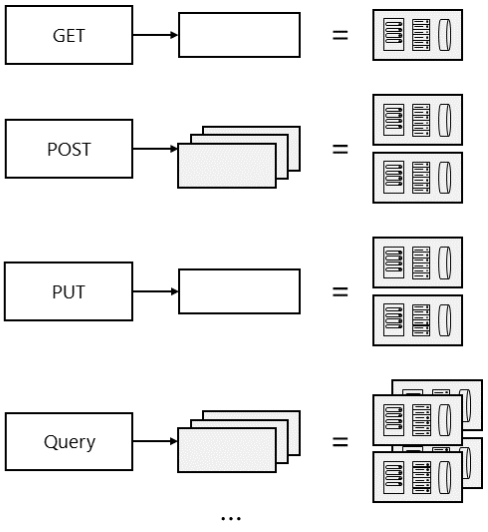
A request unit is a normalized measure of request processing cost. A single request unit represents the processing capacity that's required to read, via self-link or ID, a single item that is 1 kilobyte (KB) and that consists of 10 unique property values (excluding system properties). A request to create (insert), replace, or delete the same item consumes more processing from the service and thereby requires more request units.

Normalized across various access methods

1 RU = 1 read of 1 KB document

Each request consumes fixed RUs

Applies to reads, writes, queries, and stored procedure execution



Document Database

A document database is conceptually similar to a key/value store, except that it stores a collection of named fields and data (known as documents), each of which could be simple scalar items or compound elements such as lists and child collections. There are several ways in which you can encode the data in a document's fields, including

using Extensible Markup Language (XML), YAML, JavaScript Object Notation (JSON), Binary JSON (BSON), or even storing it as plain text. Unlike key/value stores, the fields in documents are exposed to the storage management system, enabling an application to query and filter data by using the values in these fields.

Typically, a document contains the entire data for an entity. What items constitute an entity are application specific. For example, an entity could contain the details of a customer, an order, or a combination of both. A single document may contain information that would be spread across several relational tables in an RDBMS.

Key	Document

1001	<pre>[
	  {
	  "customerId": 344,
	  "orderItems": [
	  {
	  "productId": 4524,
	  "quantity": 1,
	  "price": 125.67
	  },
	  {
	  "productId": 3311,
	  "quantity": 4,
	  "price": 73.06
	  }
	  ],
	  "orderDate": "2017-10-18T12:27:30 +04:00"
	  }
	]</pre>

1002	<pre>[
	  {
	  "customerId": 263,
	  "orderItems": [
	  {
	  "productId": 4076,
	  "quantity": 3,
	  "price": 257.64
	  }
	  ],
	  "orderDate": "2014-01-31T02:09:02 +05:00"
	  }
	]</pre>

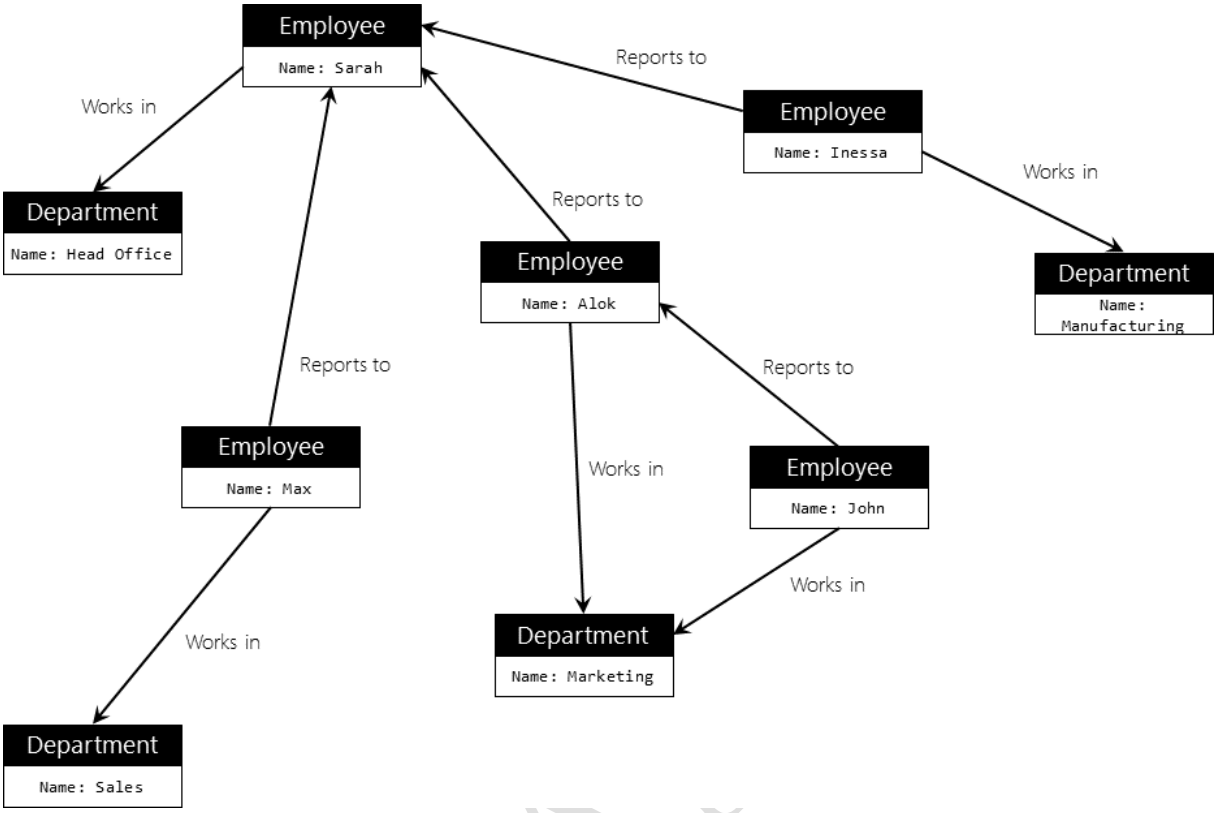
1003	<pre>[
	  {
	  "customerId": 308,
	  "orderItems": [
	  {
	  "productId": 1957,
	  "quantity": 1,
	  "price": 279.63
	  }
	  ],
	  "orderDate": "2016-09-18T01:33:53 +04:00"
	  }
	]</pre>

Graph databases

A graph database stores two types of information, nodes and edges. You can think of nodes as entities. Edges which specify the relationships between nodes. Both nodes and edges can have properties that provide information about that node or edge, similar to columns in a table. Edges can also have a direction indicating the nature of the relationship.

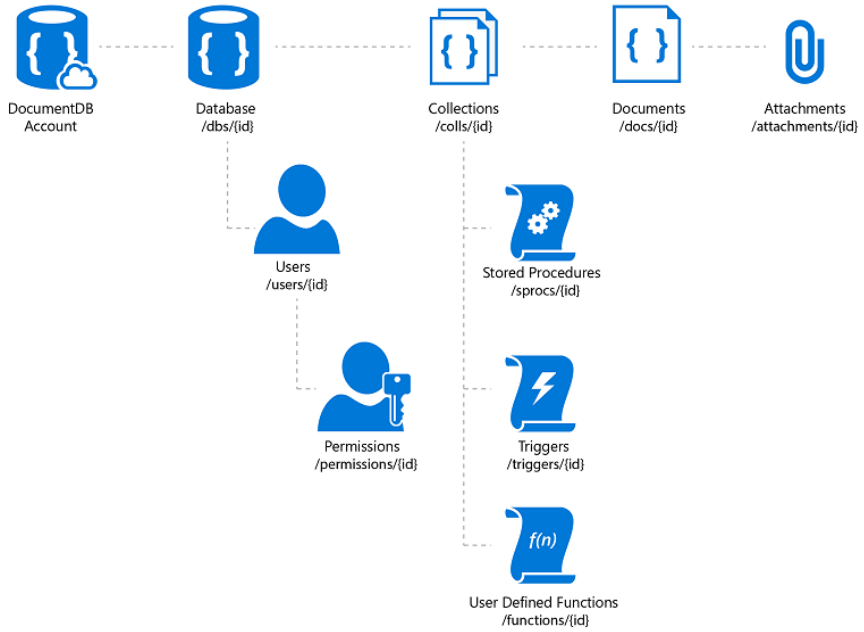
The purpose of a graph database is to allow an application to efficiently perform queries that traverse the network of nodes and edges, and to analyze the relationships between entities. The following diagram shows an organization's personnel database structured as a graph. The entities are employees and departments, and the

edges indicate reporting relationships and the department in which employees work. In this graph, the arrows on the edges show the direction of the relationships.



Core (SQL) OR DocumentDB

The image below shows the relationships between the resources:



https://dssdemodb.documents.azure.com:443/dbs/<database-id>/colls/<col-id>/docs/<doc-id>

Addressing a resource

All resources are URI addressable. The value of the **_self** property of a resource represents the relative URI of the resource. The format of the URI consists of the /<feed>/{_rid} path segments:

Value of the _self	Description
/dbs	Feed of databases under a database account
/dbs/{dbName} UriFactory.CreateDatabaseUri(DatabaseId)	Database with an id matching the value {dbName}
/dbs/{dbName}/colls/	Feed of collections under a database
/dbs/{dbName}/colls/{collName}	Collection with an id matching the value {collName}
/**dbs**/{dbName}/**colls**/{collName}/**docs**	Feed of documents under a collection
/dbs/{dbName}/colls/{collName}/docs/{docId}	Document with an id matching the value {doc}
/dbs/{dbName}/users/	Feed of users under a database
/dbs/{dbName}/users/{userId}	User with an id matching the value {user}
/dbs/{dbName}/users/{userId}/permissions	Feed of permissions under a user
/dbs/{dbName}/users/{userId}/permissions/{permissionId}	Permission with an id matching the value {permission}

Each resource has a unique user defined name exposed via the id property. The id is a user defined string, of up to 256 characters that is unique within the context of a specific parent resource.

Note: for documents, if the user does not specify an id, our supported SDKs will automatically generate a unique id for the document.

Managing CRUD Operations using Portal

1. Azure Portal → +Create a resource → Azure Cosmos DB
2. ID=dsscosmosdbacccount, API=**SQL**. . . → Create
3. **Add Collection**: Data Explorer → New Collection → Database Id=Organization, Collection Id=Employees, → OK
4. Add Sample data: Click on New Document

```
{
   "id": "1",
   "name": "E1",
   "department": "Development",
   "isPermanent": false
}
```

5. We can as well update and delete documents from the same interface

Managing CRUD Operations through Code

1. Create a new ASP.NET MVC Web Application
2. Add reference to NuGet package **Microsoft.Azure.DocumentDB**.
3. Add the following to <appSettings> in web.config

```
<add key="endpoint" value="https://dssdocdbacct.documents.azure.com:443/" />
<add key="authKey" value="ZlrzNoS5BWCXREOLGR8dHRHFTXWxGl4lIcv0KDuHVUA==" />
<add key="database" value="Organization" />
<add key="collection" value="Employees" />
```

4. Add new class DocumentDBRepository

```
public static class DocumentDBRepository<T> where T : class
{
   private static readonly string DatabaseId = ConfigurationManager.AppSettings["database"];
   private static readonly string CollectionId = ConfigurationManager.AppSettings["collection"];
   private static DocumentClient client;
   public static async Task<T> GetItemAsync(string id)
   {
        try
        {
           Document document = await client.ReadDocumentAsync(UriFactory.CreateDocumentUri(DatabaseId,
CollectionId, id));
           return (T)(dynamic)document;
        }
        catch (DocumentClientException e)
        {
           if (e.StatusCode == System.Net.HttpStatusCode.NotFound)
              return null;
           else
              throw;
        }
   }
   public static async Task<IEnumerable<T>> GetItemsAsync(Expression<Func<T, bool>> predicate)
   {
      IDocumentQuery<T> query = client.CreateDocumentQuery<T>(
        UriFactory.CreateDocumentCollectionUri(DatabaseId, CollectionId),
        new FeedOptions { MaxItemCount = -1 })
        .Where(predicate)
        .AsDocumentQuery();
```

```
        List<T> results = new List<T>();
        while (query.HasMoreResults)
        {
            results.AddRange(await query.ExecuteNextAsync<T>());
        }
        return results;
    }
    public static async Task<Document> CreateItemAsync(T item)
    {
        return await client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri(DatabaseId,
CollectionId), item);
    }
    public static async Task<Document> UpdateItemAsync(string id, T item)
    {
        return await client.ReplaceDocumentAsync(UriFactory.CreateDocumentUri(DatabaseId, CollectionId, id),
item);
    }
    public static async Task DeleteItemAsync(string id)
    {
        await client.DeleteDocumentAsync(UriFactory.CreateDocumentUri(DatabaseId, CollectionId, id));
    }
    public static void Initialize()
    {
        client = new DocumentClient(new Uri(ConfigurationManager.AppSettings["endpoint"]),
ConfigurationManager.AppSettings["authKey"]);
        CreateDatabaseIfNotExistsAsync().Wait();
        CreateCollectionIfNotExistsAsync().Wait();
    }
    private static async Task CreateDatabaseIfNotExistsAsync()
    {
        try
        {
            await client.ReadDatabaseAsync(UriFactory.CreateDatabaseUri(DatabaseId));
        }
        catch (DocumentClientException e)
        {
```

```
        if (e.StatusCode == System.Net.HttpStatusCode.NotFound)
        {
            await client.CreateDatabaseAsync(new Database { Id = DatabaseId });
        }
        else
        {
            throw;
        }
    }
  }
  private static async Task CreateCollectionIfNotExistsAsync()
  {
    try
    {
      await client.ReadDocumentCollectionAsync(UriFactory.CreateDocumentCollectionUri(DatabaseId,
CollectionId));
    }
    catch (DocumentClientException e)
    {
      if (e.StatusCode == System.Net.HttpStatusCode.NotFound)
      {
        await client.CreateDocumentCollectionAsync(
          UriFactory.CreateDatabaseUri(DatabaseId),
          new DocumentCollection { Id = CollectionId },
          new RequestOptions { OfferThroughput = 1000 });
      }
      else
      {
        throw;
      }
    }
  }
}
```

5. Add the following to Application_Start in Global.asax

```
DocumentDBRepository<Models.Employee>.Initialize();
```

6. Add the new class to Modals folder

```
public class Employee
{
   [JsonProperty(PropertyName = "id")]
   public string Id { get; set; }
   [JsonProperty(PropertyName = "name")]
   public string Name { get; set; }
   [JsonProperty(PropertyName = "department")]
   public string Department { get; set; }
   [JsonProperty(PropertyName = "isPermanent")]
   public bool IsPermanent { get; set; }
}
```

7. Add following views under Views\Employees → Right Click → Add → View → Select Template → Add
    a. Index
    b. Details
    c. Create
    d. Edit
    e. Delete
8. Add EmployeesController as below

```
public class EmployeesController : Controller
{
   // GET: Employees
   public async Task<ActionResult> Index()
   {
      return View(await DocumentDBRepository<Employee>.GetItemsAsync(emp => true));
   }
   // GET: Employees/Details/5
   public async Task<ActionResult> Details(string id)
   {
      if (id == null)
      {
         return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
      }
      Employee employee = await DocumentDBRepository<Employee>.GetItemAsync(id);
      if (employee == null)
      {
         return HttpNotFound();
```

```
        }
        return View(employee);
    }
    // GET: Employees/Create
    public ActionResult Create()
    {
        return View();
    }
    // POST: Employees/Create
    // To protect from overposting attacks, please enable the specific properties you want to bind to, for
    // more details see http://go.microsoft.com/fwlink/?LinkId=317598.
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> Create([Bind(Include = "Id,Name,Department,IsPermanent")] Employee
employee)
    {
        if (ModelState.IsValid)
        {
            await DocumentDBRepository<Employee>.CreateItemAsync(employee);
            return RedirectToAction("Index");
        }
        return View(employee);
    }

    // GET: Employees/Edit/5
    public async Task<ActionResult> Edit(string id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        Employee employee = await DocumentDBRepository<Employee>.GetItemAsync(id);
        if (employee == null)
        {
            return HttpNotFound();
        }
```

```
    return View(employee);
}
// POST: Employees/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Edit([Bind(Include = "Id,Name,Department,IsPermanent")] Employee employee)
{
    if (ModelState.IsValid)
    {
        await DocumentDBRepository<Employee>.UpdateItemAsync(employee.Id, employee);
        return RedirectToAction("Index");
    }
    return View(employee);
}

// GET: Employees/Delete/5
public async Task<ActionResult> Delete(string id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Employee employee = await DocumentDBRepository<Employee>.GetItemAsync(id);
    if (employee == null)
    {
        return HttpNotFound();
    }
    return View(employee);
}

// POST: Employees/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<ActionResult> DeleteConfirmed(string id)
```

```
  {
    await DocumentDBRepository<Employee>.DeleteItemAsync(id);
    return RedirectToAction("Index");
  }
  protected override void Dispose(bool disposing)
  {
    if (disposing)
    {
    }
    base.Dispose(disposing);
  }
}
```

Example - 2

Following items will be demonstrated.

- Creating and connecting to an Azure Cosmos DB account
- Configuring your Visual Studio Solution
- Creating an online database
- Creating a collection
- Creating JSON documents
- Querying the collection
- Replacing a document
- Deleting a document

Format of JSON Object we want to deal with.

Steps:

1. Create an Azure Cosmos DB account.
2. More Services → Azure Cosmos DB → +Add
3. ID=dssdocdbacct →API=SQL (DocumentDB). . . → Create
4. Create a **Console Based Application** in Visual Studio
5. Right Click on Project → Manage NuGet Packages → Browse Tab → Search **Azure DocumentDB** → Install
6. Edit the Program.cs file as below.

```
public class Family
{
```

```
    [JsonProperty(PropertyName = "id")]
    public string Id { get; set; }
    public string LastName { get; set; }
    public Parent[] Parents { get; set; }
    public Child[] Children { get; set; }
    public Address Address { get; set; }
    public bool IsRegistered { get; set; }
    public override string ToString()
    {
        return JsonConvert.SerializeObject(this);
    }
}
public class Parent
{
    public string FamilyName { get; set; }
    public string FirstName { get; set; }
}
public class Child
{
    public string FamilyName { get; set; }
    public string FirstName { get; set; }
    public string Gender { get; set; }
    public int Grade { get; set; }
    public Pet[] Pets { get; set; }
}
public class Pet
{
    public string GivenName { get; set; }
}
public class Address
{
    public string State { get; set; }
    public string County { get; set; }
    public string City { get; set; }
}
```

```
class Program
{
   private const string EndpointUrl = "https://dssdocdb.documents.azure.com:443/";
   private const string PrimaryKey =
"jeFGquQnr2nWmCEsOXXdahL7B4EXSt68V8TC5NR8s8Un4AyBRV8WVaenROoVZR2NeYv8jTdy8ElDVfy7cf575A==";
   private DocumentClient client;
   static void Main(string[] args)
   {
         Program p = new Program();
         p.GetStartedDemo().Wait();
         Console.WriteLine("End of demo, press any key to exit.");
         Console.ReadKey();
      }
   }
   private async Task GetStartedDemo()
   {
      this.client = new DocumentClient(new Uri(EndpointUrl), PrimaryKey);

      //CREATE DATABASE
      Database db = new Database() { Id = "FamilyDB" };
      await this.client.CreateDatabaseIfNotExistsAsync(db);

      //CREATE DOCUMENT COLLECTION
      DocumentCollection col = new DocumentCollection();
      col.Id = "FamilyCollection";
      Uri dbUri = UriFactory.CreateDatabaseUri("FamilyDB");
      await this.client.CreateDocumentCollectionIfNotExistsAsync(dbUri, col);

      //CREATE A FAMILY OBJECT (Document)
      Family andersenFamily = new Family
      {
         Id = "Andersen.1",
         LastName = "Andersen",
         Parents = new Parent[]
         {
```

```
            new Parent { FirstName = "Thomas" },
            new Parent { FirstName = "Mary Kay" }
        },
        Children = new Child[]
        {
            new Child
            {
                FirstName = "Henry",
                Gender = "female",
                Grade = 5,
                Pets = new Pet[]
                {
                    new Pet { GivenName = "Fluffy" }
                }
            }
        },
        Address = new Address {State = "WA", County = "King", City = "Seattle"},
        IsRegistered = true
    };

    //CREATE JSON DOCUMENT
    await this.CreateFamilyDocumentIfNotExists("FamilyDB", "FamilyCollection", andersenFamily);

    //CREATE FAMILY OBJECT (another document)
    Family wakefieldFamily = new Family
    {
        Id = "Wakefield.7",
        LastName = "Wakefield",
        Parents = new Parent[]
        {
            new Parent { FamilyName = "Wakefield", FirstName = "Robin" },
            new Parent { FamilyName = "Miller", FirstName = "Ben" }
        },
        Children = new Child[]
        {
            new Child
```

```
            {
                FamilyName = "Merriam",
                FirstName = "Jesse",
                Gender = "female",
                Grade = 8,
                Pets = new Pet[]
                {
                    new Pet { GivenName = "Goofy" },
                    new Pet { GivenName = "Shadow" }
                }
            },
            new Child
            {
                FamilyName = "Miller",
                FirstName = "Lisa",
                Gender = "female",
                Grade = 1
            }
        },
        Address = new Address { State = "NY", County = "Manhattan", City = "NY" },
        IsRegistered = false
    };

    //CREATE JSON DOCUMENT
    await this.CreateFamilyDocumentIfNotExists("FamilyDB", "FamilyCollection", wakefieldFamily);

    //QUERY COSMOS DB RESOURCE
    this.ExecuteSimpleQuery("FamilyDB", "FamilyCollection");

    //REPLACE JSON OBJECT
    andersenFamily.Children[0].Grade = 6;
    await this.ReplaceFamilyDocument("FamilyDB", "FamilyCollection", "Andersen.1", andersenFamily);
    this.ExecuteSimpleQuery("FamilyDB", "FamilyCollection");

    //DELETE JSON DOCUMENT
    await this.DeleteFamilyDocument("FamilyDB", "FamilyCollection", "Andersen.1");
```

```
    //DELETE THE DATABASE
    await this.client.DeleteDatabaseAsync(UriFactory.CreateDatabaseUri("FamilyDB"));
  }

  //CREATE JSON DOCUMENT
  private async Task CreateFamilyDocumentIfNotExists(string databaseName, string collectionName, Family
family)
  {
    try
    {
      await this.client.ReadDocumentAsync(UriFactory.CreateDocumentUri(databaseName, collectionName,
family.Id));
      this.WriteToConsoleAndPromptToContinue("Found {0}", family.Id);
    }
    catch (DocumentClientException de)
    {
      if (de.StatusCode == HttpStatusCode.NotFound)
      {
        await this.client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri(databaseName,
collectionName), family);
        this.WriteToConsoleAndPromptToContinue("Created Family {0}", family.Id);
      }
      else
      {
        throw;
      }
    }
  }

  //QUERY COSMOS DB RESOURCES
  private void ExecuteSimpleQuery(string databaseName, string collectionName)
  {
    // Set some common query options
    FeedOptions queryOptions = new FeedOptions { MaxItemCount = -1 };
```

```
        // Here we find the Andersen family via its LastName
        IQueryable<Family> familyQuery = this.client.CreateDocumentQuery<Family>(
            UriFactory.CreateDocumentCollectionUri(databaseName, collectionName), queryOptions)
            .Where(f => f.LastName == "Andersen");

        // The query is executed synchronously here, but can also be executed asynchronously via the
IDocumentQuery<T> interface
        Console.WriteLine("Running LINQ query...");
        foreach (Family family in familyQuery)
        {
            Console.WriteLine("\tRead {0}", family);
        }

        // Now execute the same query via direct SQL
        IQueryable<Family> familyQueryInSql = this.client.CreateDocumentQuery<Family>(
            UriFactory.CreateDocumentCollectionUri(databaseName, collectionName),
            "SELECT * FROM Family WHERE Family.LastName = 'Andersen'",
            queryOptions);

        Console.WriteLine("Running direct SQL query...");
        foreach (Family family in familyQueryInSql)
        {
            Console.WriteLine("\tRead {0}", family);
        }

        Console.WriteLine("Press any key to continue ...");
        Console.ReadKey();
    }

    //REPLACE JSON DOCUMENT
    private async Task ReplaceFamilyDocument(string databaseName, string collectionName, string familyName,
Family updatedFamily)
    {
        await this.client.ReplaceDocumentAsync(UriFactory.CreateDocumentUri(databaseName, collectionName,
familyName), updatedFamily);
        this.WriteToConsoleAndPromptToContinue("Replaced Family {0}", familyName);
```

```
    }

    //DELETE JSON DOCUMENT
    private async Task DeleteFamilyDocument(string databaseName, string collectionName, string documentName)
    {
        await this.client.DeleteDocumentAsync(UriFactory.CreateDocumentUri(databaseName, collectionName,
documentName));
        Console.WriteLine("Deleted Family {0}", documentName);
    }
    private void WriteToConsoleAndPromptToContinue(string format, params object[] args)
    {
        Console.WriteLine(format, args);
        Console.WriteLine("Press any key to continue ...");
        Console.ReadKey();
    }
}
```

```
{
  "id": "AndersenFamily",
  "lastName": "Andersen",
  "parents": [
    { "firstName": "Thomas" },
    { "firstName": "Mary Kay"}
  ],
  "children": [
    {
      "firstName": "Henriette Thaulow",
      "gender": "female",
      "grade": 5,
      "pets": [{ "givenName": "Fluffy" }]
    }
  ],
  "address": { "state": "WA", "county": "King",
"city": "seattle" },
  "creationDate": 1431620472,
  "isRegistered": true
```

```
{
  "id": "WakefieldFamily",
  "parents": [
    { "familyName": "Wakefield", "givenName": "Robin" },
    { "familyName": "Miller", "givenName": "Ben" }
  ],
  "children": [
    {
      "familyName": "Merriam",
      "givenName": "Jesse",
      "gender": "female", "grade": 1,
      "pets": [
        { "givenName": "Goofy" },
        { "givenName": "Shadow" }
      ]
    },
    {
      "familyName": "Miller",
      "givenName": "Lisa",
```

}	"gender": "female", "grade": 8 }], "address": { "state": "NY", "county": "Manhattan", "city": "NY" }, "creationDate": 1431620462, "isRegistered": false }

More Queries and Results:

SELECT * FROM Families f WHERE f.id = "AndersenFamily"

Fetches only one Entity in array.

SELECT {"Name":f.id, "City":f.address.city} **AS Family**
FROM Families f
WHERE f.address.city = f.address.state
	[{
	"Family": {
	"Name": "WakefieldFamily",
	"City": "NY"
	}
	}]

SELECT c.givenName FROM Families f JOIN c IN f.children WHERE f.id = 'WakefieldFamily' ORDER BY f.address.city ASC	[{ "givenName": "Jesse" }, { "givenName": "Lisa"}]

SELECT f.**address**
FROM Families f
WHERE f.id = "AndersenFamily"
	[{
	"**address**": {
	"state": "WA",
	"county": "King",
	"city": "seattle"
	}
	}]

SELECT f.address.**state**, f.address.**city** FROM Families f WHERE f.id = "AndersenFamily"	[{ "state": "WA", "city": "seattle" }]

More About Queries: https://docs.microsoft.com/en-us/azure/cosmos-db/sql-api-sql-query