# Working with Geospatial data in Azure DocumentDB

This article is an introduction to the geospatial functionality in [Azure DocumentDB](https://azure.microsoft.com/services/documentdb/). After reading this, you will be able to answer the following questions:

* How do I store spatial data in Azure DocumentDB?
* How can I query geospatial data in Azure DocumentDB in SQL and LINQ?
* How do I enable or disable spatial indexing in DocumentDB?

Please see this [Github project](https://github.com/Azure/azure-documentdb-dotnet/blob/master/samples/code-samples/Geospatial/Program.cs) for code samples.

## Introduction to spatial data

Spatial data describes the position and shape of objects in space. In most applications, these correspond to objects on the earth, i.e. geospatial data. Spatial data can be used to represent the location of a person, a place of interest, or the boundary of a city, or a lake. Common use cases often involve proximity queries, for e.g., “find all coffee shops near my current location”.

### GeoJSON

DocumentDB supports indexing and querying of geospatial point data that’s represented using the [GeoJSON specification](http://geojson.org/geojson-spec.html). GeoJSON data structures are always valid JSON objects, so they can be stored and queried using DocumentDB without any specialized tools or libraries. The DocumentDB SDKs provide helper classes and methods that make it easy to work with spatial data.

### Points, linestrings and polygons

A **Point** denotes a single position in space. In geospatial data, a point represents the exact location, which could be a street address of a grocery store, a kiosk, an automobile or a city. A point is represented in GeoJSON (and DocumentDB) using its coordinate pair or longitude and latitude. Here’s an example JSON for a point.

**Points in DocumentDB**

{  
 "type":"Point",  
 "coordinates":[ 31.9, -4.8 ]  
}

[AZURE.NOTE] The GeoJSON specification specifies longitude first and latitude second. Like in other mapping applications, longitude and latitude are angles and represented in terms of degrees. Longitude values are measured from the Prime Meridian and are between -180 and 180.0 degrees, and latitude values are measured from the equator and are between -90.0 and 90.0 degrees.

DocumentDB interprets coordinates as represented per the WGS-84 reference system. Please see below for more details about coordinate reference systems.

This can be embedded in a DocumentDB document as shown in this example of a user profile containing location data:

**Use Profile with Location stored in DocumentDB**

{  
 "id":"documentdb-profile",  
 "screen\_name":"@DocumentDB",  
 "city":"Redmond",  
 "topics":[ "NoSQL", "Javascript" ],  
 "location":{  
 "type":"Point",  
 "coordinates":[ 31.9, -4.8 ]  
 }  
}

In addition to points, GeoJSON also supports LineStrings and Polygons. **LineStrings** represent a series of two or more points in space and the line segments that connect them. In geospatial data, linestrings are commonly used to represent highways or rivers. A **polygon** is a boundary of connected points that forms a closed LineString. Polygons are commonly used to represent natural formations like lakes or political jurisdictions like cities and states. Here’s an example of a polygon in DocumentDB.

**Polygons in DocumentDB**

{  
 "type":"Polygon",  
 "coordinates":[  
 [ 31.8, -5 ],  
 [ 31.8, -4.7 ],  
 [ 32, -4.7 ],  
 [ 32, -5 ],  
 [ 31.8, -5 ]  
 ]  
}

[AZURE.NOTE] The GeoJSON specification requires that for valid polygons, the last coordinate pair provided should be the same as the first, to create a closed shape.

Points within a polygon must be specified in counter-clockwise order. A polygon specified in clockwise order represents the inverse of the region within it.

In addition to Point, LineString and Polygon, GeoJSON also specifies the representation for how to group multiple geospatial locations, as well as how to associate arbitrary properties with geolocation as a **Feature**. Since these objects are valid JSON, they can all be stored and processed in DocumentDB. However DocumentDB only supports automatic indexing of points.

### Coordinate reference systems

Since the shape of the earth is irregular, coordinates of geospatial data is represented in many coordinate reference systems (CRS), each with their own frames of reference and units of measurement. For example, the “National Grid of Britain” is a reference system is very accurate for the United Kingdom, but not outside it.

The most popular CRS in use today is the World Geodetic System [WGS-84](http://earth-info.nga.mil/GandG/wgs84/). GPS devices, and many mapping services including Google Maps and Bing Maps APIs use WGS-84. DocumentDB supports indexing and querying of geospatial data using the WGS-84 CRS only.

## Creating documents with spatial data

When you create documents that contain GeoJSON values, they are automatically indexed with a spatial index in accordance to the indexing policy of the collection. If you’re working with a DocumentDB SDK in a dynamically typed language like Python or Node.js, you must create valid GeoJSON.

**Create Document with Geospatial data in Node.js**

var userProfileDocument = {  
 "name":"documentdb",  
 "location":{  
 "type":"Point",  
 "coordinates":[ -122.12, 47.66 ]  
 }  
};  
  
client.createDocument(`dbs/${databaseName}/colls/${collectionName}`, userProfileDocument, (err, created) => {  
 // additional code within the callback  
});

If you’re working with the .NET (or Java) SDKs, you can use the new Point and Polygon classes within the Microsoft.Azure.Documents.Spatial namespace to embed location information within your application objects. These classes help simplify the serialization and deserialization of spatial data into GeoJSON.

**Create Document with Geospatial data in .NET**

using Microsoft.Azure.Documents.Spatial;  
  
public class UserProfile  
{  
 [JsonProperty("name")]  
 public string Name { get; set; }  
  
 [JsonProperty("location")]  
 public Point Location { get; set; }  
   
 // More properties  
}  
  
await client.CreateDocumentAsync(  
 UriFactory.CreateDocumentCollectionUri("db", "profiles"),   
 new UserProfile   
 {   
 Name = "documentdb",   
 Location = new Point (-122.12, 47.66)   
 });

If you don’t have the latitude and longitude information, but have the physical addresses or location name like city or country, you can look up the actual coordinates by using a geocoding service like Bing Maps REST Services. Learn more about Bing Maps geocoding [here](https://msdn.microsoft.com/library/ff701713.aspx).

## Querying spatial types

Now that we’ve taken a look at how to insert geospatial data, let’s take a look at how to query this data using DocumentDB using SQL and LINQ.

### Spatial SQL built-in functions

DocumentDB supports the following Open Geospatial Consortium (OGC) built-in functions for geospatial querying. For more details on the complete set of built-in functions in the SQL language, please refer to [Query DocumentDB](/documentation/articles/documentdb-sql-query).

Usage

Description

ST\_DISTANCE (point\_expr, point\_expr)

Returns the distance between the two GeoJSON point expressions.

ST\_WITHIN (point\_expr, polygon\_expr)

Returns a Boolean expression indicating whether the GeoJSON point specified in the first argument is within the GeoJSON polygon in the second argument.

ST\_ISVALID

Returns a Boolean value indicating whether the specified GeoJSON point or polygon expression is valid.

ST\_ISVALIDDETAILED

Returns a JSON value containing a Boolean value if the specified GeoJSON point or polygon expression is valid, and if invalid, additionally the reason as a string value.

Spatial functions can be used to perform proximity queries against spatial data. For example, here’s a query that returns all family documents that are within 30 km of the specified location using the ST\_DISTANCE built-in function.

**Query**

SELECT f.id   
FROM Families f   
WHERE ST\_DISTANCE(f.location, {'type': 'Point', 'coordinates':[31.9, -4.8]}) < 30000

**Results**

[{  
 "id": "WakefieldFamily"  
}]

If you include spatial indexing in your indexing policy, then “distance queries” will be served efficiently through the index. For more details on spatial indexing, please see the section below. If you don’t have a spatial index for the specified paths, you can still perform spatial queries by specifying x-ms-documentdb-query-enable-scan request header with the value set to “true”. In .NET, this can be done by passing the optional **FeedOptions** argument to queries with [EnableScanInQuery](https://msdn.microsoft.com/library/microsoft.azure.documents.client.feedoptions.enablescaninquery.aspx#P:Microsoft.Azure.Documents.Client.FeedOptions.EnableScanInQuery) set to true.

ST\_WITHIN can be used to check if a point lies within a polygon. Commonly polygons are used to represent boundaries like zip codes, state boundaries, or natural formations. Again if you include spatial indexing in your indexing policy, then “within” queries will be served efficiently through the index.

Polygon arguments in ST\_WITHIN can contain only a single ring, i.e. the polygons must not contain holes in them. Check the [DocumentDB limits](/documentation/articles/documentdb-limits) for the maximum number of points allowed in a polygon for an ST\_WITHIN query.

**Query**

SELECT \*   
FROM Families f   
WHERE ST\_WITHIN(f.location, {  
 'type':'Polygon',   
 'coordinates': [[[31.8, -5], [32, -5], [32, -4.7], [31.8, -4.7], [31.8, -5]]]  
})

**Results**

[{  
 "id": "WakefieldFamily",  
}]

[AZURE.NOTE] Similar to how mismatched types works in DocumentDB query, if the location value specified in either argument is malformed or invalid, then it will evaluate to **undefined** and the evaluated document to be skipped from the query results. If your query returns no results, run ST\_ISVALIDDETAILED To debug why the spatail type is invalid.

ST\_ISVALID and ST\_ISVALIDDETAILED can be used to check if a spatial object is valid. For example, the following query checks the validity of a point with an out of range latitude value (-132.8). ST\_ISVALID returns just a Boolean value, and ST\_ISVALIDDETAILED returns the Boolean and a string containing the reason why it is considered invalid.

\*\* Query \*\*

SELECT ST\_ISVALID({ "type": "Point", "coordinates": [31.9, -132.8] })

**Results**

[{  
 "$1": false  
}]

These functions can also be used to validate polygons. For example, here we use ST\_ISVALIDDETAILED to validate a polygon that is not closed.

**Query**

SELECT ST\_ISVALIDDETAILED({ "type": "Polygon", "coordinates": [[   
 [ 31.8, -5 ], [ 31.8, -4.7 ], [ 32, -4.7 ], [ 32, -5 ]   
 ]]})

**Results**

[{  
 "$1": {   
 "valid": false,   
 "reason": "The Polygon input is not valid because the start and end points of the ring number 1 are not the same. Each ring of a polygon must have the same start and end points."   
 }  
}]

### LINQ Querying in the .NET SDK

The DocumentDB .NET SDK also providers stub methods Distance() and Within() for use within LINQ expressions. The DocumentDB LINQ provider translates these method calls to the equivalent SQL built-in function calls (ST\_DISTANCE and ST\_WITHIN respectively).

Here’s an example of a LINQ query that finds all documents in the DocumentDB collection whose “location” value is within a radius of 30km of the specified point using LINQ.

**LINQ query for Distance**

foreach (UserProfile user in client.CreateDocumentQuery<UserProfile>(UriFactory.CreateDocumentCollectionUri("db", "profiles"))  
 .Where(u => u.ProfileType == "Public" && a.Location.Distance(new Point(32.33, -4.66)) < 30000))  
{  
 Console.WriteLine("\t" + user);  
}

Similarly, here’s a query for finding all the documents whose “location” is within the specified box/polygon.

**LINQ query for Within**

Polygon rectangularArea = new Polygon(  
 new[]  
 {  
 new LinearRing(new [] {  
 new Position(31.8, -5),  
 new Position(32, -5),  
 new Position(32, -4.7),  
 new Position(31.8, -4.7),  
 new Position(31.8, -5)  
 })  
 });  
  
foreach (UserProfile user in client.CreateDocumentQuery<UserProfile>(UriFactory.CreateDocumentCollectionUri("db", "profiles"))  
 .Where(a => a.Location.Within(rectangularArea)))  
{  
 Console.WriteLine("\t" + user);  
}

Now that we’ve taken a look at how to query documents using LINQ and SQL, let’s take a look at how to configure DocumentDB for spatial indexing.

## Indexing

As we described in the [Schema Agnostic Indexing with Azure DocumentDB](http://www.vldb.org/pvldb/vol8/p1668-shukla.pdf) paper, we designed DocumentDB’s database engine to be truly schema agnostic and provide first class support for JSON. The write optimized database engine of DocumentDB now also natively understands spatial data represented in the GeoJSON standard.

In a nutshell, the geometry is projected from geodetic coordinates onto a 2D plane then divided progressively into cells using a **quadtree**. These cells are mapped to 1D based on the location of the cell within a **Hilbert space filling curve**, which preserves locality of points. Additionally when location data is indexed, it goes through a process known as **tessellation**, i.e. all the cells that intersect a location are identified and stored as keys in the DocumentDB index. At query time, arguments like points and polygons are also tessellated to extract the relevant cell ID ranges, then used to retrieve data from the index.

If you specify an indexing policy that includes spatial index for /\* (all paths), then all points found within the collection are indexed for efficient spatial queries (ST\_WITHIN and ST\_DISTANCE). Spatial indexes do not have a precision value, and always use a default precision value.

The following JSON snippet shows an indexing policy with spatial indexing enabled, i.e. index any GeoJSON point found within documents for spatial querying. If you are modifying the indexing policy using the Azure Portal, you can specify the following JSON for indexing policy to enable spatial indexing on your collection.

**Collection Indexing Policy JSON with Spatial enabled**

{  
 "automatic":true,  
 "indexingMode":"Consistent",  
 "includedPaths":[  
 {  
 "path":"/\*",  
 "indexes":[  
 {  
 "kind":"Range",  
 "dataType":"String",  
 "precision":-1  
 },  
 {  
 "kind":"Range",  
 "dataType":"Number",  
 "precision":-1  
 },  
 {  
 "kind":"Spatial",  
 "dataType":"Point"  
 }  
 ]  
 }  
 ],  
 "excludedPaths":[  
 ]  
}

Here’s a code snippet in .NET that shows how to create a collection with spatial indexing turned on for all paths containing points.

**Create a collection with spatial indexing**

DocumentCollection spatialData = new DocumentCollection()  
spatialData.IndexingPolicy = new IndexingPolicy(new SpatialIndex(DataType.Point)); //override to turn spatial on by default  
collection = await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("db"), spatialData);

And here’s how you can modify an existing collection to take advantage of spatial indexing over any points that are stored within documents.

**Modify an existing collection with spatial indexing**

Console.WriteLine("Updating collection with spatial indexing enabled in indexing policy...");  
collection.IndexingPolicy = new IndexingPolicy(new SpatialIndex(DataType.Point));  
await client.ReplaceDocumentCollectionAsync(collection);  
  
Console.WriteLine("Waiting for indexing to complete...");  
long indexTransformationProgress = 0;  
while (indexTransformationProgress < 100)  
{  
 ResourceResponse<DocumentCollection> response = await client.ReadDocumentCollectionAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"));  
 indexTransformationProgress = response.IndexTransformationProgress;  
  
 await Task.Delay(TimeSpan.FromSeconds(1));  
}

[AZURE.NOTE] If the location GeoJSON value within the document is malformed or invalid, then it will not get indexed for spatial querying. You can validate location values using ST\_ISVALID and ST\_ISVALIDDETAILED.

If your collection definition includes a partition key, indexing transformation progress is not reported.

## Next steps

Now that you’ve learnt about how to get started with geospatial support in DocumentDB, you can:

* Start coding with the [Geospatial .NET code samples on Github](https://github.com/Azure/azure-documentdb-dotnet/blob/e880a71bc03c9af249352cfa12997b51853f47e5/samples/code-samples/Geospatial/Program.cs)
* Get hands on with geospatial querying at the [DocumentDB Query Playground](http://www.documentdb.com/sql/demo#geospatial)
* Learn more about [DocumentDB Query](/documentation/articles/documentdb-sql-query)
* Learn more about [DocumentDB Indexing Policies](/documentation/articles/documentdb-indexing-policies)