Open project

Show DocumentDB nuget package

Open portal.azure.com

Create account

Data+Storage -> Azure DocumentDB (free)

Show Keys and copy paste URI and primary key into notepad

There are different ways to connect including connection string but we will use URI and key for this demo

**Availability**

99.99% Azure PaaS SLA

**Consistency (show slide)**

Talk about consistency - 4 different types,

Strong is within one region, globally eventual

Bounded similar to strong with some lag but gives global level performance

Session - strong within an user session, eventual otherwise

Eventual - if no further writes, data eventually syncs

75% use session. Started with two added bounded staleness with 90ms latency and then added session for better local perf but worse global perf

Account level consistency is global default

Can we have consistency at collection or document level?

Bounded staleness is adjustable

**Collections**

Where you store data

You can specify scaling (partitioning) and performance (throughput) at this level

**Partitions**

May span one or more partitions - number of partitions = size of data

Each partition = 10GB

Managed by Azure, completely transparent to use

Partition keys used by documents, hashed in doc to determine the partition. Same partition key = same partition

Multi-partition only available at Standard level, other levels single partition only (use standard and tweak as needed)

**Throughput**

Request Units (RUs). 1RU = reading a 1KB doc

Stored docs and throughput evenly distributed among partitions

Copy paste json into RU estimator <https://www.documentdb.com/capacityplanner>

Paste into collection

Use query explorer to run a select and where clause

Open query playground <https://www.documentdb.com/sql/demo>

**Create Document DB Account**

New -> Data + Storage -> DocumentDB

Enter Document Account Name

Show Keys – copy URI and key to notepad

Show default consistency (explain why strong or session)

Create database

Create collection

Explain throughput and partitioning and storage

After creating collection, show settings for TTL

Delete database and connection

**Create Web App**

Open Visual Studio

Open ONETUGAzureDocumentDB project

Create simple ASP.NET MVC project

Manage Nuget package ->Online -> Install Microsoft Azure DocumentDB Client Library

Highlight references Microsoft Azure DocumentDB client

Open Model folder -> Add Item Class. Copy paste code into namespace

using Newtonsoft.Json;

public class Item

{

[JsonProperty(PropertyName = "id")]

public string Id { get; set; }

[JsonProperty(PropertyName = "name")]

public string Name { get; set; }

[JsonProperty(PropertyName = "desc")]

public string Description { get; set; }

[JsonProperty(PropertyName = "isComplete")]

public bool Completed { get; set; }

}

Add empty Item controller

Copy paste Index 5 times

Change to Create, Edit, Details and Delete

Right-click on Index and Add View -> Template List -> Model class Item -> Select Layout from Shared Views

Right-click on Creat and Add View -> Template Create -> Model class Item -> Select Layout from Shared Views

Right-click on Edit and Add View -> Template Create -> Model class Item -> Select Layout from Shared Views

Right-click on Delete and Add View -> Template Create -> Model class Item -> Select Layout from Shared Views

Right-click on Details and Add View -> Template Create -> Model class Item -> Select Layout from Shared Views

**DocumentDB Repository**

Add DocumentDBRepository.cs class and add following code inside namespace:

Point out ids, document, collection and client

Point out static method to read database and collection and create them if they don’t exist

Point out

using System;

using System.Collections.Generic;

using System.Configuration;

using System.Linq;

using Microsoft.Azure.Documents;

using Microsoft.Azure.Documents.Client;

using Microsoft.Azure.Documents.Linq;

using Models;

public static class DocumentDBRepository

{

private static string databaseId;

private static string collectionId;

private static Database database;

private static DocumentCollection collection;

private static DocumentClient client;

private static string DatabaseId

{

get

{

if (string.IsNullOrEmpty(databaseId))

{

databaseId = ConfigurationManager.AppSettings["database"];

}

return databaseId;

}

}

private static string CollectionId

{

get

{

if (string.IsNullOrEmpty(collectionId))

{

collectionId = ConfigurationManager.AppSettings["collection"];

}

return collectionId;

}

}

private static Database Database

{

get

{

if (database == null)

{

database = ReadOrCreateDatabase();

}

return database;

}

}

private static DocumentCollection Collection

{

get

{

if (collection == null)

{

collection = ReadOrCreateCollection(Database.SelfLink);

}

return collection;

}

}

private static DocumentClient Client

{

get

{

if (client == null)

{

string endpoint = ConfigurationManager.AppSettings["endpoint"];

string authKey = ConfigurationManager.AppSettings["authKey"];

Uri endpointUri = new Uri(endpoint);

client = new DocumentClient(endpointUri, authKey);

}

return client;

}

}

}

private static DocumentCollection ReadOrCreateCollection(string databaseLink)

{

var col = Client.CreateDocumentCollectionQuery(databaseLink)

.Where(c => c.Id == CollectionId)

.AsEnumerable()

.FirstOrDefault();

if (col == null)

{

col = Client.CreateDocumentCollectionAsync(databaseLink, new DocumentCollection { Id = CollectionId }).Result;

}

return col;

}

private static Database ReadOrCreateDatabase()

{

var db = Client.CreateDatabaseQuery()

.Where(d => d.Id == DatabaseId)

.AsEnumerable()

.FirstOrDefault();

if (db == null)

{

db = Client.CreateDatabaseAsync(new Database { Id = DatabaseId }).Result;

}

return db;

}

**Config**

Point out config settings in the code and copy-paste this to web.config

<add key="endpoint" value="URI" />

<add key="authKey" value="PRIMARY KEY" />

<add key="database" value="ONETUGToDoList" />

<add key="collection" value="ONETUGItems" />

Replace values or URI and Primary key from notepad

**Read items**

Copy-paste this code and point out it simply queries for items in the database

public static List<Item> GetAllItems()

{

return Client.CreateDocumentQuery<Item>(Collection.DocumentsLink)

.AsEnumerable()

.ToList<Item>();

}

**List Page**

Add this code to Index method of Item Controller

var items = DocumentDBRepository.GetIncompleteItems();

return this.View(items);

**Route**

Open App\_Start -> routeconfig.cs and replace Home with Item

Now run project, empty list page pops up and show how database ONETUGToDoList and collection ONETUGItems were created

Jump into ReadOrCreateDatabase and ReadOrCreateCollection methods

Drill down to collection, open document explorer and show no documents

**Create Items:**

Replace create method in controller with this code, show the CreateItem method

public ActionResult Create()

{

return this.View();

}

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<ActionResult> Create([Bind(Include = "Id,Name,Description,Completed")] Item item)

{

if (ModelState.IsValid)

{

await DocumentDBRepository.CreateItemAsync(item);

return this.RedirectToAction("Index");

}

return this.View(item);

}

Add dependency for Models and Threading

Add create method to repository class

public static async Task<Document> CreateItemAsync(Item item)

{

return await Client.CreateDocumentAsync(Collection.SelfLink, item);

}

Add dependency for Threading

Run project and walk through creating a new item or two

Open Document explorer and show newly created items

Switch to query explorer and run this:

SELECT \* FROM ONETUGItems i

Then run

SELECT \* FROM ONETUGItems i

where i.name = "Azure DocumentDB talk"

Open SQL Demo <https://www.documentdb.com/sql/demo> and talk about different types of queries

Point out nested property querying under ORDER BY

Point out geospatial querying

**Editing**

Add this code to controller and add system net reference

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<ActionResult> Edit([Bind(Include = "Id,Name,Description,Completed")] Item item)

{

if (ModelState.IsValid)

{

await DocumentDBRepository.UpdateItemAsync(item);

return this.RedirectToAction("Index");

}

return this.View(item);

}

public ActionResult Edit(string id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

Item item = (Item)DocumentDBRepository.GetItem(id);

if (item == null)

{

return this.HttpNotFound();

}

return this.View(item);

}

Add getitem, update item and get document methods to repository class

public static Item GetItem(string id)

{

return Client.CreateDocumentQuery<Item>(Collection.DocumentsLink)

.Where(d => d.Id == id)

.AsEnumerable()

.FirstOrDefault();

}

public static Document GetDocument(string id)

{

return Client.CreateDocumentQuery(Collection.DocumentsLink)

.Where(d => d.Id == id)

.AsEnumerable()

.FirstOrDefault();

}

public static async Task<Document> UpdateItemAsync(Item item)

{

Document doc = GetDocument(item.Id);

return await Client.ReplaceDocumentAsync(doc.SelfLink, item);

}

Run the project and show Edit by checking a checkbox

**Delete**

Add this code to the Item Controller and point out delete methods

public ActionResult Delete(string id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

Item item = (Item)DocumentDBRepository.GetItem(id);

if (item == null)

{

return this.HttpNotFound();

}

return this.View(item);

}

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<ActionResult> DeleteConfirmed([Bind(Include = "Id")] string id)

{

await DocumentDBRepository.DeleteItemAsync(id);

return this.RedirectToAction("Index");

}

Add this method to the repository class

public static async Task DeleteItemAsync(string id)

{

Document doc = GetDocument(id);

await Client.DeleteDocumentAsync(doc.SelfLink);

}

Run project and delete an item and show it disappearing in the query explorer

**Details**

public ActionResult Details(string id)

{

var item = DocumentDBRepository.GetItem(id);

return this.View(item);

}