# NoSQL tutorial: Build a DocumentDB C# console application

[AZURE.SELECTOR] - [.NET](/documentation/articles/documentdb-get-started) - [Node.js](/documentation/articles/documentdb-nodejs-get-started)

Welcome to the NoSQL tutorial for the DocumentDB .NET SDK! After following this tutorial, you’ll have a console application that creates and queries DocumentDB resources.

We’ll cover:

* Creating and connecting to a DocumentDB 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
* Deleting the database

Don’t have time? Don’t worry! The complete solution is available on [GitHub](https://github.com/Azure-Samples/documentdb-dotnet-getting-started). Jump to the [Get the complete solution section](#GetSolution) for quick instructions.

Afterwards, please use the voting buttons at the top or bottom of this page to give us feedback. If you’d like us to contact you directly, feel free to include your email address in your comments.

Now let’s get started!

## Prerequisites

Please make sure you have the following:

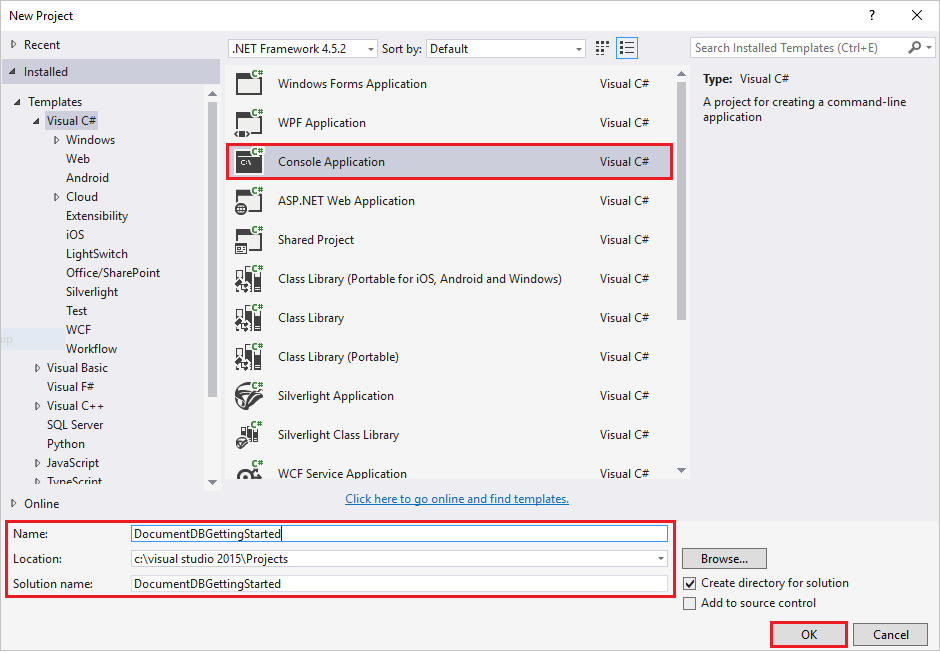
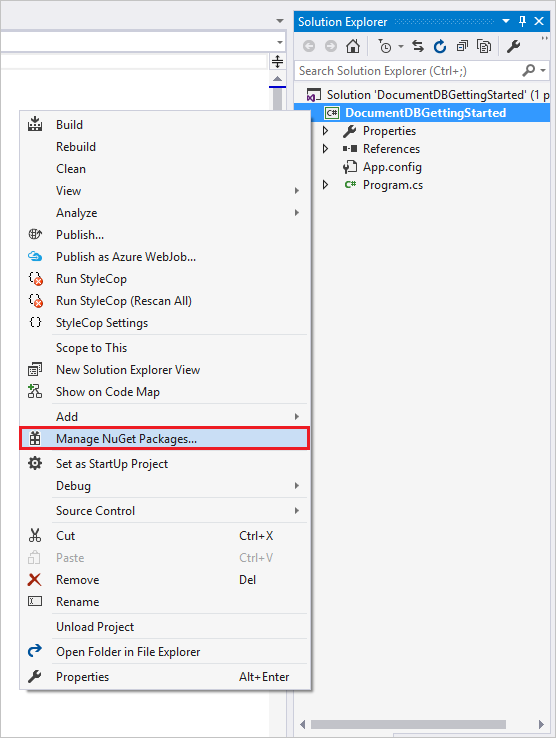
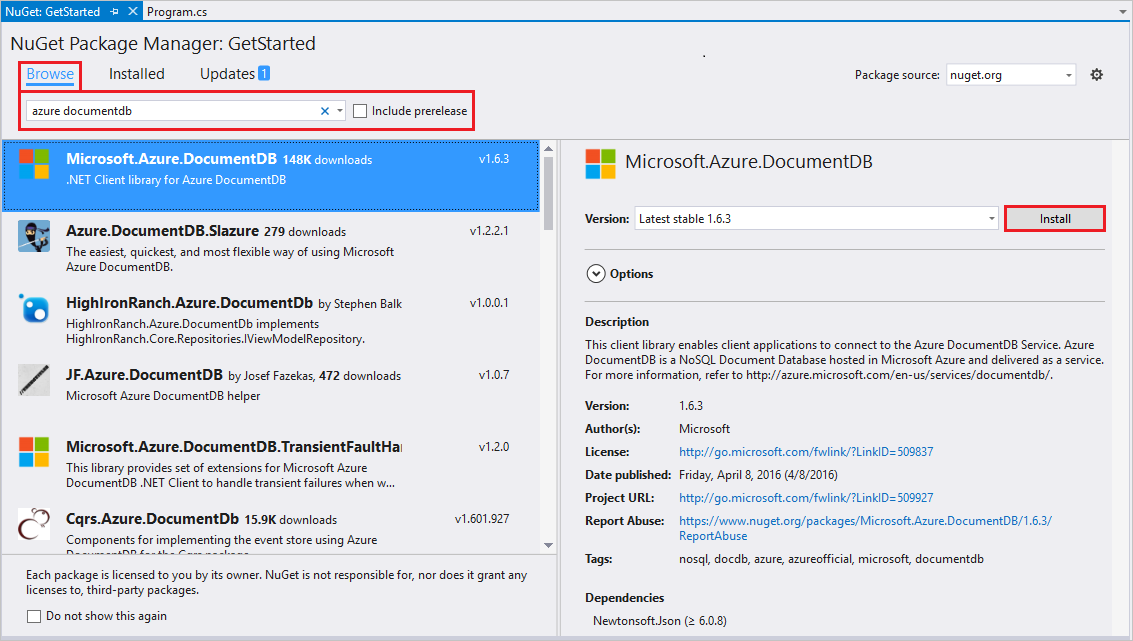
* An active Azure account. If you don’t have one, you can sign up for a [Free Trial](https://azure.microsoft.com/pricing/free-trial/).
* [Visual Studio 2013 / Visual Studio 2015](http://www.visualstudio.com/).
* .NET Framework 4.6

## Step 1: Create a DocumentDB account

Let’s create a DocumentDB account. If you already have an account you want to use, you can skip ahead to [Setup your Visual Studio Solution](#SetupVS).

[AZURE.INCLUDE [documentdb-create-dbaccount](../includes/documentdb-create-dbaccount.md)]

## Step 2: Setup your Visual Studio solution

1. Open **Visual Studio 2015** on your computer.
2. On the **File** menu, select **New**, and then choose **Project**.
3. In the **New Project** dialog, select **Templates** / **Visual C#** / **Console Application**, name your project, and then click **OK**. 
4. In the **Solution Explorer**, right click on your new console application, which is under your Visual Studio solution.
5. Then without leaving the menu, click on **Manage NuGet Packages…** 
6. In the **Nuget** tab, click **Browse**, and type **azure documentdb** in the search box.
7. Within the results, find **Microsoft.Azure.DocumentDB** and click **Install**. The package ID for the DocumentDB Client Library is [Microsoft.Azure.DocumentDB](https://www.nuget.org/packages/Microsoft.Azure.DocumentDB) 

Great! Now that we finished the setup, let’s start writing some code. You can find a completed code project of this tutorial at [GitHub](https://github.com/Azure-Samples/documentdb-dotnet-getting-started/blob/master/src/Program.cs).

## Step 3: Connect to a DocumentDB account

First, add these references to the beginning of your C# application, in the Program.cs file:

using System;  
using System.Linq;  
using System.Threading.Tasks;  
  
// ADD THIS PART TO YOUR CODE  
using System.Net;  
using Microsoft.Azure.Documents;  
using Microsoft.Azure.Documents.Client;  
using Newtonsoft.Json;

[AZURE.IMPORTANT] In order to complete this NoSQL tutorial, make sure you add the dependencies above.

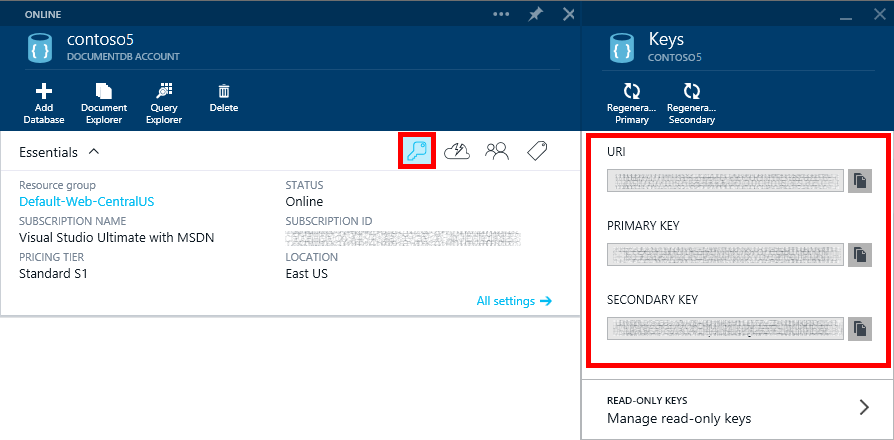
Now, add these two constants and your *client* variable underneath your public class *Program*.

public class Program  
{  
 // ADD THIS PART TO YOUR CODE  
 private const string EndpointUri = "<your endpoint URI>";  
 private const string PrimaryKey = "<your key>";  
 private DocumentClient client;

Next, head to the [Azure Portal](https://portal.azure.com) to retrieve your URI and primary key. The DocumentDB URI and primary key are necessary for your application to understand where to connect to and for DocumentDB to trust your application’s connection.

In the Azure Portal, navigate to your DocumentDB account from Step 1.

Click on the **keys** icon in the **Essentials** bar. Copy the URI and replace with the copied URI in your program. Copy the primary key and replace with the copied key in your program.



Screen shot of the Azure Portal used by the NoSQL tutorial to create a C# console application. Shows a DocumentDB account, with the ACTIVE hub highlighted, the KEYS button highlighted on the DocumentDB account blade, and the URI, PRIMARY KEY and SECONDARY KEY values highlighted on the Keys blade

We’ll start the getting started application by creating a new instance of the **DocumentClient**.

Below the **Main** method, add this new asynchronous task called **GetStartedDemo**, which will instantiate our new **DocumentClient**.

static void Main(string[] args)  
{  
}  
  
// ADD THIS PART TO YOUR CODE  
private async Task GetStartedDemo()  
{  
 this.client = new DocumentClient(new Uri(EndpointUri), PrimaryKey);  
}

Add the following code to run your asynchronous task from your **Main** method. The **Main** method will catch exceptions and write them to the console.

static void Main(string[] args)  
{  
 // ADD THIS PART TO YOUR CODE  
 try  
 {  
 Program p = new Program();  
 p.GetStartedDemo().Wait();  
 }  
 catch (DocumentClientException de)  
 {  
 Exception baseException = de.GetBaseException();  
 Console.WriteLine("{0} error occurred: {1}, Message: {2}", de.StatusCode, de.Message, baseException.Message);  
 }  
 catch (Exception e)  
 {  
 Exception baseException = e.GetBaseException();  
 Console.WriteLine("Error: {0}, Message: {1}", e.Message, baseException.Message);  
 }  
 finally  
 {  
 Console.WriteLine("End of demo, press any key to exit.");  
 Console.ReadKey();  
 }

Press **F5** to run your application.

Congratulations! You have successfully connected to a DocumentDB account, let’s now take a look at working with DocumentDB resources.

## Step 4: Create a database

Before you add the code for creating a database, add a helper method for writing to the console.

Copy and paste the **WriteToConsoleAndPromptToContinue** method underneath the **GetStartedDemo** method.

// ADD THIS PART TO YOUR CODE  
private void WriteToConsoleAndPromptToContinue(string format, params object[] args)  
{  
 Console.WriteLine(format, args);  
 Console.WriteLine("Press any key to continue ...");  
 Console.ReadKey();  
}

Your DocumentDB [database](documentdb-resources.md#databases) can be created by using the [CreateDatabaseAsync](https://msdn.microsoft.com/library/microsoft.azure.documents.client.documentclient.createdatabaseasync.aspx) method of the **DocumentClient** class. A database is the logical container of JSON document storage partitioned across collections.

Copy and paste the **CreateDatabaseIfNotExists** method underneath the **WriteToConsoleAndPromptToContinue** method.

// ADD THIS PART TO YOUR CODE  
private async Task CreateDatabaseIfNotExists(string databaseName)  
{  
 // Check to verify a database with the id=FamilyDB does not exist  
 try  
 {  
 await this.client.ReadDatabaseAsync(UriFactory.CreateDatabaseUri(databaseName));  
 this.WriteToConsoleAndPromptToContinue("Found {0}", databaseName);  
 }  
 catch (DocumentClientException de)  
 {  
 // If the database does not exist, create a new database  
 if (de.StatusCode == HttpStatusCode.NotFound)  
 {  
 await this.client.CreateDatabaseAsync(new Database { Id = databaseName });  
 this.WriteToConsoleAndPromptToContinue("Created {0}", databaseName);  
 }  
 else  
 {  
 throw;  
 }  
 }  
}

Copy and paste the following code to your **GetStartedDemo** method underneath the client creation. This will create a database named *FamilyDB*.

private async Task GetStartedDemo()  
{  
 this.client = new DocumentClient(new Uri(EndpointUri), PrimaryKey);  
  
 // ADD THIS PART TO YOUR CODE  
 await this.CreateDatabaseIfNotExists("FamilyDB");

Press **F5** to run your application.

Congratulations! You have successfully created a DocumentDB database.

## Step 5: Create a collection

[AZURE.WARNING] **CreateDocumentCollectionAsync** will create a new collection with reserved throughput, which has pricing implications. For more details, please visit our [pricing page](https://azure.microsoft.com/pricing/details/documentdb/).

A [collection](documentdb-resources.md#collections) can be created by using the [CreateDocumentCollectionAsync](https://msdn.microsoft.com/library/microsoft.azure.documents.client.documentclient.createdocumentcollectionasync.aspx) method of the **DocumentClient** class. A collection is a container of JSON documents and associated JavaScript application logic.

Copy and paste the **CreateDocumentCollectionIfNotExists** method underneath your **CreateDatabaseIfNotExists** method.

// ADD THIS PART TO YOUR CODE  
private async Task CreateDocumentCollectionIfNotExists(string databaseName, string collectionName)  
{  
 try  
 {  
 await this.client.ReadDocumentCollectionAsync(UriFactory.CreateDocumentCollectionUri(databaseName, collectionName));  
 this.WriteToConsoleAndPromptToContinue("Found {0}", collectionName);  
 }  
 catch (DocumentClientException de)  
 {  
 // If the document collection does not exist, create a new collection  
 if (de.StatusCode == HttpStatusCode.NotFound)  
 {  
 DocumentCollection collectionInfo = new DocumentCollection();  
 collectionInfo.Id = collectionName;  
  
 // Configure collections for maximum query flexibility including string range queries.  
 collectionInfo.IndexingPolicy = new IndexingPolicy(new RangeIndex(DataType.String) { Precision = -1 });  
  
 // Here we create a collection with 400 RU/s.  
 await this.client.CreateDocumentCollectionAsync(  
 UriFactory.CreateDatabaseUri(databaseName),  
 new DocumentCollection { Id = collectionName },  
 new RequestOptions { OfferThroughput = 400 });  
  
 this.WriteToConsoleAndPromptToContinue("Created {0}", collectionName);  
 }  
 else  
 {  
 throw;  
 }  
 }  
}

Copy and paste the following code to your **GetStartedDemo** method underneath the database creation. This will create a document collection named *FamilyCollection*.

this.client = new DocumentClient(new Uri(EndpointUri), PrimaryKey);  
  
 await this.CreateDatabaseIfNotExists("FamilyDB");  
  
 // ADD THIS PART TO YOUR CODE  
 await this.CreateDocumentCollectionIfNotExists("FamilyDB", "FamilyCollection");

Press **F5** to run your application.

Congratulations! You have successfully created a DocumentDB document collection.

## Step 6: Create JSON documents

A [document](/documentation/articles/documentdb-resources#documents) can be created by using the [CreateDocumentAsync](https://msdn.microsoft.com/library/microsoft.azure.documents.client.documentclient.createdocumentasync.aspx) method of the **DocumentClient** class. Documents are user defined (arbitrary) JSON content. We can now insert one or more documents. If you already have data you’d like to store in your database, you can use DocumentDB’s [Data Migration tool](/documentation/articles/documentdb-import-data).

First, we need to create a **Family** class that will represent objects stored within DocumentDB in this sample. We will also create **Parent**, **Child**, **Pet**, **Address** subclasses that are used within **Family**. Note that documents must have an **Id** property serialized as **id** in JSON. Create these classes by adding the following internal sub-classes after the **GetStartedDemo** method.

Copy and paste the **Family**, **Parent**, **Child**, **Pet**, and **Address** classes underneath the **WriteToConsoleAndPromptToContinue** method.

private void WriteToConsoleAndPromptToContinue(string format, params object[] args)  
{  
 Console.WriteLine(format, args);  
 Console.WriteLine("Press any key to continue ...");  
 Console.ReadKey();  
}  
  
// ADD THIS PART TO YOUR CODE  
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; }  
}

Copy and paste the **CreateFamilyDocumentIfNotExists** method underneath your **CreateDocumentCollectionIfNotExists** method.

// ADD THIS PART TO YOUR CODE  
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;  
 }  
 }  
}

And insert two documents, one each for the Andersen Family and the Wakefield Family.

Copy and paste the following code to your **GetStartedDemo** method underneath the document collection creation.

await this.CreateDatabaseIfNotExists("FamilyDB");  
  
await this.CreateDocumentCollectionIfNotExists("FamilyDB", "FamilyCollection");  
  
// ADD THIS PART TO YOUR CODE  
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 = "Henriette Thaulow",  
 Gender = "female",  
 Grade = 5,  
 Pets = new Pet[]  
 {  
 new Pet { GivenName = "Fluffy" }  
 }  
 }  
 },  
 Address = new Address { State = "WA", County = "King", City = "Seattle" },  
 IsRegistered = true  
};  
  
await this.CreateFamilyDocumentIfNotExists("FamilyDB", "FamilyCollection", andersenFamily);  
  
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  
};  
  
await this.CreateFamilyDocumentIfNotExists("FamilyDB", "FamilyCollection", wakefieldFamily);

Press **F5** to run your application.

Congratulations! You have successfully created two DocumentDB documents.

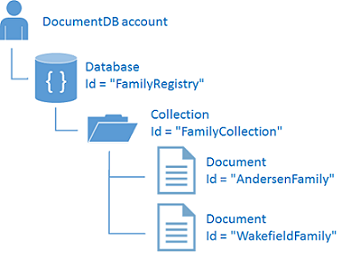


Diagram illustrating the hierarchical relationship between the account, the online database, the collection, and the documents used by the NoSQL tutorial to create a C# console application

## Step 7: Query DocumentDB resources

DocumentDB supports rich [queries](/documentation/articles/documentdb-sql-query) against JSON documents stored in each collection. The following sample code shows various queries - using both DocumentDB SQL syntax as well as LINQ - that we can run against the documents we inserted in the previous step.

Copy and paste the **ExecuteSimpleQuery** method underneath your **CreateFamilyDocumentIfNotExists** method.

// ADD THIS PART TO YOUR CODE  
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();  
}

Copy and paste the following code to your **GetStartedDemo** method underneath the second document creation.

await this.CreateFamilyDocumentIfNotExists("FamilyDB", "FamilyCollection", wakefieldFamily);  
  
// ADD THIS PART TO YOUR CODE  
this.ExecuteSimpleQuery("FamilyDB", "FamilyCollection");

Press **F5** to run your application.

Congratulations! You have successfully queried against a DocumentDB collection.

The following diagram illustrates how the DocumentDB SQL query syntax is called against the collection you created, and the same logic applies to the LINQ query as well.

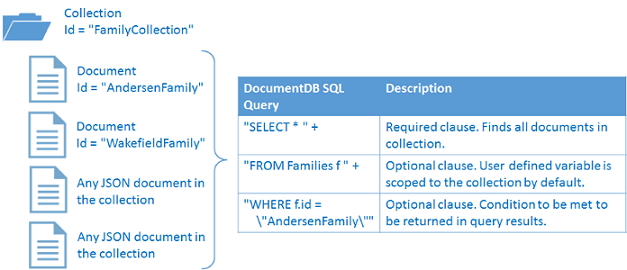


Diagram illustrating the scope and meaning of the query used by the NoSQL tutorial to create a C# console application

The [FROM](/documentation/articles/documentdb-sql-query#from-clause) keyword is optional in the query because DocumentDB queries are already scoped to a single collection. Therefore, “FROM Families f” can be swapped with “FROM root r”, or any other variable name you choose. DocumentDB will infer that Families, root, or the variable name you chose, reference the current collection by default.

## Step 8: Replace JSON document

DocumentDB supports replacing JSON documents.

Copy and paste the **ReplaceFamilyDocument** method underneath your **ExecuteSimpleQuery** method.

// ADD THIS PART TO YOUR CODE  
private async Task ReplaceFamilyDocument(string databaseName, string collectionName, string familyName, Family updatedFamily)  
{  
 try  
 {  
 await this.client.ReplaceDocumentAsync(UriFactory.CreateDocumentUri(databaseName, collectionName, familyName), updatedFamily);  
 this.WriteToConsoleAndPromptToContinue("Replaced Family {0}", familyName);  
 }  
 catch (DocumentClientException de)  
 {  
 throw de;  
 }  
}

Copy and paste the following code to your **GetStartedDemo** method underneath the query execution. After replacing the document, this will run the same query again to view the changed document.

await this.CreateFamilyDocumentIfNotExists("FamilyDB", "FamilyCollection", wakefieldFamily);  
  
this.ExecuteSimpleQuery("FamilyDB", "FamilyCollection");  
  
// ADD THIS PART TO YOUR CODE  
// Update the Grade of the Andersen Family child  
andersenFamily.Children[0].Grade = 6;  
  
await this.ReplaceFamilyDocument("FamilyDB", "FamilyCollection", "Andersen.1", andersenFamily);  
  
this.ExecuteSimpleQuery("FamilyDB", "FamilyCollection");

Press **F5** to run your application.

Congratulations! You have successfully replaced a DocumentDB document.

## Step 9: Delete JSON document

DocumentDB supports deleting JSON documents.

Copy and paste the **DeleteFamilyDocument** method underneath your **ReplaceFamilyDocument** method.

// ADD THIS PART TO YOUR CODE  
private async Task DeleteFamilyDocument(string databaseName, string collectionName, string documentName)  
{  
 try  
 {  
 await this.client.DeleteDocumentAsync(UriFactory.CreateDocumentUri(databaseName, collectionName, documentName));  
 Console.WriteLine("Deleted Family {0}", documentName);  
 }  
 catch (DocumentClientException de)  
 {  
 throw de;  
 }  
}

Copy and paste the following code to your **GetStartedDemo** method underneath the second query execution.

await this.ReplaceFamilyDocument("FamilyDB", "FamilyCollection", "Andersen.1", andersenFamily);  
  
this.ExecuteSimpleQuery("FamilyDB", "FamilyCollection");  
  
// ADD THIS PART TO CODE  
await this.DeleteFamilyDocument("FamilyDB", "FamilyCollection", "Andersen.1");

Press **F5** to run your application.

Congratulations! You have successfully deleted a DocumentDB document.

## Step 10: Delete the database

Deleting the created database will remove the database and all children resources (collections, documents, etc.).

Copy and paste the following code to your **GetStartedDemo** method underneath the document delete to delete the entire database and all children resources.

this.ExecuteSimpleQuery("FamilyDB", "FamilyCollection");  
  
await this.DeleteFamilyDocument("FamilyDB", "FamilyCollection", "Andersen.1");  
  
// ADD THIS PART TO CODE  
// Clean up/delete the database  
await this.client.DeleteDatabaseAsync(UriFactory.CreateDatabaseUri("FamilyDB"));

Press **F5** to run your application.

Congratulations! You have successfully deleted a DocumentDB database.

## Step 11: Run your C# console application all together!

Hit F5 in Visual Studio to build the application in debug mode.

You should see the output of your get started app. The output will show the results of the queries we added and should match the example text below.

Created FamilyDB  
Press any key to continue ...  
Created FamilyCollection  
Press any key to continue ...  
Created Family Andersen.1  
Press any key to continue ...  
Created Family Wakefield.7  
Press any key to continue ...  
Running LINQ query...  
 Read {"id":"Andersen.1","LastName":"Andersen","District":"WA5","Parents":[{"FamilyName":null,"FirstName":"Thomas"},{"FamilyName":null,"FirstName":"Mary Kay"}],"Children":[{"FamilyName":null,"FirstName":"Henriette Thaulow","Gender":"female","Grade":5,"Pets":[{"GivenName":"Fluffy"}]}],"Address":{"State":"WA","County":"King","City":"Seattle"},"IsRegistered":true}  
Running direct SQL query...  
 Read {"id":"Andersen.1","LastName":"Andersen","District":"WA5","Parents":[{"FamilyName":null,"FirstName":"Thomas"},{"FamilyName":null,"FirstName":"Mary Kay"}],"Children":[{"FamilyName":null,"FirstName":"Henriette Thaulow","Gender":"female","Grade":5,"Pets":[{"GivenName":"Fluffy"}]}],"Address":{"State":"WA","County":"King","City":"Seattle"},"IsRegistered":true}  
Replaced Family Andersen.1  
Press any key to continue ...  
Running LINQ query...  
 Read {"id":"Andersen.1","LastName":"Andersen","District":"WA5","Parents":[{"FamilyName":null,"FirstName":"Thomas"},{"FamilyName":null,"FirstName":"Mary Kay"}],"Children":[{"FamilyName":null,"FirstName":"Henriette Thaulow","Gender":"female","Grade":6,"Pets":[{"GivenName":"Fluffy"}]}],"Address":{"State":"WA","County":"King","City":"Seattle"},"IsRegistered":true}  
Running direct SQL query...  
 Read {"id":"Andersen.1","LastName":"Andersen","District":"WA5","Parents":[{"FamilyName":null,"FirstName":"Thomas"},{"FamilyName":null,"FirstName":"Mary Kay"}],"Children":[{"FamilyName":null,"FirstName":"Henriette Thaulow","Gender":"female","Grade":6,"Pets":[{"GivenName":"Fluffy"}]}],"Address":{"State":"WA","County":"King","City":"Seattle"},"IsRegistered":true}  
Deleted Family Andersen.1  
End of demo, press any key to exit.

Congratulations! You’ve completed this NoSQL tutorial and have a working C# console application!

## Get the complete NoSQL tutorial solution

To build the GetStarted solution that contains all the samples in this article, you will need the following:

* An active Azure account. If you don’t have one, you can sign up for a [free account](/free/).
* A [DocumentDB account](/documentation/articles/documentdb-create-account).
* The [GetStarted](https://github.com/Azure-Samples/documentdb-dotnet-getting-started) solution available on GitHub.

To restore the references to the DocumentDB .NET SDK in Visual Studio, right-click the **GetStarted** solution in Solution Explorer, and then click **Enable NuGet Package Restore**. Next, in the App.config file, update the EndpointUrl and AuthorizationKey values as described in [Connect to a DocumentDB account](#Connect).

## Next steps

* Want a more complex ASP.NET MVC NoSQL tutorial? See [Build a web application with ASP.NET MVC using DocumentDB](/documentation/articles/documentdb-dotnet-application).
* Learn how to [monitor a DocumentDB account](/documentation/articles/documentdb-monitor-accounts).
* Run queries against our sample dataset in the [Query Playground](https://www.documentdb.com/sql/demo).
* Learn more about the programming model in the Develop section of the [DocumentDB documentation page](https://azure.microsoft.com/documentation/services/documentdb/).